DDD Сообщество

РЕАЛИЗАЦИЯ МЕТОДОВ ПРЕДМЕТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

ВОН ВЕРНОН

Предисловие ЭРИКА ЭВАНСА

Реализация методов предметно-ориентированного проектирования

Implementing Domain-Driven Design

Vaughn Vernon

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Реализация методов предметно-ориентированного проектирования

Вон Вернон

Москва • Санкт-Петербург • Киев
2016

ББК 32.973.26-018.2.75
В35
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция докт. физ.-мат. наук *Д.А. Клюшина*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Вернон, Вон.
В35 Реализация методов предметно-ориентированного проектирования. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 688 с. : ил. — Парал. тит. англ.
ISBN 978-5-8459-1881-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Вон Вернон

Реализация методов предметно-ориентированного проектирования

Литературный редактор *Л.Н. Красножон*
Верстка *Л.В. Чернокозинская*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 16.11.2015. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 43,0. Уч.-изд. л. 35,21.
Тираж 300 экз. Заказ № 7215

Отпечатано способом ролевой струйной печати
в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1881-9 (рус.)
ISBN 978-0-321-83457-7 (англ.)

Оглавление

Содержание

Отзывы о книге "Реализация методов предметно-ориентированного проектирования"

Написав книгу Реализация *методов предметно-ориентированного проектирования*, Вон внес важный вклад не только в литературу о предметно-ориентированном проектировании (Domain-Driven Design — DDD), но и в более широкую область архитектуры промышленных приложений. Например, в ключевых главах об архитектуре и хранилищах Вон показывает, насколько точно предметно-ориентированное проектирование соответствует постоянно растущему ряду архитектурных стилей и технологий для поддержания непрерывности существования данных в промышленных приложениях, включая SOA и REST, NoSQL и Data Grid, появившихся за десять лет, прошедших после первого издания фундаментальной книги Эрика Эванса (Eric Evans). Вон должным образом освещает "систему сдержек и противовесов" DDD — реализацию сущностей, объектов значений, агрегатов, служб, событий, фабрик и хранилищ, — сопровождая изложение многочисленными примерами и ценной информацией, являющейся результатом десятилетнего практического опыта. Короче говоря, я бы назвал эту книгу обстоятельной. Для разработчиков программного обеспечения любой квалификации, стремящихся повысить свой уровень в области проектирования и реализации предметно-ориентированных промышленных приложений с учетом лучших достижений профессиональной практики, книга *Реализация методов предметно-ориентированного проектирования* станет кладезем знаний, ценой больших усилий добытых специалистами в области DDD и архитектуры промышленных приложений за последние десятилетия.

Рэнди Стаффорд (Randy Stafford),
архитектор больших проектов,
разработчик Oracle Coherence

Предметно-ориентированное проектирование — это мощный набор интеллектуальных инструментов, которые могут оказать серьезное влияние на эффективность команды при создании систем с интенсивным программным обеспечением. Дело в том, что многие разработчики потеряли много времени, пытаясь применить эти интеллектуальные инструменты, и действительно нуждались в более конкретном руководстве. В этой книге Вон создает недостающие звенья между теорией и практикой. Он не только проливает свет на многие недооцененные элементы DDD, но и связывает между собой новые концепции, такие как разделение ответственности на команды и запросы и источники событий, которые многие практикующие

специалисты в области DDD использовали с большим успехом. Эта книга должна стать настольной для всех, кто ищет способы внедрения принципов DDD в практику.

Уди Даан (Udi Dahan),
создатель каркаса NServiceBus

В течение многих лет разработчики, стремящиеся воплотить на практике принципы DDD, просили о более конкретной помощи в их реализации. Вон выполнил превосходную работу по преодолению разрыва между теорией и практикой с акцентом на полноценную реализацию. Он рисует яркую картину того, что принципы DDD могут сделать в современных проектах, и дает много практических советов о том, как решать типичные проблемы, возникающие на протяжении жизненного цикла проекта.

Альберто Брандолини (Alberto Brandolini),
инструктор по DDD, сертифицированный Эриком Эвансом
и компанией Domain Language, Inc.

Книга *Реализация методов предметно-ориентированного проектирования* имеет замечательную особенность: она посвящена сложной и содержательной теме и раскрывает ее четко, с нюансами, весело и изящно. Книга написана в увлекательном и дружелюбном стиле. Автор выступает в роли доверительного советника, дающего полезные советы по реализации самых важных аспектов. К тому времени, когда вы закончите читать книгу, вы будете в состоянии применять все важные понятия в области DDD и делать многое другое. Когда я читал книгу, я подчеркнул много разделов... Я часто возвращаюсь к ней и рекомендую ее другим.

Пол Райнер (Paul Rayner),
главный консультант и владелец компании Virtual Genius, LLC.,
инструктор по DDD, сертифицированный Эриком Эвансом
и компанией Domain Language, Inc.,
основатель и соруководитель компании DDD Denver

Одна из важных тем, имеющих отношение к принципам DDD, которые я преподаю, касается способов соединения всех идей и компонентов в полноценную работоспособную реализацию. С появлением этой книги сообщество DDD приобрело полный справочник, который подробно описывает все эти вопросы. Книга *Реализация методов предметно-ориентированного проектирования* посвящена всем аспектам создания систем на основе принципов DDD, от мелких деталей до крупномасштабного анализа.

Это отличный справочник и превосходное дополнение к фундаментальной книге Эрика Эванса о DDD.

Патрик Фредрикссон (Patrik Fredriksson),
инструктор по DDD,
сертифицированный Эриком Эвансом
и компанией Domain Language, Inc.

Если вы заботитесь о качестве программного обеспечения — а вы должны это делать, — то книга *Реализация методов предметно-ориентированного проектирования* станет для вас сборником советов, ведущих к быстрому успеху. Книга предлагает очень простое, но строгое обсуждение стратегических и тактических шаблонов DDD, позволяющих разработчикам немедленно перейти от понимания к действию. Будущее программное обеспечение для бизнеса извлечет выгоду из четкого руководства, изложенного в этой книге.

Дейв Мюрхед (Dave Muirhead),
главный консультант компании Blue River Systems Group

Есть теоретические и практические вопросы DDD, которые должен знать каждый разработчик, и эта книга — недостающая часть головоломки. Настоятельно рекомендую!

Рикард Оберг (Rickard Öberg),
обладатель звания Java Champion
и разработчик компании Neo Technology

В книге *Реализация методов предметно-ориентированного проектирования* Вон демонстрирует нисходящий подход к DDD, перенося на передний план стратегические шаблоны, такие как ограниченный контекст и карты контекстов, и отодвигая на задний план шаблоны структурных элементов сущностей, значений и служб. В книге систематически используется анализ ситуаций, и чтобы извлечь из них максимальную пользу, вы должны будете потратить на них много времени. Но если вы сделаете это, то будете в состоянии оценить ценность применения DDD к сложной предметной области; основные моменты иллюстрируются многочисленными заметками на полях, схемами, таблицами и кодом. Таким образом, если вы хотите создать солидную систему DDD, использующую современные архитектурные стили, книга Вона будет для вас полезной.

Дэн Хейвуд (Dan Haywood),
автор книги *Domain-Driven Design with Naked Objects*

В книге используется нисходящий подход к описанию DDD, который плавно объединяет стратегические и тактические шаблоны. Теория сочетается с практическими рекомендациями по реализации современных архитектурных стилей. На протяжении всей книги Вон выделяет важность сосредоточенности на предметной области с учетом технических ограничений. В результате читатель лучше понимает роль подхода DDD, для чего он предназначен и, что еще важнее, для чего он не предназначен. Я и моя команда много раз сталкивались с проблемами при внедрении принципов DDD. Взяв на вооружение рекомендации, приведенные в книге *Реализация методов предметно-ориентированного проектирования*, мы смогли преодолеть эти проблемы и немедленно преобразовать наши усилия в бизнес-ценности.

Лев Городинский (Lev Gorodinski),
главный архитектор DrillSpot.com

Эта книга посвящается моим дорогим Николь и Тристану.
Благодарю за вашу любовь, поддержку и терпение.

Введение

В своей новой книге Вон Вернон описывает предметно-ориентированное проектирование (Domain-Driven Design — DDD) оригинальным способом, основанным на новых объяснениях понятий, новых примерах и необычном выборе тем. Я полагаю, что этот новый, альтернативный способ изложения поможет читателям понять тонкости подхода DDD, особенно такие абстрактные, как АГРЕГАТЫ[1] и ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. Помимо того, что разные люди предпочитают разные стили, тонкие абстракции трудно освоить без многократных объяснений.

Кроме того, в книге изложены достижения последних девяти лет, которые были представлены в статьях и докладах, но до сих пор не упоминались в книгах. Благодаря этому среди структурных элементов моделей наряду с СУЩНОСТЯМИ и ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ оказались СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ. Кроме того, в книге автор обсуждает архитектурный стиль БОЛЬШОЙ КОМ ГРЯЗИ и помещает его в КАРТУ КОНТЕКСТА. Он объясняет гексагональную архитектуру, которая лучше описывает то, что мы делаем, чем многоуровневая архитектура.

Впервые я ознакомился с материалом книги почти два года назад (хотя к тому времени Вон уже работал над своей книгой в течение некоторого времени). На первом совещании по DDD некоторые из нас зарегистрировались для доклада на определенные темы, которые мы считали новыми или особенно важными для сообщества, ищущего ответы на конкретные вопросы. Вон взялся написать об агрегатах и написал ряд превосходных статей о них (которые стали главой в этой книге).

Кроме того, на совещании был достигнут консенсус относительно того, что многие практики могли бы извлечь пользу из более инструктивного описания некоторых шаблонов DDD. Честный ответ на почти любой вопрос о разработке программного обеспечения звучит как "Смотря по обстоятельствам". Однако это не очень полезно для людей, которые хотят научиться применять метод. Человек, изучающий новую тему, нуждается в конкретном руководстве. Эмпирические правила нельзя применять во всех без исключения ситуациях. Обычно они либо хорошо работают, либо применяются как первое приближение. Их определенность отражает философию подхода к решению проблем. В книге Вона есть хорошее сочетание прямого совета с обсуждением компромиссов, которые мешают принимать простые решения.

Новые шаблоны, такие как СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, стали основной частью DDD, а специалисты научились применять и адаптировать эти шаблоны в новых архитектурах и технологиях. За девять лет после выхода

[1] Особенности употребления названий шаблонов в тексте объясняются автором ниже, в разделе "Руководство по использованию книги" (табл. 1).

моей книги *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем*[2] в подходе DDD появились новые концепции, а писатели выработали новые способы изложения его основных принципов. Книга Вона представляет собой наиболее полное объяснение новых представлений о применении DDD.

Эрик Эванс (Eric Evans),
Domain Language, Inc.

[2] Русский перевод: Эванс Э. *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.* : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2012. — 448 с. В дальнейшем цитаты, ссылки и номера страниц приводятся по русскоязычному изданию, например [Эванс, с. 351]. — *Примеч. ред.*

Предисловие

Вычисления показывают, что наша идея неосуществима.
Осталось только одно: осуществить ее.

Пьер-Жорж Латекоэр,
французский пионер воздухоплавания

Именно этим мы и займемся. Предметно-ориентированное проектирование программного обеспечения — слишком важное дело, чтобы оставить разработчиков без четких указаний по его успешной реализации.

Взлет-посадка

Когда я был ребенком, мой отец научился летать на небольших самолетах. Часто мы летали всей семьей. Иногда мы летали в другой аэропорт на обед, а потом возвращались. Когда у отца было меньше времени, но он хотел полетать, мы просто кружили над аэродромом, отрабатывая взлет и посадку.

Мы предпринимали и более долгие путешествия. В таких случаях у нас всегда была карта маршрута, которую папа рисовал заранее. Поскольку мы были еще детьми, нам поручали высматривать ориентиры на земле, чтобы не сбиться с пути. Это было очень увлекательно, потому что распознавать мелкие объекты на земле было довольно трудно. На самом деле я уверен, что папа всегда знал, где мы находились. У него на приборной панели были все средства ориентирования, и он имел лицензию на "слепой" полет.

Вид сверху изменил мои представления об окружающем мире. Время от времени папа и я пролетали над нашим сельским домом. Рассматривая дом с высоты в несколько сотен футов, я видел окрестности совсем по-другому. Когда папа кружил над домом, мама и мои сестры выбегали во двор, чтобы помахать нам. Я знал, что это были они, несмотря на то, что я не мог видеть их лица. Мы не могли разговаривать. Если бы я выкрикнул что-нибудь в окно самолета, то они не услышали бы меня. Я видел изгородь, отделяющую наш участок от дороги. На земле я мог бы пройти по нему, как по бревну. Сверху изгородь была похожа на тщательно переплетенные прутья. Там был огромный луг, который я каждое лето косил, проходя на косилке полосу за полосой. С воздуха я видел только море зеленого цвета, а не травинки.

Я любил эти моменты полета. Они врезались в мою память так, будто это было еще вчера. Но хотя мне очень нравилось летать, мне всегда хотелось на землю. И какими бы захватывающими ни были посадки с немедленным взлетом после

касания земли, они происходили слишком быстро, чтобы я мог почувствовать твердую почву под ногами.

Посадка с помощью DDD

Знакомство с предметно-ориентированным проектированием можно сравнить с ощущением ребенка от полета. Вид сверху — потрясающий, но иногда вещи выглядят настолько незнакомыми, что нетрудно потерять ориентиры. Перебраться из точки А в точку Б кажется нереальным. Людям, имеющим опыт работы в рамках DDD, всегда кажется, что они знают, где находятся. Они уже давно нарисовали маршрут и полностью доверяют своим навигационным приборам. Множество других людей чувствуют себя неуверенно. Им нужна возможность "приземлиться и пришвартоваться". Кроме того, им необходима карта, чтобы из точки, где они находятся, переместиться в точку, в которой они должны быть.

В книге *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем* [Эванс] Эрик Эванс сделал то, что называют бессмертной работой. Я твердо убежден, что эта книга будет настольной у разработчиков в течение многих десятилетий. Как и в других работах о шаблонах, автор поднял точку зрения на большую высоту, чтобы расширить горизонты. Однако, если мы хотим понять основы реализации DDD, возникают сложности. Кроме того, обычно мы хотим видеть больше подробных примеров. Вот если бы только мы могли приземлиться и побыть на земле немного дольше или даже просто поехать домой или в другое знакомое место!

Помимо прочего, я хотел мягко посадить читателей на землю, овладеть их самолетом и помочь вернуться домой по известному наземному маршруту. Это поможет вам понять методы реализации DDD на основе примеров, использующих знакомые инструменты и технологии. И так как ни один из нас не может оставаться дома все время, я также помогу вам путешествовать в места, где вы никогда не были, исследуя по пути новые ландшафты. Иногда путь будет крут, но, придерживаясь правильной тактики, можно безопасно взойти на гору. В этом восхождении вы узнаете об альтернативной архитектуре и шаблонах, предназначенных для интеграции многих моделей предметной области. Вы можете оказаться на неизвестной территории. Вы найдете подробное описание стратегического моделирования с многократной интеграцией и даже научитесь разрабатывать автономные службы.

Моя цель — предоставить карту, чтобы помочь вам осуществлять как короткие прогулки, так и длительные, сложные походы, получая удовольствие от окружающих вас деталей, не теряя ориентиров и не получая травм.

Изображение ландшафта и построение карты полета

Вполне очевидно, что при разработке программного обеспечения мы всегда связываем одни сущности с другими. Мы связываем объекты с базой данных или пользовательским интерфейсом, а также выполняем обратные действия. Мы связываем объекты с различными приложениями, включая те, которые могут быть использованы в других системах и приложениях. Исходя из этого, вполне естественно стремиться создать связь высокоуровневых шаблонов Эванса с реализацией.

Даже если вы уже несколько раз сталкивались с предметно-ориентированным проектированием, есть возможность извлечь из этого подхода дополнительную пользу. Иногда такой подход называют *облегченным DDD*. Мы сосредоточимся на ОБЪЕКТАХ (OBJECTS) и СЛУЖБАХ (SERVICES), возможно, предпримем смелую попытку разработать АГРЕГАТЫ (AGGREGATES) и попробуем управлять постоянным хранением АГРЕГАТОВ, используя ХРАНИЛИЩА (REPOSITORIES). Эти шаблоны всем знакомы, поэтому мы будем использовать именно их. Кроме того, попутно мы можем применить ОБЪЕКТЫ-ЗНАЧЕНИЯ (VALUE OBJECTS). Все эти шаблоны относятся к категории тактических шаблонов, которые носят скорее технический характер. Они помогают разбираться в серьезных проблемах, связанных с программным обеспечением, применяя навыки хирурга и скальпель. Однако нам надо еще очень многое узнать, чтобы овладеть этими и другими тактическими шаблонами. Я связываю их с реализацией.

Вы когда-нибудь выходили за пределы тактического моделирования? Вы когда-нибудь оказывались на "другой стороне" DDD, в области *стратегических проектных шаблонов*? Если вы никогда не использовали ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT) и КАРТУ КОНТЕКСТА (CONTEXT MAP), то, вероятно, также ничего не знаете о шаблоне ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE).

Если и существует единственное "изобретение", которое Эванс предоставил сообществу разработчиков программного обеспечения, это шаблон ЕДИНЫЙ ЯЗЫК. Как минимум он извлек этот шаблон из пыльных архивов премудростей проектирования. Это командный шаблон, который используется для фиксации понятий и терминов определенной предметной области в самой модели программного обеспечения. Модель программного обеспечения включает существительные, прилагательные, глаголы и более содержательные выражения, на которых формально говорит группа разработчиков, которая включает одного или нескольких экспертов в проблемной области. Однако было бы ошибкой считать, что единый язык ограничен только словами. Точно так же, как любой естественный язык отражает мышление тех, кто говорит на нем, единый язык отражает ментальную модель экспертов в предметной области, в которой вы работаете. Таким образом,

программное обеспечение и тесты, проверяющие соответствие модели принципам предметной области, фиксируют и твердо придерживаются правил языка, на котором думает и говорит команда проектировщиков. ЕДИНЫЙ ЯЗЫК так же ценен, как и другие стратегические и тактические шаблоны моделирования, а в некоторых случаях имеет дополнительные преимущества.

Проще говоря, применение облегченного подхода DDD приводит к созданию низкоуровневых моделей предметной области. Именно по этой причине шаблоны ЕДИНЫЙ ЯЗЫК, ОГРАНИЧЕННЫЙ КОНТЕКСТ и КАРТА КОНТЕКСТОВ являются такими многообещающими. Вы получаете больше, чем малопонятный жаргон команды. Язык команды в явном ограниченном контексте, выраженном как модель предметной области, добавляет истинную бизнес-ценность и дает нам уверенность, что мы реализуем корректное программное обеспечение. Даже с технической точки зрения это помогает нам создавать ясные модели с более мощными характеристиками, которые меньше уязвимы для ошибок. Таким образом, я связываю стратегические проектные шаблоны с понятными примерами реализации.

Эта книга отображает ландшафт DDD способом, который позволяет осознать преимущества как стратегического, так и тактического проектирования. Она связывает в одно целое бизнес-ценности и технические средства, глубоко погружая читателей в детали.

Было бы разочарованием, если бы все, что мы когда-либо делали с помощью подхода DDD, "осталось на земле". Застревая в деталях, мы забыли бы, что вид с высоты птичьего полета тоже многому учит. Не ограничивайтесь походами по пересеченной местности. Наберитесь смелости сесть в кресло пилота и посмотрите на проблему с высоты, о которой мы говорили. Учебные полеты с помощью стратегических шаблонов ОГРАНИЧЕННЫЙ КОНТЕКСТ и КАРТА КОНТЕКСТОВ дадут вам более широкое представление об их полноценной реализации. Когда вы совершите полет на самолете DDD, я буду считать, что достиг своей цели.

Обзор глав

Перейдем к обзору содержания глав и покажем, какую пользу из них можно извлечь.

Глава 1. Знакомство с DDD

В этой главе описаны преимущества подхода DDD и показано, как использовать большинство из них. Вы узнаете, какую пользу подход DDD может принести вашему проекту и вашей команде, когда вы сталкиваетесь со сложной системой. Вы научитесь оценивать проекты, чтобы выяснить, стоит ли применять к ним

подход DDD. Мы рассмотрим общепринятые альтернативы DDD и покажем, почему они часто приводят к проблемам. В главе закладываются основы DDD, поскольку в ней объясняется, как выполнить первые этапы проекта, и даже даются советы, как убедить руководство, экспертов в предметной области и членов команды разработчиков применить подход DDD. Все это позволит вам справиться с трудностями использования DDD.

Мы рассмотрим сценарий, описывающий исследование проекта, в котором участвуют гипотетическая компания и ее команда проектировщиков, сталкивающаяся с реальными проблемами DDD. Компания, заключившая контракт на создание инновационной продукции на основе платформы SaaS в многоарендной архитектуре, совершает многочисленные ошибки, характерные для первичного знакомства с подходом DDD, но делает важные открытия, помогающие решить проблемы и поддержать проект "на плаву". С проектом связано большинство разработчиков, поскольку для управления проектом используется приложение на основе методологии Scrum. Этот сценарий закладывает основы для следующих глав. Каждый стратегический и тактический шаблон описывается с точки зрения членов команды. В главе показано, как они делают свои ошибки и достигают успеха в освоении DDD.

Глава 2. ПРЕДМЕТНАЯ ОБЛАСТЬ, ПРЕДМЕТНАЯ ПОДОБЛАСТЬ И ОГРАНИЧЕННЫЕ КОНТЕКСТЫ

Что такое ПРЕДМЕТНАЯ ОБЛАСТЬ (DOMAIN), ПРЕДМЕТНАЯ ПОДОБЛАСТЬ (SUBDOMAIN) и СМЫСЛОВОЕ ЯДРО (CORE DOMAIN)? Что такое ОГРАНИЧЕННЫЙ КОНТЕКСТ? Как и почему его следует применять? Мы отвечаем на эти вопросы, анализируя ошибки, которые делает команда проектировщиков в нашем сценарии. На первом этапе своего проекта DDD они неправильно понимают ПОДОБЛАСТЬ, в которой они работают, ее ОГРАНИЧЕННЫЙ КОНТЕКСТ и ЕДИНЫЙ ЯЗЫК. Они практически ничего не знают о стратегическом проектировании и используют лишь тактические шаблоны для решения технических проблем. Это приводит к проблемам, связанным с первоначальной моделью предметной области. К счастью, они вовремя понимают, что произошло.

Главная мысль, которая проводится к этой главе, заключается в том, что применение ОГРАНИЧЕННОГО КОНТЕКСТА позволяет правильно разграничить и разделить модели. В ней не только анализируются распространенные ошибки при работе с шаблонами, но и даются советы по их эффективной реализации. В главе показано, как члены команды исправляют ошибки и в результате создают два разных ОГРАНИЧЕННЫХ КОНТЕКСТА. Это приводит к концепциям моделирования, правильно отделенным в третьем ОГРАНИЧЕННОМ КОНТЕКСТЕ, новому СМЫСЛОВОМУ ЯДРУ и основному примеру, используемому в книге.

Эта глава должна вызвать сильный интерес у читателей, испытывающих большие трудности, связанные с исключительно формальным применением подхода DDD. Если вы не знакомы со стратегическим проектированием, то эта глава покажет вам правильное направление.

Глава 3. КАРТЫ КОНТЕКСТОВ

Шаблон КАРТА КОНТЕКСТОВ — мощный инструмент, помогающий команде понять свою предметную область, очертить границы между разными моделями и выделить их реальную или потенциальную общность. Этот метод не сводится к рисованию диаграмм системной архитектуры. Его цель — понимание отношений между разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ на предприятии и между шаблонами, используемыми для четкого отображения объектов одной модели в объекты другой модели. Этот инструмент играет важную роль для успешной работы с ограниченными контекстами на сложном предприятии. В главе описана работа команды проектировщиков, применяющих ОТОБРАЖЕНИЕ КОНТЕКСТА для понимания проблем, созданных их первым ОГРАНИЧЕННЫМ КОНТЕКСТОМ (глава 2). Затем в ней показано, как два итоговых ограниченных контекста влияют на проектирование и реализацию нового СМЫСЛОВОГО ЯДРА.

Глава 4. АРХИТЕКТУРА

Модель МНОГОУРОВНЕВАЯ АРХИТЕКТУРА (LAYERED ARCHITECTURE) знают все. Но являются ли уровни единственным способом описания приложения DDD или можно использовать другие архитектуры? В этой главе мы покажем, как использовать подход DDD в таких архитектурах, как ГЕКСАГОНАЛЬНАЯ (ПОРТЫ И АДАПТЕРЫ), СЕРВИСНО-ОРИЕНТИРОВАННАЯ, REST, CQRS, СОБЫТИЙНАЯ (КАНАЛЫ и ФИЛЬТРЫ, ДОЛГОВРЕМЕННЫЕ ПРОЦЕССЫ, или САГИ, ПОРОЖДЕНИЕ СОБЫТИЙ), а также DATA FABRIC/GRID-BASED. Некоторые из этих архитектур будут положены в основу проекта.

Глава 5. СУЩНОСТИ

Первым из тактических шаблонов DDD рассматривается шаблон СУЩНОСТЬ (ENTITY). Команда проекта слишком глубоко изучила этот шаблон, просмотрев возможность применить важный шаблон ОБЪЕКТ-ЗНАЧЕНИЕ (VALUE OBJECT). В результате возникла дискуссия о том, как избежать широко распространенного избыточного использования шаблона СУЩНОСТЬ, обусловленного чрезмерным влиянием баз данных и принципов постоянного хранения данных.

Научившись правильно применять этот шаблон, вы увидите множество примеров правильного проектирования сущностей. Как выразить ЕДИНЫЙ ЯЗЫК посредством СУЩНОСТЕЙ? Как тестировать, реализовывать и хранить СУЩНОСТИ? Мы ответим на каждый из этих вопросов.

Глава 6. ОБЪЕКТЫ-ЗНАЧЕНИЯ

На ранних этапах проектирования команда пропустила важные возможности моделирования с помощью шаблона ОБЪЕКТ-ЗНАЧЕНИЕ. Она слишком сосредоточилась на индивидуальных атрибутах СУЩНОСТЕЙ, не обратив внимания на то, что многочисленные связанные друг с другом атрибуты образуют некое неизменяемое целое. Проектирование на основе шаблона ОБЪЕКТ-ЗНАЧЕНИЕ рассматривается с разных точек зрения. В главе показано, как идентифицировать специальные характеристики модели, чтобы определить, какой шаблон использовать — ОБЪЕКТ-ЗНАЧЕНИЕ или СУЩНОСТЬ. Помимо этого, в главе рассматривается такая важная тема, как роль ОБЪЕКТОВ-ЗНАЧЕНИЙ в интеграции и моделировании СТАНДАРТНЫХ ТИПОВ (STANDARD TYPES). После этого показано, как проектировать предметно-ориентированные тесты, как реализовывать типы ЗНАЧЕНИЙ (VALUES) и как избежать вредного влияния механизмов постоянного хранения данных на хранение ЗНАЧЕНИЙ в качестве компонентов АГРЕГАТОВ.

Глава 7. СЛУЖБЫ

В главе показано, как распознать ситуации, в которых концепцию предметной области следует моделировать как мелкомодульную СЛУЖБУ (SERVICE), не имеющую состояния. Вы узнаете, когда следует проектировать СЛУЖБУ, а не СУЩНОСТЬ или ОБЪЕКТ-ЗНАЧЕНИЕ, и как реализовать СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICES) для воплощения бизнес-логики и достижения целей технической интеграции. Описываются ситуации, в которых применяются службы, и демонстрируются методы их разработки.

Глава 8. СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

В книге Эрика Эванса нет формального описания шаблона СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS). Вы узнаете, почему СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, публикуемые моделью, являются такими мощными, а также как их можно использовать для достижения разных целей, в том числе для поддержки интеграции и автономных бизнес-служб. Несмотря на то что приложения посылают и обрабатывают разнообразные технические события, события предметной области обладают особенными характеристиками. В главе даются

рекомендации по проектированию и реализации, а также описываются существующие возможности и компромиссы. Затем показано, как создать механизм ИЗДАТЕЛЬ-ПОДПИСЧИК (PUBLISH-SUBSCRIBE), как публикуются события предметной области для подписчиков внутри предприятия, как создать ХРАНИЛИЩЕ СОБЫТИЙ (EVENT STORE) и управлять им, и как правильно решать типичные проблемы. Каждая из этих тем обсуждается в свете усилий команды проекта, направленных на стремление наилучшим образом использовать имеющиеся возможности.

Глава 9. МОДУЛИ

Как организовать объекты, существующие в рамках модели, в контейнеры правильного размера, имеющие ограниченное сцепление с объектами, находящимися в других контейнерах? Как назвать эти контейнеры, чтобы они отражали ЕДИНЫЙ ЯЗЫК? Как использовать более современные средства модульного проектирования, такие как OSGi и Jigsaw, поддерживаемые языками и каркасами? Здесь вы узнаете, как использовать МОДУЛИ (MODULES) в разных проектах.

Глава 10. АГРЕГАТЫ

АГРЕГАТЫ, вероятно, являются самым плохо понимаемым среди всех тактических шаблонов DDD. Впрочем, если принять определенные эмпирические правила, можно упростить шаблон АГРЕГАТЫ и облегчить его реализацию. Мы покажем, как преодолеть сложности использования шаблона АГРЕГАТ, создающего границы согласованности вокруг кластеров малых объектов. Обращая слишком большое внимание на относительно неважные аспекты АГРЕГАТОВ, команда проектировщиков в нашем сценарии столкнулась с разными препятствиями. Вместе с командой мы пройдем по нескольким путям, которые окажутся ошибочными, и покажем, как исправить ситуацию. В результате читатели должны более глубоко понять СМЫСЛОВОЕ ЯДРО. Мы увидим, как команда исправит свои ошибки, правильно применив концепцию транзакционной и конечной согласованности, и как она создаст более гибкую и эффективную модель в среде распределенной обработки.

Глава 11. ФАБРИКИ

В книге [Gamma et al.] много говорится о ФАБРИКАХ (FACTORIES), так зачем же писать о них еще раз? Это простая глава, в которой никто не пытается заново изобрести велосипед. Мы попытаемся понять, в каких ситуациях должны существовать ФАБРИКИ. Разумеется, есть хорошие рекомендации, указывающие, когда

следует проектировать ФАБРИКИ в рамках подхода DDD. Мы покажем, как наша команда создает ФАБРИКИ в рамках СМЫСЛОВОГО ЯДРА, чтобы упростить клиентский интерфейс и защитить пользователя модели от катастрофических ошибок в многоарендной среде.

Глава 12. ХРАНИЛИЩА

Является ли ХРАНИЛИЩЕ простым ОБЪЕКТОМ ДОСТУПА К ДАННЫМ (DATA ACCESS OBJECT — DAO)? А если нет, то в чем разница? Почему проектирование ХРАНИЛИЩ следует осуществлять как моделирование коллекций, а не баз данных? Мы покажем, как ХРАНИЛИЩЕ используется в сочетании с технологией ORM (Object Relational Mapping), поддерживающей связный решеточный распределенный кеш (coherence grid-based distributed cache) и использующей хранилище NoSQL "ключ–значение". Каждый из этих механизмов постоянного хранения данных стал доступен команде проекта благодаря мощи и универсальности шаблона ХРАНИЛИЩЕ.

Глава 13. Интеграция ОГРАНИЧЕННЫХ КОНТЕКСТОВ

После освоения высокоуровневых методов отображения контекстов и тактических шаблонов возникает вопрос "Как осуществить интеграцию моделей?" Какие возможности для интеграции предоставляет подход DDD? Глава раскрывает несколько способов интеграции с помощью шаблона КАРТА КОНТЕКСТОВ (CONTEXT MAP). Она содержит рекомендации по интеграции СМЫСЛОВОГО ЯДРА с другими вспомогательными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, введенными в предыдущих главах.

Глава 14. Приложение

Мы разработали модель с помощью ЕДИНОГО ЯЗЫКА СМЫСЛОВОГО ЯДРА, а также примерные тесты для проверки ее работоспособности и правильности. А что делать другим членам команды проекта, разрабатывающим приложения, сопровождающие нашу модель? Должны ли они использовать ОБЪЕКТЫ ПЕРЕДАЧИ ДАННЫХ (DATA TRANSFER OBJECTS — DTO) для обмена информацией между моделью и пользовательским интерфейсом? Или они должны использовать другие возможности для передачи состояния модели компонентам представления? Как работают библиотека ПРИКЛАДНЫХ СЛУЖБ и инфраструктура? Ответы на эти вопросы, проиллюстрированные нашим учебным проектом, можно найти в этой главе.

Приложение. АГРЕГАТЫ и ИСТОЧНИКИ СОБЫТИЙ: A+ES

ИСТОЧНИКИ СОБЫТИЙ (EVENT SOURCING) — это важная техническая концепция хранения агрегатов, лежащая в основе разработки событийно-ориентированной архитектуры (Event-Driven Architecture). ИСТОЧНИКИ СОБЫТИЙ можно использовать для представления состояния агрегата в целом, которое является следствием последовательности событий, произошедших с момента его создания. События используются для восстановления состояния агрегата с помощью их воспроизведения в том же порядке, в котором они происходили. Предпосылкой для этого является тот факт, что такой подход упрощает обеспечение постоянного хранения данных и позволяет воплотить концепции со сложными поведенческими характеристиками. Кроме того, события сами по себе оказывают большое влияние пользовательские и внешние системы.

Язык Java и инструменты разработки

В большинстве примеров в книге использован язык Java. Я мог бы привести примеры на языке C#, но пришел к выводу, что следует использовать Java.

Во-первых, как это ни досадно, я считаю, что члены сообщества Java пренебрегают хорошими методами проектирования и разработки. В настоящее время сложно найти ясную и четкую модель предметной области в большинстве проектов на языке Java. Мне кажется, что методология Scrum и другие технологии гибкого проектирования вытеснили тщательное моделирование, а разработчикам внушают, что список заданий (product backlog) — это всего лишь набор проектов. Большинство проектировщиков, использующих методы гибкой разработки, мало задумываются о том, как их задания повлияют на базовую бизнес-модель. По-моему очевидно, что методология Scrum, например, никогда не предназначалась для замены проектирования. Независимо от того, сколько менеджеров по проектам и продукции хотели бы видеть вас бодро марширующими на жестоком пути непрерывного развертывания (continuous delivery), технология Scrum — это не просто средство доставить удовольствие энтузиастам диаграмм Ганта (Gantt chart), хотя во многих случаях она используется только для этого.

Я считаю, что это большая проблема, и хотел бы вернуть сообщество Java к моделированию предметной области, заставив его подумать о том, насколько большую пользу могут им принести солидные, а не гибкие и быстрые методы проектирования.

Использованию подхода DDD на платформе .NET посвящено достаточное количество хороших книг, например книга Jimmy Nilsson *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* [Nilsson]. Благодаря книге Джимми и деятельности других специалистов, продвигающих принципы Alt.NET, в

сообществе .NET появилось много хороших методов проектирования и разработки. Разработчики на языке Java должны принять это к сведению.

Во-вторых, мне хорошо известно, что сообщество C#.NET не имеет проблем с пониманием кода на языке Java. Поскольку большинство членов сообщества DDD использует язык C#.NET и большинство рецензентов предыдущего издания были программистами на C#, я не получал никаких жалоб, касавшихся понимания кода на языке Java. Итак, я убежден, что использование языка Java в книге никоим образом не оттолкнет программистов на языке C#.

Следует добавить, что пока я писал книгу, произошел значительный сдвиг от реляционных баз данных в сторону документных хранилищ и хранилищ "ключ–значение". Это настолько серьезное явление, что Мартин Фаулер (Martin Fowler) быстро подобрал для них общее название "агрегатно-ориентированное хранилище". Это подходящее название, которое хорошо описывает преимущества использования хранилищ NoSQL в проектах DDD.

В ходе моей консультационной деятельности я убедился, что многие по-прежнему остаются приверженцами реляционных баз данных и объектно-реляционного отображения. По этой причине я полагаю, что с практической точки зрения энтузиастам NoSQL было бы полезно последовать моим советам, касающимся методов объектно-реляционного отображения для моделей предметной области. Однако я знаю, что это может вызвать ко мне презрение со стороны тех, кто думает, что объектно-реляционное рассогласование (impedance mismatch) не заслуживает внимания. Отлично, я принимаю огонь на себя, потому что существует огромное множество людей, которые должны как-то ежедневно справляться с этим рассогласованием, вызывая возможные насмешки со стороны меньшинства.

Кроме того, в главе 12, "Хранилища", я даю рекомендации по использованию документных хранилищ, хранилищ "ключ–значение" и хранилищ тип Data Fabric/Grid-Based. В нескольких местах я обсуждаю, как использование хранилища NoSQL могло бы повлиять на альтернативный проект агрегатов и их компонентов. Вероятно, тенденция к интенсивному использованию хранилищ NoSQL в этом секторе продолжится, и разработчики объектно-реляционных проектов должны учесть это обстоятельство. Как видите, я понимаю аргументы обеих сторон и с обеими согласен. Все это часть продолжающихся споров, вызванных технологическими тенденциями, и эти споры необходимы, чтобы позитивные изменения продолжались.

Благодарности

Я благодарен прекрасным сотрудникам издательства Addison-Wesley за предоставленную мне возможность опубликовать книгу под торговой маркой, имеющей высокую репутацию. Как я много раз говорил в своих лекциях и выступлениях, издательство Addison-Wesley хорошо понимает значение подхода DDD. Кристофер Гузиковский (Christopher Guzikowski) и Крис Зан (Chris Zahn) (Доктор Зет) постоянно поддерживали мои усилия на протяжении процесса редактирования книги. Я не забыл тот день, когда Кристофер Гузиковский позвонил мне, чтобы сообщить, что выбрал меня одним из своих авторов. Я помню, как он уговаривал меня отбросить сомнения, когда книга еще только задумывалась. Разумеется, именно Доктор Зет убедил меня, что книга готова к публикации. Благодарю моего технического редактора Элизабет Райан (Elizabeth Ryan) за координацию процесса публикации. Отдельное спасибо моему литературному редактору Барбаре Вуд (Barbara Wood).

Эрик Эванс посвятил своей первой выдающейся работе по DDD пять лет. Без принципов, выросших из языка Smalltalk и шаблонов, а также уточненных самим Эриком Эвансом, многие разработчики вынуждены были бы поставлять плохое программное обеспечение. К сожалению, эта проблема является более распространенной, чем хотелось бы. Как рассказывает Эрик, плохое качество разработки программного обеспечения и безынициативность апатичных команд разработчиков программного обеспечения почти подвели его к решению уйти из программной инженерии навсегда. Мы должны горячо поблагодарить Эрика за то, что он сосредоточил свою энергию на образовании, а не на новой карьере.

После первой конференции по DDD, состоявшейся в 2011 году, на которую Эрик меня пригласил, стало ясно, что лидеры должны выработать набор рекомендаций для разработчиков, желающих успешно использовать подход DDD. Большая часть книги уже была написана, и было понятно, в чем нуждаются разработчики. Я предложил написать эссе об эмпирических правилах работы с агрегатами и решил, что эта серия из трех частей под названием *Effective Aggregate Design* (*Реальное агрегатное проектирование*) должна лечь в основу главы 10 этой книги. После распространения этого произведения через веб-сайт `dddcommunity.org` стало ясно, насколько востребованы такие рекомендации. Я благодарю лидеров сообщества DDD, давших рецензию на это эссе и тем самым обеспечивших ценную обратную связь. Эрик Эванс и Пол Райнер (Paul Rayner) написали несколько подробных отзывов на это эссе. Я также получил отзывы от Уди Даана (Udi Dahan), Грэга Янга (Greg Young), Джимми Нильссона (Jimmy Nilsson), Никласа Хедмана (Niclas Hedhman) и Рикарда Оберга (Rickard Öberg).

Особую благодарность я выражаю Рэнди Стаффорду, который уже долгое время является членом сообщества DDD. Посетив конференцию по DDD, я приехал

несколько лет назад в Денвер, и Рэнди убедил меня принять более активное участие в работе сообщества DDD. Немного позже Рэнди познакомил меня с Эриком Эвансом, и я смог изложить ему свои идеи о сплочении сообщества DDD. Поскольку мои идеи были довольно крупномасштабными и труднодостижимыми, Эрик убедил нас сформировать небольшую группу под руководством лидеров сообщества DDD, которая могла бы поставить более реальные цели в ближайшей перспективе. Из этих дискуссий выросла идея провести в 2011 году конференцию по DDD. Не стоит и говорить, что без уговоров Рэнди сформулировать свои мысли о подходе DDD эта книга никогда не появилась бы на свет, и, возможно, даже не состоялась бы сама конференция. Несмотря на то что Рэнди был слишком занят работой над продуктом Oracle Coherence, чтобы внести свой вклад в эту книгу, возможно, в будущем мы еще сможем объединить наши усилия.

Огромная благодарность Ринату Абдуллину (Rinat Abdullin), Стефану Тилькову (Stefan Tilkov) и Уэсу Уильямсу (Wes Williams) за их вклад в разделы, посвященные специальным темам. Один человек практически не может знать все, что связано с подходом DDD, и уж совершенно невозможно быть экспертом во всех областях разработки программного обеспечения. По этой причине я обратился к экспертам в конкретных областях, которые описываются в нескольких разделах главы 4 и в приложении А. Я благодарю Стефана Тилькова за уникальные знания об архитектурном стиле, которыми он владеет, Уэса Уильямса — за его опыт работы с технологией GemFire и Рината Абдуллина — за то, что он все время делился со мной своим постоянно увеличивающимся опытом по реализации шаблона АГРЕГАТ на основе шаблона ИСТОЧНИК СОБЫТИЙ.

Один из моих первых рецензентов, Лев Городинский (Lev Gorodinski), также присоединился к проекту. Впервые я встретил Леву на конференции по DDD в Денвере. Он дал много советов на основе своего опыта реализации подхода DDD со своей командой в г. Боулдер-Сити, штат Колорадо. Надеюсь, что моя книга помогла Льву так же, как его критические замечания помогли мне. Мне кажется, что будущее подхода DDD зависит и от Льва.

Многие люди сделали замечания хотя бы к одной главе, а некоторые даже к нескольким главам. Наиболее ценные советы дали Гойко Адзич (Gojko Adzic), Альберто Брандолини (Alberto Brandolini), Уди Даан (Udi Dahan), Дэн Хейвуд (Dan Haywood), Дэйв Мюрхед (Dave Muirhead) и Стефан Тильков (Stefan Tilkov). В частности, Дэн Хейвуд и Гойко Адзич прислали свои советы одними из первых, когда текст книги был еще далек от совершенства. Я рад, что они вытерпели неудобства и поправили меня. Взгляды Альберто Брандолини на стратегическое проектирование в целом и шаблон КАРТА КОНТЕКСТОВ в частности помогли мне сосредоточиться на самом важном. Дэйв Мюрхед, обладающий огромным опытом в объектно-ориентированном проектировании, моделировании предметной области, а также в обеспечении постоянного хранения объектов и технологии In-Memory

Data Grid — включая технологии GemFire и Coherence, — дал несколько советов по истории и уточнил детали, относящиеся к постоянного хранения объектов. Помимо вклада, относящегося к архитектуре REST, Стефан Тильков изложил дополнительные сведения об архитектуре в целом, а также об архитектурных стилях SOA и КАНАЛЫ И ФИЛЬТРЫ в частности. В заключение Уди Даан проверил правильность описания и помог мне прояснить некоторые концепции архитектуры CQRS, ДОЛГОВРЕМЕННЫЕ ПРОЦЕССЫ (или САГИ), а также обмен сообщениями в каркасе NServiceBus. Среди других рецензентов я хотел бы упомянуть Рината Абдуллина, Свейна Арне Акенхаузена (Svein Arne Ackenhausen), Хавьера Руиса Арангурена (Javier Ruiz Aranguren), Уильяма Домана (William Doman), Чака Дарфи (Chuck Durfee), Крейга Хоффа (Craig Hoff), Эдена Джеймсона (Aeden Jameson), Дживея By (Jiwei Wu), Джошуа Малеца (Josh Maletz), Тома Марса (Tom Marrs), Майкла Маккарти (Michael McCarthy), Роба Мейдала (Rob Meidal), Джона Сленка (Jon Slenk), Аарона Стоктона (Aaron Stockton), Тома Стоктона (Tom Stockton), Криса Саттона (Chris Sutton,) и Уэса Уильямса.

Компания Scorpio Steele создала фантастические иллюстрации к книге. Она изобразила членов команды IDDD в виде супергероев, которыми они и являются на самом деле. На другом конце спектра находится литературная рецензия моего хорошего друга Керри Гилберта (Kerry Gilbert). Читатели могут убедиться, что с технической точки зрения я не сделал ни одной ошибки, но Керри подверг мой текст грамматическому разбору.

Мои отец и мать вдохновляли и поддерживали меня на протяжении всей моей жизни. Мой отец — персонаж AJ в юмористическом комиксе "Ковбойская логика", который встречается по всей книге — не просто ковбой. Не поймите меня неправильно. Быть отличным ковбоем — это уже здорово! Мой отец не только любит самолеты, но и является дипломированным инженером-строителем, геодезистом и талантливым негоциантом. Он по-прежнему любит заниматься математикой и исследовать галактики. Помимо всего остального, чему он меня научил, отец показал мне, как решать задачи о равнобедренном треугольнике, когда мне было примерно десять лет. Спасибо, папа, за то, что ты выявил мои технические способности в таком раннем возрасте. Спасибо моей маме, одной из самых хороших женщин, которых я знаю. Она всегда вдохновляла и поддерживала меня в моих начинаниях. Кроме того, я перенял у нее стойкость к испытаниям. Я мог бы продолжать до бесконечности, но и в этом случае не смог бы выразить все восхищение своей мамой.

Несмотря на то что книга содержит посвящение моей любимой жене Николь и нашему чудесному сыну Тристану, я не мог не упомянуть их особо. Благодаря им я смог начать и закончить эту книгу. Без их поддержки и воодушевления это не было бы возможным. Огромное спасибо, мои дорогие родные и близкие!

Об авторе

Вон Вернон — ветеран среди специалистов по проектированию и архитектуре программного обеспечения, обладающий более чем двадцатипятилетним опытом. Он является идейным лидером движения за упрощение проектирования и реализации программного обеспечения с помощью инновационных методов. В начале 1980-х годов он программировал на объектно-ориентированных языках, а в начале 1990-х стал применять принципы предметно-ориентированного проектирования, занимаясь моделированием на языке Smalltalk. Он обладает большим опытом в широкой сфере деловой активности, включая аэрокосмическую промышленность, защиту окружающей среды, геодезию, страхование, здравоохранение и телекоммуникации. Он достиг успеха в своих технических начинаниях, создавая повторно используемые каркасы, библиотеки и инструменты для ускорения разработки программного обеспечения. Он консультирует и выступает по всему миру, обучая специалистов методам реализации предметно-ориентированного проектирования. Его последние достижения описаны на веб-сайте `www.VaughnVernon.co` и в Твиттере: `@VaughnVernon`.

Руководство по использованию книги

Книга Эрика Эванса *Предметно-ориентированное проектирование* показала, что такое язык крупных шаблонов. Язык шаблонов — это множество взаимозависимых шаблонов программного обеспечения. Любой отдельно взятый шаблон зависит от одного или нескольких других шаблонов, которые, в свою очередь, зависят от него. Что из этого следует?

Это значит, что когда вы читаете любую главу этой книги, то сталкиваетесь с неким шаблоном DDD, который в этой главе не обсуждается и который вам еще не известен. Не пугайтесь и не прекращайте чтения. Весьма вероятно, что этот шаблон подробно объясняется в другой главе.

Для того чтобы распутать сложности языка шаблонов, я использовал обозначения, описанные в табл. 1.

Таблица 1. Обозначения, используемые в книге

Обозначение	Описание
НАЗВАНИЕ ШАБЛОНА (#)	1. Первое упоминание шаблона в главе, которую вы читаете, или 2. Важная повторная ссылка на шаблон, который уже упоминался в этой главе, позволяющая уточнить, где можно получить дополнительную информацию о нем
ОГРАНИЧЕННЫЙ КОНТЕКСТ (2)	Ссылка на главу 2, в которой вы найдете более глубокое описание ОГРАНИЧЕННОГО КОНТЕКСТА
ОГРАНИЧЕННЫЙ КОНТЕКСТ	Таким образом я ссылаюсь на шаблон, который уже упоминался в текущей главе. Я не хотел утомлять читателей, выделяя каждую ссылку на шаблон полужирным шрифтом и сопровождая ее номером главы
[ССЫЛКА]	Это библиографическая ссылка на другую работу
[Эванс] или [Evans, Ref]	Я не даю обширных описаний шаблонов DDD, поэтому, если читатели захотят узнать о них больше, им следует прочитать работы Эрика Эванса. (Настоятельно рекомендую!) Ссылка [Evans] означает его классическую книгу *Domain-Driven Design*. Ссылка [Evans, Ref] означает вторую публикацию, представляющую собой отдельное, сжатое описание шаблонов, приведенное в работе [Evans], исправленное и расширенное

Окончание табл. 1

Обозначение	Описание
[Gamma et al.] [Fowler, P of EAA]	Ссылка [Gamma et al.][3] относится к книге *Design Patterns.* Ссылка [Fowler, P of EAA] относится к книге Мартина Фаулера (Martin Fowler) *Patterns of Enterprise Application Architecture.*[4] Я часто ссылаюсь на эту книгу. Хотя я ссылаюсь и на другие публикации, эти ссылки будут встречаться намного чаще. Подробности можно найти в библиографии, приведенной в конце книги

Если вы начинаете чтение с середины главы и видите ссылку ОГРАНИЧЕННЫЙ КОНТЕКСТ, то помните, что в книге, скорее всего, есть глава, посвященная этому шаблону. Просто просмотрите предметный указатель и найдите нужную страницу.

Если вы уже читали книгу [Эванс] и в некоторой степени знакомы с шаблонами, то, скорее всего, вы будете использовать мою книгу для прояснения своего понимания принципов DDD и выяснения идей о том, как улучшить ваш стиль проектирования. В этом случае вам не нужна общая картина. Однако если вы впервые сталкиваетесь с подходом DDD, то следующая глава поможет вам понять, как шаблоны связаны друг с другом и чем эта книга может быть вам полезной. Итак, вперед!

Общая картина DDD

Сначала я опишу один из столпов подхода DDD — **ЕДИНЫЙ ЯЗЫК (1)**. Он применяется внутри отдельного **ОГРАНИЧЕННОГО КОНТЕКСТА (2)**. Очевидно, что вам необходимо ознакомиться с главными принципами предметно-ориентированного моделирования. Просто помните, что как бы ваши программные модели ни были хорошо разработаны тактически, стратегически они должны отражать точный ЕДИНЫЙ ЯЗЫК в четко ОГРАНИЧЕННОМ КОНТЕКСТЕ.

[3] Русский перевод: Гамма Э. и др. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* — СПб. : Питер, 2013. — 368 с. В дальнейшем приводятся ссылки на русскоязычное издание, например [Гамма и др.]. — *Примеч. ред.*

[4] Русский перевод: Фаулер М. и др. *Шаблоны корпоративных приложений.* — М. : ООО "И.Д. Вильямс", 2012. — 448 с. В дальнейшем приводятся ссылки на русскоязычное издание, например [Фаулер и др.]. — *Примеч. ред.*

Стратегическое моделирование

ОГРАНИЧЕННЫЙ КОНТЕКСТ — это концептуальная граница применимости модели предметной области. Он создает контекст единого языка, на котором говорят команда и с помощью которого точно выражается модель разрабатываемого программного обеспечения (рис. Р.1).

Рис. Р.1. Диаграмма, иллюстрирующая ОГРАНИЧЕННЫЙ КОНТЕКСТ и связанный с ним ЕДИНЫЙ ЯЗЫК

По мере освоения стратегического проектирования вы поймете, что **КАРТЫ КОНТЕКСТОВ (3)**, показанные на рис. Р.2, работают гармонично. Ваша команда будет использовать КАРТУ КОНТЕКСТОВ для того, чтобы понять свою предметную область проекта.

Покамыпросторассматриваемобщуюкартинустратегическогопроектирования на основе принципов DDD. Очень важно в ней хорошо разобраться.

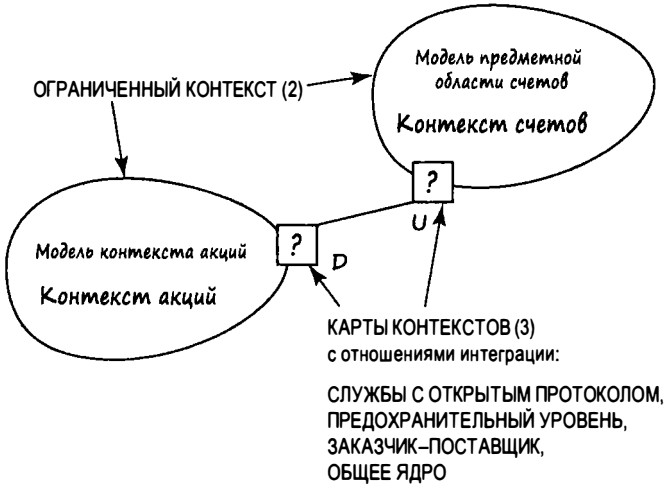


Рис. Р.2. КАРТЫ КОНТЕКСТОВ демонстрируют отношения между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ

Архитектура

Иногда новый ОГРАНИЧЕННЫЙ КОНТЕКСТ или один из существующих, взаимодействующие через КАРТУ КОНТЕКСТОВ, обязаны подчиниться новой **АРХИТЕКТУРЕ (4)**. Следует помнить, что ваши стратегические и тактические модели предметной области должны быть архитектурно нейтральными. Тем не менее и внутри каждой модели, и в отношениях между этими моделями прослеживаются определенные архитектурные характеристики. Для реализации ОГРАНИЧЕННОГО КОНТЕКСТА хорошо подходит ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА (HEXAGONAL ARCHITECTURE), которая хорошо сочетается с другими архитектурами, такими как СЕРВИСНО-ОРИЕНТИРОВАННАЯ (SERVICE-ORIENTED), REST, СОБЫТИЙНО-УПРАВЛЯЕМАЯ (EVENT-DRIVEN) и др. ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА показана на рис. Р.3. На первый взгляд, она может показаться довольно громоздкой, но на самом деле этот архитектурный стиль чрезвычайно удобен для развертывания.

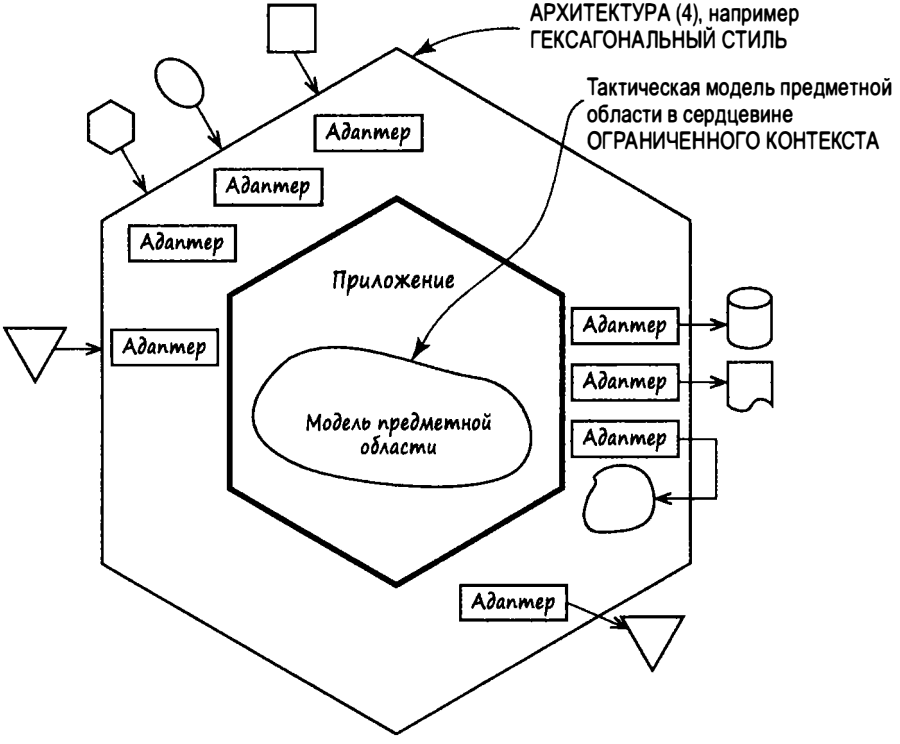


Рис. Р.3. ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА с моделью предметной области в основе программного обеспечения

Иногда возникает соблазн сделать слишком сильный упор на архитектуре, а не на важности тщательной разработки модели, основанной на принципах DDD. Архитектура важна, но архитектурные веяния приходят и уходят. Правильно расставляйте приоритеты, главное внимание уделяя модели предметной области, которая имеет более высокую бизнес-ценность и является более долговечной.

Тактическое моделирование

Тактическое моделирование осуществляется внутри ОГРАНИЧЕННОГО КОНТЕКСТА с помощью предметно-ориентированных шаблонов структурных элементов. Одним из наиболее важных шаблонов тактического проектирования является **АГРЕГАТ (10)**, показанный на рис. P.4.

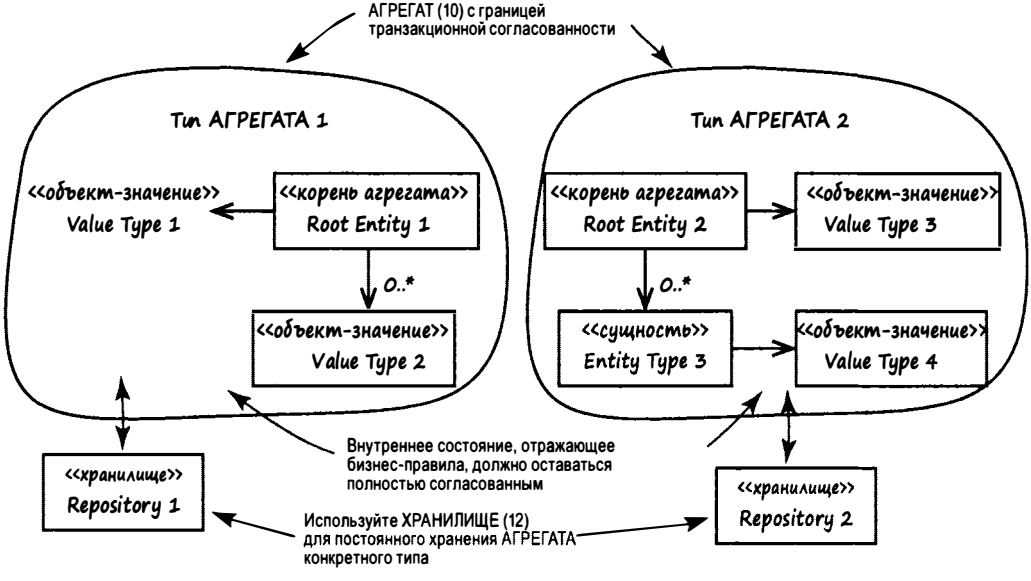


Рис. P.4. Два типа АГРЕГАТОВ с границами транзакционной согласованности

Агрегат состоит либо из отдельной **СУЩНОСТИ (5)**, либо из кластера СУЩНОСТЕЙ и **ОБЪЕКТОВ-ЗНАЧЕНИЙ (6)**, которые должны быть транзакционно согласованными на протяжении существования АГРЕГАТА. Несмотря на то что умение эффективно моделировать АГРЕГАТЫ является довольно важным, этот метод является одним из наиболее плохо понимаемых среди всех компонентов DDD. Может возникнуть вопрос "Раз этот метод так важен, почему АГРЕГАТЫ описываются не в начале книги?" Прежде всего, тактические шаблоны описываются точно в таком же порядке, как и в книге [Эванс]. Кроме того, поскольку АГРЕГАТЫ основаны на других тактических шаблонах, мы опишем основные

структурные элементы — такие, как СУЩНОСТИ и ОБЪЕКТЫ-ЗНАЧЕНИЯ — до более сложного шаблона АГРЕГАТ.

Экземпляр агрегата сохраняется с помощью **ХРАНИЛИЩА (12)**. Позднее он там обнаруживается и извлекается. Этот процесс представлен на рис. Р.4.

Для выполнения бизнес-операций, которые невозможно естественным образом выразить посредством операций над СУЩНОСТЬЮ или ОБЪЕКТОМ-ЗНАЧЕНИЕМ, используйте **СЛУЖБЫ (7)**, не имеющие состояний, как показано на рис. Р.5.

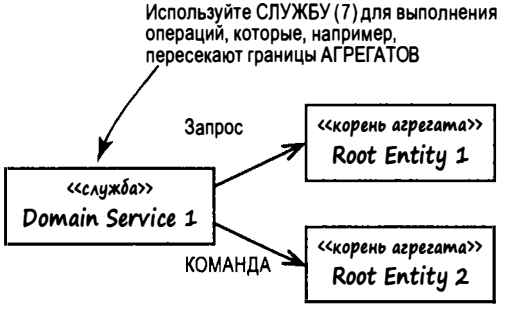


Рис. Р.5. СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ **выполняет операции, характерные для предметной области, в которых могут участвовать многие объекты предметной области**

Для индикации важных событий в предметной области используйте **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**. Их можно моделировать несколькими способами. Если они являются следствием операций определенного АГРЕГАТА, то он сам опубликует СОБЫТИЕ, как показано на рис. Р.6.

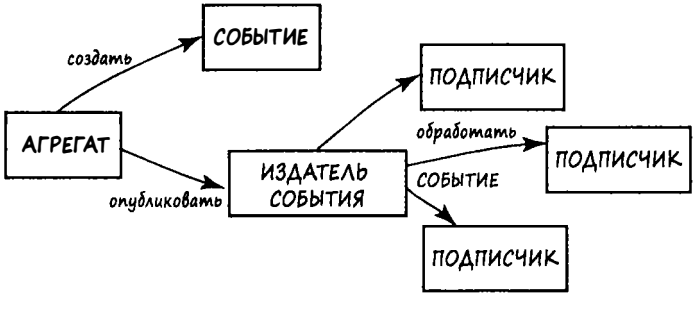


Рис. Р.6. СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ **могут публиковаться** АГРЕГАТАМИ

Очень важно правильно спроектировать **МОДУЛИ (9)**, хотя, как правило, этому вопросу уделяется мало внимания. В своей простейшей форме МОДУЛЬ можно представить в виде пакета в языке Java или пространства имен в языке C#. Напомним, что если попытаться проектировать МОДУЛИ механически, а не в

соответствии с ЕДИНЫМ ЯЗЫКОМ, то они могут причинить вред, а не принести пользу. На рис. Р.7 показано, что МОДУЛИ должны содержать ограниченное количество взаимозависимых объектов предметной области.

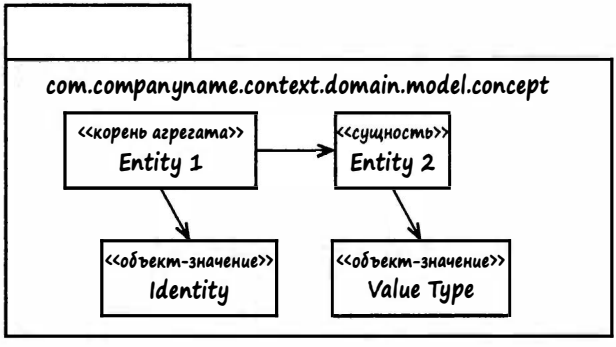


Рис. Р.7. МОДУЛЬ содержит и упорядочивает связанные объекты предметной области

Разумеется, реализация принципов DDD намного шире, чем описанные выше темы, и я даже не пытаюсь перечислить их все в этом разделе. В принципе, вся книга именно об этом. Я думаю, что это руководство станет вам опорой в реализации принципов DDD. Итак, счастливого пути!

Да, просто для того, чтобы дать вам представление о ковбойской логике, привожу один из его образцов.

Ковбойская логика

AJ: Не бойся откусить больше, чем сможешь прожевать. Твой рот намного больше, чем ты думаешь. ;)

LB: Вероятно, ты хотел сказать "ум", AJ. твой ум больше, чем ты думаешь!

Глава 1

Знакомство с DDD

Дизайн — это не то, как предмет выглядит,
а то, как он работает.

Стив Джобс

Все мы боремся за высокое качество разрабатываемого нами программного обеспечения. Определенный уровень качества можно обеспечить с помощью тестов, помогающих нам избежать фатального количества ошибок в поставляемых программах. Впрочем, даже если бы мы научились создавать совершенно безошибочное программное обеспечение, это еще не гарантировало бы, что модель обеспечения качества завершена. Модель программного обеспечения, т.е. *способ*, которым программное обеспечение выражает решение бизнес-проблемы, и в этом случае может быть далекой от совершенства. Поставка программного обеспечения с несколькими дефектами, очевидно, вполне допустима. Однако мы стремимся настолько повысить качество модели программного обеспечения, чтобы она точно отражала поставленные бизнес-цели и наша работа была выполнена на *отлично*.

Подход к разработке программного обеспечения, называемый *предметно-ориентированным проектированием* (Domain-Driven Design — DDD), позволяет обеспечить высокое качество проектирования моделей программного обеспечения. Правильно реализуя подход DDD, можно достичь уровня, на котором проект точно описывает работу программного обеспечения. Цель этой книги — помочь читателям правильно реализовать подход DDD.

Вы можете ничего не знать о принципах DDD, применять их без особого успеха или даже освоить в полной мере. Независимо от вашего уровня освоения DDD, вы, без сомнения, читаете эту книгу, потому что хотите лучше реализовывать его принципы, и вы можете этого достичь. План главы поможет вам уточнить свои цели.

Назначение главы

- Показать, чем подход DDD может быть полезен для ваших проектов и коллективов при решении сложных задач.
- Продемонстрировать, как оценить проект, чтобы понять, следует ли применять принципы DDD.

- Рассмотреть традиционные альтернативы подходу DDD и объяснить, почему они часто порождают проблемы.
- Изложить основы подхода DDD и научить его применению на первых этапах проекта.
- Научить преподносить подход DDD своему руководству, экспертам в предметной области и техническому персоналу.
- Указать сложности, связанные с использованием подхода DDD, и способы их преодоления.
- Продемонстрировать пример работы команды, осваивающей подход DDD.

Чего следует ожидать от подхода DDD? Это не трудоемкий, интенсивный и формальный процесс, препятствующий успеху. Наоборот, вы столкнетесь с методами гибкой разработки программ, которые вам, вероятно, уже знакомы. Кроме приемов гибкой разработки, вы освоите методы, помогающие глубже понять вашу предметную область и в перспективе разрабатывать тестируемые, гибкие, организованные, тщательно продуманные, высококачественные модели программного обеспечения.

Подход DDD предоставляет разработчикам инструменты для *стратегического и тактического моделирования*, необходимые для разработки высококачественного программного обеспечения, соответствующего основным бизнес-целям.

Могу ли я применить принципы DDD

Вы сможете реализовать подход DDD, если у вас есть

- стремление к созданию превосходного программного обеспечения и упорство в достижении цели;
- огромное желание учиться, стремление к совершенству и смелость признавать свои недостатки;
- способность понимать шаблоны программного обеспечения и умение правильно их применять;
- опыт и настойчивость в исследовании проектных альтернатив с помощью испытанных методов гибкой разработки;
- мужество бросить вызов устоявшемуся порядку;
- желание и возможность уделять внимание деталям, эксперименту и поискам нового;
- стремление искать способы делать программы интеллектуальнее и лучше.

Я не собираюсь говорить вам, что научиться этому просто. Честно говоря, кривая обучения может быть крутой. Тем не менее эта книга была написана, чтобы сгладить кривую обучения как можно больше. Моя цель — помочь вам и вашей команде реализовать подход DDD с наибольшей пользой.

Подход DDD не сводится к технологии. Его основные принципы относятся к способам обсуждения, умению слушать, пониманию, открытости и деловой ценности, т.е. к тому, что позволяет концентрировать знания. Если Вы способны *понимать бизнес*, которым занимается ваша компания, то можете как минимум участвовать в процессе разработки модели программного обеспечения, чтобы выработать **ЕДИНЫЙ ЯЗЫК**. Несомненно, вам придется больше узнать о бизнесе, намного больше. Однако вы уже встали на путь освоения подхода DDD, если понимаете принципы своего бизнеса и получаете удовольствие от создания отличного программного обеспечения. Это дает вам надлежащую опору для полного освоения подхода DDD.

Может ли помочь многолетний опыт разработки программного обеспечения? Может. Однако опыт разработки программного обеспечения не заменяет способности слушать и учиться у *экспертов в предметной области*, людей, которые знают больше всех о некоторой специализированной области бизнеса. Вы получите большое преимущество, если сможете общаться с теми, кто редко использует малопонятный технический жаргон или вообще на нем не говорит. Вам придется внимательно слушать и слушать, уважать их точку зрения и признать, что они знают намного больше, чем вы.

Привлечение экспертов в предметной области приносит большую пользу

Вы получите большое преимущество, если привлечете к разработке тех, кто редко выражается на техническом арго или вообще им не пользуется. Поскольку вам придется учиться у них, высока вероятность того, что они также будут учиться у вас.

В подходе DDD вам может больше всего понравиться то, что эксперты в предметной области также вынуждены слушать вас. Вы — такой же член команды, как и они. Как это ни странно, эксперты в предметной области не все знают о своем бизнесе, и им придется больше узнать о нем. В то время как вы будете учиться у них, высока вероятность, что они будут учиться у вас. Ваши вопросы о том, что они знают, скорее всего, выявят то, чего они не знают. Вы будете непосредственно помогать всем членам команды достигать более глубокого понимания бизнеса, *даже формируя сам бизнес*.

Замечательно, когда команда учится и растет вместе. Если вы дадите ей шанс, подход DDD сделает это возможным.

Но у нас нет экспертов в предметной области!

Эксперт в предметной области — это не должность. Это человек, прекрасно знающий род деятельности, которой вы занимаетесь. Вероятно, он владеет огромной информацией о предметной области и может работать проектировщиком и даже продавцом.

То как называется его должность, должно интересовать вас в последнюю очередь. Вам нужны люди, которые знают о предметной области намного больше, чем кто-либо другой, включая вас. *Найдите их. Слушайте. Учитесь. Проектируйте программу.*

Итак, мы начали с простых советов. Однако я не собираюсь утверждать, что технические детали не имеют значения и что успеха можно достичь и без них. Вы должны будете овладеть довольно сложными понятиями *моделирования предметной области* (domain modeling). Но это совсем не означает, что вы столкнетесь с неразрешимыми проблемами. Если вы достаточно умны, чтобы понять, что написано в книге *Head First Design Patterns* [Freeman et al.], одолеть книгу *Design Patterns* [Гамма и др.] и даже разобраться в еще более сложных шаблонах, есть высокая вероятность, что вы сможете освоить подход DDD. Можете мне поверить: я сделаю все, что смогу, чтобы вы меня поняли, независимо от уровня вашей подготовки.

Что такое модель предметной области?

Это программная модель конкретной сферы деловой активности, которой вы занимаетесь. Часто она воплощается в виде объектной модели, в которой объекты обладают данными и поведением в буквальном смысле этих слов.

Создание уникальной, тщательно продуманной модели предметной области, лежащей в основе стратегического приложения или подсистемы, играет решающую роль в практике DDD. Применяя подход DDD, вы получите компактные, очень специализированные модели предметной области и никогда не сможете описать деятельность всего предприятия в виде отдельной, большой модели. Ух ты, так это же хорошо!

Рассмотрим выгоды, которые можно извлечь из подхода DDD. Наверняка в этом списке вы найдете пользу и для себя.

- ***Новичок, младший разработчик***: "Я молод, у меня есть свежие идеи, я энергичен и хочу приносить пользу. Один из проектов, в которых я работаю, раздражает меня. Я не ожидал, что моя первая работа после выпуска из университета сведется к организации передачи данных между почти идентичными объектами. Зачем так усложнять архитектуру, если это — все, что в ней происходит? Что *не так*? Когда я пытаюсь изменять программу, она часто перестает работать. Кто-нибудь понимает на самом деле, что она должна делать?

Например, сейчас я должен добавить новые и сложные функциональные возможности. В итоге я вынужден постоянно упаковывать унаследованные классы в *адаптеры*, чтобы защититься от остальных классов. *Никакого удовольствия*. Я уверен, что есть еще что-то, что я могу сделать, кроме круглосуточного кодирования и отладки программ. Что бы это ни было, я собираюсь выяснить это и овладеть им. Я слышал от других, что существует подход DDD. Это похоже на Банду Четырех, но для предметной области. Отлично!"

Этому парню можно помочь.

- ***Разработчик среднего уровня***: "Несколько месяцев назад меня включили в новую систему. Пришла моя очередь помочь проекту. Я понимаю, как это сделать, но, встречаясь со старшими разработчиками, испытываю недоумение. Иногда вещи кажутся странными, но я не знаю, почему. Я собираюсь изменить способ, которым разрабатывался этот проект. Я знаю, что применения существующих технологий для решения проблемы недостаточно для того, чтобы вырваться далеко вперед. Мне нужен *надежный метод разработки программного обеспечения*, который помог бы мне стать мудрым и опытным разработчиком программного обеспечения. Новый сотрудник, один из старших архитекторов, подал идею насчет чего-то под названием DDD. Я слушаю".

  Вы уже кажетесь старшим разработчиком. Продолжайте читать. Ваша дальновидность будет вознаграждена.

- ***Старший разработчик, архитектор***: "Я использовал DDD в нескольких проектах, но в новой должности делаю это впервые. Мне нравится мощь *тактических шаблонов*, но есть многое, что я мог бы применить помимо них, и в частности — *стратегическое проектирование*. Когда я читал книгу [Эванс], меня больше всего вдохновил ЕДИНЫЙ ЯЗЫК. *Это мощная вещь*. Я много раз пытался убедить своих коллег и руководство принять подход DDD. Эта перспектива увлекла одного младшего разработчика и нескольких коллег среднего и старшего уровней. Руководство отнеслось к этой идее прохладнее. Я лишь недавно поступил на работу в эту компанию, хотя и на ведущую должность. Похоже, организация меньше заинтересована в "прорывных" решениях, чем я думал. Но это не имеет значения. Я не сдаюсь. Вместе с другими разработчиками, которых вдохновила эта идея, *я уверен, что мы можем реализовать этот подход*. Выигрыш будет намного больше, чем ожидается. Мы установим более тесный контакт с настоящими бизнесменами — экспертами в предметной области — и *фактически вложим капитал в наши решения*, а не будем просто повторять итерацию за итерацией".

  Это именно то, что *должен* делать лидер. В этой книге содержится много рекомендаций, помогающих преуспеть в стратегическом проектировании.

- ***Эксперт в предметной области***: "В течение долгого времени я привлекался к выработке IT-решений наших бизнес-проблем. Возможно, я слишком многого хочу, но мне жаль, что разработчики не до конца поняли, чем мы занимаемся. Они всегда снисходительно разговаривают с нами, как с глупыми людьми. Они не понимают, что без нас у них не было работы и повода возиться с компьютерами. Разработчики как-то странно рассказывают о том, что делает наше программное обеспечение. Если мы говорим об А, то они говорят, что А на самом деле называется Б. *Похоже, нам нужен какой-то словарь и справочник каждый раз, когда мы пытаемся сообщить то, что нам нужно.* Если мы не позволяем разработчикам говорить об А, называя его Б, они отказываются от сотрудничества. Таким образом, мы потратили много времени. *Почему программное обеспечение не может работать в соответствии с образом мышления экспертов*?"

  Вы правы. Одна из самых больших проблем — ложная потребность в переводе между деловыми людьми и техническими специалистами. Эта глава предназначена для вас. Как вы увидите, *подход DDD уравнивает в правах вас и разработчиков.*

  Сюрприз! Некоторые разработчики уже научились вас понимать. Надо им помочь.

- ***Менеджер:*** "Мы поставляем программное обеспечение. Оно не всегда приводит к превосходным результатам, и кажется, что его изменения отнимают больше времени, чем должны. Разработчики говорят о некоторой предметной области то одно, то другое. Я не уверен, что мы должны сильно сосредоточиваться на еще одном способе или методологии, как будто она представляет собой золотой ключик. Я слышал все это тысячу раз. Мы попробуем новый способ, затем мода пройдет, и мы вскоре вернемся к старому проверенному методу. Я считаю, что мы должны сохранять выбранный курс и прекратить мечтания, но команда продолжает преследовать меня. Они упорно трудились, и поэтому я должен их слушать. *Они — умные люди и заслуживают шанса улучшить ситуацию* до того, как начнется работа. Я дал им некоторое время, чтобы они поучились и освоили новый подход, прежде чем я обращусь к высшему руководству за поддержкой. Я думаю, что мог бы получить одобрение, если бы смог убедить своего босса в справедливости требований команды *вложить средства в приобретение специального программного обеспечения и централизовать знания о бизнесе.* Моя работа стала бы легче, *если бы я смог наладить доверительные отношения и сотрудничество между моими командами и бизнес-экспертами.* Так или иначе это то, что я, по слухам, могу сделать".

  Хороший менеджер!

Кем бы вы ни были, вот важное предостережение. Чтобы преуспеть с подходом DDD, *необходимо многому научиться*, действительно многому. Тем не менее это не должно быть большой проблемой. Вы умны и должны учиться постоянно. И все же все мы оказываемся перед проблемой.

> *Лично я всегда готов учиться, хотя мне не всегда нравится, когда меня учат.*
>
> — Уинстон Черчилль

Собственно, книга начинается с этого места. Я попытался сделать обучение как можно более приятным, излагая самые важные факты, которые важно знать для того, чтобы с успехом реализовать подход DDD.

Вы можете спросить "Почему я должен применять подход DDD?" Справедливый вопрос.

Почему необходимо применять подход DDD

Фактически я уже привел довольно серьезные причины, по которым подход DDD имеет так много практического смысла. Рискуя нарушить принцип DRY ("Don't repeat yourself" — "Не повторяйся"), я все же повторю их еще раз и добавляю к более ранним аргументам. Кто-либо слышит эхо?

- Объедините экспертов в предметной области и разработчиков в одну команду, которая производит программное обеспечение, интересующее бизнес, а не только программистов. Это не значит просто терпеть противоположную группу. Это значит становиться одной сплоченной, дружной командой.
- Выражение "имеет смысл для бизнеса" означает инвестирование в программное обеспечение, которое написали бы сами эксперты в предметной области, если бы сами были программистами.
- Вы можете учить самих бизнесменов. Ни эксперт в предметной области, ни топ-менеджер, никто не знает всего о бизнесе. Это постоянный процесс открытий, который со временем становится все более увлекательным. В рамках подхода DDD каждый чему-то учится, потому что каждый вносит свой вклад в обсуждение.
- Ключевым моментом является централизация знаний, потому что благодаря этому можно гарантировать, что программное обеспечение не будет замкнуто в рамках "тайного знания" (tribal knowledge), доступного только избранным, которыми, как правило, являются разработчики.

- Между экспертами в предметной области, разработчиками программного обеспечения и программным обеспечением перевод не нужен. Возможно, потребуется лишь несколько уточнений. Это обеспечивает общий, понятный всем язык, на котором говорят все члены команды.
- Проект — это программа, а программа — это проект. Проект описывает, как работает система. Выявление наилучшей структуры программы является результатом применения экспериментальных моделей, быстро реализуемых с помощью гибких процессов проектирования.
- Подход DDD обеспечивает надежные методы разработки программного обеспечения, предназначенные для стратегического и тактического проектирования. Стратегическое проектирование помогает определить, куда направить самые крупные инвестиции в программное обеспечение, что сделать для усиления существующего программного обеспечения, чтобы система работала максимально быстро и безопасно, и кто должен быть вовлечен в проект. Тактическое проектирование помогает разработать целостную элегантную модель решения с помощью испытанных, проверенных стандартных блоков программного обеспечения.

Как любая продуманная прибыльная инвестиция, подход DDD требует определенных затрат и усилий со стороны команды. Рассмотрение типичных проблем, возникающих при проектировании любых программ, приводит к осознанию необходимости инвестирования в надежный метод разработки программного обеспечения.

Обеспечить бизнес-ценность иногда трудно

Разработка программного обеспечения, обеспечивающего истинную бизнес-ценность, отличается от разработки обычного коммерческого программного обеспечения. Программы, создающие истинные бизнес-ценности, согласованы со стратегическими бизнес-инициативами и позволяют принимать решения, дающие явное конкурентное преимущество. Такое программное обеспечение относится не к технологии, а к бизнесу.

Знания о бизнесе никогда не бывают централизованными. Команды разработчиков должны сбалансировать и определить приоритеты потребностей и запросов многочисленных участников бизнес-процесса и привлечь к работе множество людей, имеющих разные навыки, чтобы выработать функциональные и нефункциональные требования к программному обеспечению. Как, собрав всю эту информацию, команды могут бы уверены, что любое выявленное требование

приносит истинную бизнес-ценность? И что такое бизнес-ценности, как их обнаружить, определить их приоритет и реализовать?

Одно из худших проявлений несогласованности усилий при разработке коммерческого программного обеспечения — разрыв между экспертами в предметной области и разработчиками программного обеспечения. Вообще говоря, настоящие эксперты в предметной области сосредоточены на достижении бизнес-ценности. С другой стороны, разработчики программного обеспечения обычно тяготеют к технологии и техническим решениям бизнес-проблем. Это не значит, что у разработчиков программного обеспечения неправильная мотивация; это просто значит, что технологии и технические решения захватывают почти все их внимание. Даже когда разработчики программного обеспечения привлекают экспертов в предметной области, их сотрудничество в основном имеет поверхностный характер, а программное обеспечение, которое они проектируют, часто является результатом перевода/отображения мыслей и действий экспертов в интерпретации разработчиков. Созданное в итоге программное обеспечение, как правило, мало похоже на реализацию ментальной модели экспертов в предметной области. Со временем эта несогласованность приводит к убыткам. Перевод знаний о предметной области в программное обеспечение теряется, когда разработчики переходят в другие проекты или уходят из компании.

Другая, хотя и связанная с предыдущей, проблема возникает, когда один или несколько экспертов в предметной области не согласны друг с другом. Это происходит довольно часто, потому что эксперты имеют разный объем опыта в конкретной моделируемой предметной области или работают в связанных, но разных предметных областях. Часто многие так называемые "эксперты в предметной области" не имеют опыта работы в предметной области, в которой они являются скорее бизнес-аналитиками, хотя от них ожидают, что они дадут правильное направление дискуссии. Если эту ситуацию вовремя не исправить, вместо четких ментальных моделей вы получите нечеткие рекомендации, которые приведут к конфликтующим моделям программного обеспечения.

Еще хуже, если технический подход к разработке программного обеспечения по существу искажает выполнение бизнес-функций. На основе анализа разных сценариев хорошо известно, что программное обеспечение для планирования ресурсов предприятия (enterprise resource planning — ERP) часто изменяет выполнение всех бизнес-операций в организации, чтобы они соответствовали функциям системы ERP. Общую стоимость владения системой ERP невозможно вычислить в терминах стоимости лицензии и ее поддержки. Реорганизация и разрушение бизнеса может стоить намного дороже, чем два эти материальных фактора. Аналогичная ситуация складывается, когда команды разработчиков программного обеспечения навязывают бизнесу функции нового программного обеспечения.

Это может оказать негативное влияние на бизнес, его клиентов и партнеров. Более того, такая техническая интерпретация становится и ненужной, и устранимой, если применять проверенные методы разработки программного обеспечения. Решение — это главная инвестиция.

Чем может помочь подход DDD

DDD — это подход к разработке программного обеспечения, который сосредоточен на трех основных аспектах.

1. Подход DDD объединяет экспертов в предметной области и разработчиков программного обеспечения для создания системы, отражающей ментальную модель экспертов в предметной области. Это не значит, что все усилия направляются на моделирование "реального мира". Подход DDD приводит к созданию модели, наиболее полезной для бизнеса. Иногда полезные и реалистичные модели совпадают, но если они не совпадают, подход DDD отдает предпочтение полезной модели.

   Усилия экспертов в предметной области и разработчиков программного обеспечения направлены на совместную разработку ЕДИНОГО ЯЗЫКА в сферах бизнеса, на которые нацелено моделирование. ЕДИНЫЙ ЯЗЫК разрабатывается на основе командного консенсуса. Команда высказывает свои мысли и описывает модель именно на этом языке. Стоит напомнить еще раз, что команда состоит из экспертов в предметной области и разработчиков программного обеспечения. Она никогда не делится на "мы" и "они". Она — это всегда только "*мы*". Это главная бизнес-ценность, позволяющая сохранить бизнес-знания и команду на этапе относительно короткого начального периода разработки, в результате которого поставляется несколько первых версий программного обеспечения.

   Это позволяет уравнять экспертов в предметной области, которые вначале были не согласны друг с другом, и тех, кто просто мало знает о ней. Кроме того, это усиливает сплоченную команду, благодаря распространению глубоких знаний о предметной области среди всех членов команды, включая разработчиков программного обеспечения. Это можно сравнить с тренингами, которые каждая компания должна проводить для обучения своих сотрудников.

2. Подход DDD направлен на выработку стратегических инициатив бизнеса. Хотя такое стратегическое проектирование естественно включает в себя технический анализ, оно в большей степени ориентировано на стратегическое направление бизнеса. Оно помогает определить оптимальные организационные связи между командами и создать систему раннего

оповещения для распознавания ситуаций, в которых эти отношения могут вызвать крах программного обеспечения или даже всего проекта. Технические аспекты стратегического проектирования нацелены на разграничение системных и деловых вопросов, относящихся к *бизнес-службам* (business-level services). Это позволяет точнее обосновать общую *сервис-* или *бизнес-ориентированную архитектуру* (service-oriented architecture, business-driven architecture).

3. Подход DDD соответствует реальным техническим требованиям, которые предъявляются к программному обеспечению, благодаря инструментам тактического проектирования, позволяющим выполнять анализ и создавать работоспособные программы. Эти инструменты тактического проектирования позволяют экспертам создавать программное обеспечение, которое правильно кодирует ментальную модель экспертов в предметной области, допускает подробное тестирование, устойчиво к ошибкам (доказуемое высказывание), функционирует в соответствии с соглашениями об уровне услуг (service-level agreements — SLAs), а также допускает масштабирование и распределенные вычисления. Самые удачные реализации подхода DDD обычно учитывают десятки и более высокоуровневых архитектурных и низкоуровневых программных факторов, концентрируя внимание на выделении реальных бизнес-правил и инвариантов данных, одновременно защищая их от ошибок.

Используя этот подход к разработке программного обеспечения, вы и ваша команда можете достичь успеха в достижении настоящих бизнес-ценностей.

Столкновение со сложностью предметной области

Как правило, подход DDD используется в областях, наиболее важных для бизнеса. Никто не хочет тратить средства на то, что можно легко заменить. *Все хотят инвестировать в нетривиальные, сложные, наиболее значимые и важные решения, которые обещают наибольшую прибыль.* Именно поэтому мы называем такую модель **СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) (2)**. Этот проектный шаблон и второй *по значимости* шаблон **СЛУЖЕБНЫЕ ПОДОБЛАСТИ (SUPPORTING SUBDOMAINS) (2)** заслуживают наибольшего внимания. Теперь необходимо объяснить, что означает слово *сложность*.

Используйте подход DDD для упрощения, а не для усложнения

Используйте подход DDD для моделирования сложной предметной области как можно более простым способом. Никогда не используйте подход DDD, чтобы усложнить решение.

Что именно считается сложным, зависит от бизнеса. Разные компании сталкиваются с разными проблемами, находятся на разных стадиях развития и имеют разные возможности для разработки программного обеспечения. По этой причине легче объяснить значение слова "*нетривиальный*", чем "*сложный*". Итак, *ваша команда и руководство должны определить, заслуживает ли система, которую вы планируете разработать, применения подхода DDD.*

Оценочная ведомость DDD. Определите с помощью табл. 1.1, подходит ли ваш проект для применения подхода DDD. Если строка в таблице соответствует описанию вашего проекта, поставьте в правом столбце соответствующее число. Просуммируйте все баллы вашего проекта. Если сумма равна 7 или больше, серьезно задумайтесь об использовании подхода DDD.

Эта анкета может привести вашу команду к следующим выводам.

Очень плохо, что мы не сможем быстро перестроиться, если столкнемся с непредвиденной сложностью.

Конечно, но это означает лишь, что мы должны очень точно определить степень сложности проекта в самом начале. Это сэкономит нам массу времени, денег и нервов.

Приняв основное архитектурное решение и тщательно проработав несколько пользовательских сценариев, мы обычно строго придерживаемся их. Мы должны сделать правильный выбор.

Если хотя бы одно из этих утверждений соответствует высказываниям членов вашей команды, значит, у вас правильный критический стиль мышления.

Анемия и потеря памяти

Анемия — это серьезное заболевание с опасными последствиями. Когда автор выбирал название **АНЕМИЧНАЯ МОДЕЛЬ ПРЕДМЕТНОЙ ОБЛАСТИ (ANEMIC DOMAIN MODEL)** [Fowler, Anemic], *он имел в виду негативную характеристику* и совсем не хотел сказать, что если модель предметной области очень слабая, т.е. не имеет мощного специфического поведения, то это в каких-то ситуациях может оказаться полезным. Как ни странно, такие модели встречаются в нашей индустрии на каждом шагу. Проблема заключается в том, что большинство разработчиков, кажется, думают, что это вполне допустимо, и даже не знают об опасностях, которым подвергаются их системы. Это реальная проблема.

Таблица 1.1. Анкета DDD

Набрал ли ваш проект 7 или больше баллов			
Если ваш проект...	***Оценка***	***Обоснование***	***Баллы***
Если ваш проект полностью ориентирован на обработку данных и требует решения в стиле CRUD, в котором каждая операция по существу представляет собой запрос к базе данных на создание (Create), чтение (Read), изменение (Update) или удаление (Delete), то подход DDD вам не нужен. Вашей команде нужен просто редактор таблиц базы данных. Иначе говоря, если вы готовы доверить своим пользователям непосредственные операции вставки в таблицу, ее изменения, а иногда удаления, то вам не понадобится даже пользовательский интерфейс. Это нереальный сценарий, но теоретически важный. Если для создания решения вы можете использовать простые инструменты для разработки баз данных, не тратьте время и деньги вашей компании на DDD	0	На первый взгляд, это элементарно, но обычно не так просто определить, что просто, а что сложно. Это совсем не значит, что любое приложение, не являющееся чистым решением CRUD, заслуживает траты времени и денег, связанных с использованием подхода DDD. Таким образом, возможно, нам нужны другие показатели, чтобы провести разделительную черту между простым и сложным...	
Если ваша система предусматривает выполнение не более 30 бизнес-операций, то, скорее всего, она является относительно простой. Это может означать, что ваше приложение может хранить не более 30 пользовательских историй или потоков пользовательских сценариев, каждый из которых имеет самую простую бизнес-логику. Если вы можете быстро и легко разработать такое приложение, используя инструменты Ruby on Rails или Groovy and Grails, и не испытываете неудобства от недостатка мощи и средств управления сложностью и версиями, ваша система, вероятно, не требует использования подхода DDD	1	Для простоты я говорю о 25–30 отдельных бизнес-методах, а не 25–30 полноценных служебных интерфейсах, каждый из которых содержит множество методов. Второй вариант может быть сложным	
Итак, будем говорить, что 30–40 пользовательских историй или потоков пользовательских сценариев могут создать сложность. В этом случае ваша система может оказаться на территории DDD.	2	*Внимание:* очень часто сложность распознается не сразу. *Разработчики программного обеспечения очень часто недооценивают сложность и уровень усилий, требующихся для создания приложения.* Даже если мы хотим программировать с помощью	

Набрал ли ваш проект 7 или больше баллов			
Если ваш проект...	***Оценка***	***Обоснование***	***Баллы***
		инструментов Rails или Grails, это еще не значит, что мы должны это делать. В долговременной перспективе это может оказаться скорее вредным, чем полезным	
Даже если приложение пока не кажется сложным, не увеличится ли сложность в будущем? Возможно, вы не узнаете этого, пока пользователи не начнут реально работать с вашим приложением. Истинную ситуацию можно распознать с помощью вопросов, указанных в столбце "Обоснование». Здесь следует быть осторожным. Если есть хотя бы малейший шанс, что приложение может в будущем иметь как минимум умеренную сложность — в этот момент полезно быть параноиком, — то, скорее всего, она такой и станет. Это ведет к DDD	3	Здесь целесообразно рассмотреть более сложные пользовательские сценарии вместе с экспертами в предметной области и посмотреть, куда они ведут. Правда ли, что эксперты в предметной области . . . 1. ...уже спрашивали о более сложных функциональных возможностях? Если да, то это, вероятно, является признаком того, что уже сейчас или в ближайшем будущем проект станет слишком сложным для подхода CRUD; 2. ...настолько не заинтересованы в функциональных свойствах, что не хотят даже говорить о них? Тогда система, скорее всего, не является сложной	
Функциональные возможности приложения будут часто изменяться в течение нескольких лет, и вы не можете предсказать, насколько простыми будут эти изменения	4	Подход DDD может помочь вам справиться со сложностью рефакторинга вашей модели в будущем	
Вы не понимаете шаблон **ПРЕДМЕТНАЯ ОБЛАСТЬ (2)**, потому что он новый. Насколько вам известно, до вас никто ничего подобного не делал. Скорее всего, это — признак сложности или по крайней мере заслуживает внимания аналитиков, чтобы выяснить уровень сложности	5	Вы планируете работать с экспертами в предметной области и экспериментировать с моделями, чтобы уточнять их. Если, кроме того, вы отметили один или несколько вышеуказанных сценариев, то используйте подход DDD	

Интересовались ли вы когда-нибудь, как чувствует себя ваша модель: не устала ли она, не стала ли апатичной, забывчивой, неповоротливой и не нуждается ли в допинге? Если вы вдруг почувствовали приступ технической ипохондрии, обследуйтесь. Либо вам станет легче, либо ваши наихудшие опасения подтвердятся. Для обследования используйте табл. 1.2.

Таблица 1.2. История болезни вашей модели предметной области

	Да/Нет
Имеет ли программное обеспечение, которое вы называете "модель предметной области", в основном методы-получатели (get-) и установщики (set-), но не имеет или почти не имеет бизнес-логики, т.е. состоит из объектов, которые в основном являются хранилищем значений атрибутов?	
Реализована ли основная бизнес-логика в компонентах программного обеспечения, которые часто используют вашу "модель предметной области" с помощью интенсивных вызовов методов получателей и установщиков? Возможно, вы называете этот клиентский уровень вашей "модели предметной области" **СЛУЖЕБНЫМ УРОВНЕМ (SERVICE LAYER)** или **ПРИКЛАДНЫМ УРОВНЕМ (APPLICATION LAYER) (4, 14)**. Если это описание относится к вашему пользовательскому интерфейсу, напишите в следующем столбце "Да", а на доске напишите тысячу раз, что вы никогда, никогда так больше делать не будете.	
Подсказка: правильный ответ — либо "Да" на оба вопроса, либо "Нет" на оба вопроса.	

Ну как?

Если вы ответили "Нет" на оба вопроса, с вашей предметной областью все в порядке.

Если же вы ответили на оба вопроса "Да", то ваша "модель предметной области" очень-очень больна. Это анемия. Но есть и хорошая новость — прочитав эту книгу, вы сможете ее вылечить.

Если на один вопрос вы ответили "Да", а на другой — "Нет", то вы либо не хотите признаваться, либо бредите, либо у вас другое неврологическое расстройство, вызванное анемией. Что делать, если вы даете противоречивые ответы? Вернитесь к первому вопросу и снова проверьте себя. Помните, что на оба вопроса вы должны дать уверенный ответ "Да!"

Как пишет Фаулер [Fowler, Anemic], АНЕМИЧНАЯ МОДЕЛЬ ПРЕДМЕТНОЙ ОБЛАСТИ плоха тем, что вы вынуждены платить высокую цену за разработку модели предметной области, не получая от этого почти никакой выгоды. Например, из-за объектно-реляционного несоответствия разработчики таких "моделей предметной области" тратят много времени и усилий на отображение объектов в постоянное хранилище и наоборот. Это высокая цена, которая никогда не окупится.

Я бы добавил, что вам нужна совсем не модель предметной области, а просто модель данных, выполняющая проекцию реляционной модели (или другой базы данных) в объекты. Говорить, что можно еще больше приблизиться к шаблону **АКТИВНОЙ ЗАПИСИ** (**ACTIVE RECORD**) [Fowler, P of EAA], может только обманщик. Вы можете упростить архитектуру, но не задавайтесь, а просто признайтесь, что на самом деле используете форму **СЦЕНАРИЯ ТРАНЗАКЦИЙ** (**TRANSACTION SCRIPT**) [Fowler, P of EAA].

Причины анемии

Если АНЕМИЧНАЯ МОДЕЛЬ ПРЕДМЕТНОЙ ОБЛАСТИ — это лишь хилый результат плохо выполненного проектирования, то почему ее так часто используют, считая, что она вполне приемлема? Очевидно, потому, что она отражает менталитет процедурного программирования, но я не думаю, что это главная причина. Большая часть нашей индустрии создана программистами, которые просто копируют образцы кода, что, в принципе, неплохо, если образцы качественные. Однако часто эти образцы в основном предназначены для максимально упрощенной демонстрации определенной концепции или функциональной возможности интерфейса прикладного программирования (API), без учета принципов хорошего проектирования. Каждый день программисты копируют чрезмерно упрощенные образцы кода, которые обычно демонстрируют большое количество методов получателей и установщиков, ни на секунду не задумываясь о проектировании.

Есть еще одна, более старая причина. Большая часть вины за состояние, в котором мы сейчас пребываем несет древняя история языка Visual Basic компании Microsoft. Я не утверждаю, что Visual Basic был плохим языком и интегрированной средой разработки (IDE), потому что он всегда демонстрировал высокую эффективность и определенным образом влиял на развитие индустрии в правильном направлении. Конечно, некоторым программистам удалось избежать этого влияния, но косвенно язык Visual Basic повлиял практически на каждого разработчика программного обеспечения. Просто посмотрите на хронологическую табл. 1.3.

Я говорю о влиянии свойств и страниц свойств, лежащих в основе методов получателей и установщиков, ставших такими популярными среди разработчиков форм на Visual Basic. Программисту достаточно было поместить на форму несколько элементов управления, заполнить их страницу свойств — и *все*! Он получал функционирующее Windows-приложение. Это занимало всего несколько минут по сравнению с несколькими днями, которые требовались для программирования аналогичного приложения для Windows API с помощью языка C.

Какое все это имеет отношение к АНЕМИЧНЫМ МОДЕЛЯМ ПРЕДМЕТНОЙ ОБЛАСТИ? *Стандарт JavaBean изначально был предназначен для облегчения разра-*

ботки средств визуального программирования на языке Java. Его целью был перенос возможностей Microsoft ActiveX на платформу Java. Он давал надежду, что возникнет полноценный рынок сторонних пользовательских элементов управления, аналогичных Visual Basic. Вскоре практически все каркасы или библиотеки переключились на JavaBean. Они содержали большую часть комплекта Java SDK/JDK и такие библиотеки, как Hibernate. С точки зрения подхода DDD *библиотека Hibernate предназначена для хранения моделей предметной области.* С появлением платформы .NET тенденция продолжилась.

Таблица 1.3. Хронологическая таблица перехода полноценного поведения к анемии

1980-е	1991	1992–1995	1996	1997	1998–до сих пор
Благодаря языкам Smalltalk и C++ большое влияние приобретают объекты	В языке Visual Basic появляются свойства и страницы свойств	Широкое распространение получают инструменты визуального программирования и среды IDE Visual	Выпуск Java JDK 1.0	Появляется спецификация JavaBean	Возникает лавина инструментов для Java, основанных на рефлексии, и для платформы .NET, основанных на свойствах

Интересно, что на первых порах любая модель предметной области, хранившаяся с помощью библиотеки Hibernate, должна была иметь открытые методы получателей и установщиков для каждого простого постоянно хранимого атрибута и сложной ассоциации в каждом объекте предметной области. Это значило, что если бы вы захотели разработать свой объект POJO (Plain Old Java Object) с полнофункциональным интерфейсом, то должны были бы раскрыть его внутреннюю структуру, чтобы библиотека Hibernate могла хранить и восстанавливать ваши объекты предметной области. Конечно, вы могли бы попытаться скрыть открытый интерфейс JavaBean, но большая часть разработчиков об этом не думали и даже не понимали, почему они должны это делать.

Должен ли я использовать средства объектно-реляционного отображения в рамках подхода DDD?

Предыдущая критика библиотеки Hibernate относилась к прошлому. В настоящее время библиотека Hibernate уже поддерживает скрытые методы получателей и установщиков и даже прямой доступ к полям. В последующих главах я покажу, как избежать анемии в моделях, используя библиотеку Hibernate и другие механизмы хранения. Так что не беспокойтесь.

Большинство, если не все, веб-каркасы также работают в соответствии со стандартом JavaBean. Если вы хотите заполнить свои веб-страницы Java-объектами, то должны соблюдать спецификации JavaBean. Если вы хотите заполнить Java-объект формами HTML и передать их на сервер, то ваш Java-объект должен соответствовать спецификациям JavaBean.

В настоящее время практически каждый каркас требует использования открытых свойств простых объектов. Большинство разработчиков подвергаются воздействию анемичных моделей на всех предприятиях. Признайте это. Вы ведь уже обжигались о них, не так ли? В результате сложилась ситуация, которую можно назвать просто: *повсеместная анемия*.

Как анемия влияет на вашу модель

Ладно, допустим, вы согласились с тем, что я сказал правду, и она вас раздражает. Как *повсеместная анемия* должна реагировать на *потерю памяти*? Что вы обычно видите, когда читаете клиентский код АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ (например, **ПРИКЛАДНОЙ СЛУЖБЫ** (**APPLICATION SERVICE**) **(4, 14)**, аналога СЦЕНАРИЯ ТРАНЗАКЦИЙ (TRANSACTION SCRIPT)? Рассмотрим рудиментарный пример.

```
@Transactional
public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
    String streetAddress1, String streetAddress2,
    String city, String stateOrProvince,
    String postalCode, String country,
    String homePhone, String mobilePhone,
    String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }

    customer.setCustomerFirstName(customerFirstName);
    customer.setCustomerLastName(customerLastName);
    customer.setStreetAddress1(streetAddress1);
    customer.setStreetAddress2(streetAddress2);
    customer.setCity(city);
    customer.setStateOrProvince(stateOrProvince);
    customer.setPostalCode(postalCode);
    customer.setCountry(country);
    customer.setHomePhone(homePhone);
    customer.setMobilePhone(mobilePhone);
```

```
    customer.setPrimaryEmailAddress(primaryEmailAddress);
    customer.setSecondaryEmailAddress (secondaryEmailAddress);

    customerDao.saveCustomer(customer);
}
```

Пример намеренно сделан простым

Признаюсь, этот пример не относится к очень интересной предметной области, но он может помочь нам проверить неидеальный проект и выяснить, как его можно намного улучшить с помощью рефакторинга. Следует понимать, что этот пример не поможет нам более эффективно сохранять данные. Он касается лишь разработки модели программного обеспечения, которая добавляет ценности к вашему бизнесу, даже если сам пример кажется не слишком ценным.

Что делает этот код? На самом деле он очень универсальный. Он сохраняет объект класса `Customer` независимо от того, новый он или уже существовал ранее. Он сохраняет объект класса `Customer` независимо от того, изменилась ли фамилия клиента или он переехал по новому адресу. Он сохраняет объект класса `Customer` независимо от того, изменился ли телефонный номер клиента или он впервые купил мобильный телефон, или и то, и другое. Он сохраняет объект класса `Customer`, даже если клиент перешел с использования Juno на Gmail или переменил место работы и получил новый адрес электронной почты. Ух ты, да это классный метод!

Но так ли это? На самом деле мы не представляем, в какой бизнес-ситуации используется метод `saveCustomer()`, — во всяком случае, не совсем. Почему этот метод создан первым? Кто-нибудь помнит его первичное предназначение и все аргументы в его пользу? Скорее всего, эти воспоминания были потеряны уже через несколько недель или месяцев после того, как этот метод было создан, а затем модифицирован. Вы мне не верите? Посмотрите на следующую версию этого же метода.

```
@Transactional
    public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
    String streetAddress1, String streetAddress2,
    String city, String stateOrProvince,
    String postalCode, String country,
    String homePhone, String mobilePhone,
    String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }
```

```
    if (customerFirstName != null) {
        customer.setCustomerFirstName(customerFirstName);
    }

    if (customerLastName != null) {
        customer.setCustomerLastName(customerLastName);
    }

    if (streetAddress1 != null) {
        customer.setStreetAddress1(streetAddress1);
    }

    if (streetAddress2 != null) {
        customer.setStreetAddress2(streetAddress2);
    }

    if (city != null) {
        customer.setCity(city);
    }

    if (stateOrProvince != null) {
        customer.setStateOrProvince(stateOrProvince);
    }

    if (postalCode != null) {
        customer.setPostalCode(postalCode);
    }

    if (country != null) {
        customer.setCountry(country);
    }

    if (homePhone != null) {
        customer.setHomePhone(homePhone);
    }

    if (mobilePhone != null) {
        customer.setMobilePhone(mobilePhone);
    }

    if (primaryEmailAddress != null) {
        customer.setPrimaryEmailAddress(primaryEmailAddress);
    }

    if (secondaryEmailAddress != null) {
        customer.setSecondaryEmailAddress (secondaryEmailAddress);
    }

    customerDao.saveCustomer(customer);
}
```

Здесь следует отметить, что данный пример не является плохим сам по себе. Многие программы, структурирующие данные, со временем становятся довольно

сложными и "съедают" большую часть бизнес-логики. Я пощадил вас и не показал худшие свойства этого примера, но, возможно, вы видите их сами.

Теперь каждый параметр, отличающийся от объекта `customerId`, является необязательным. Теперь этот метод можно использовать для того, чтобы сохранить объект класса `Customer` по крайней мере в десятке бизнес-ситуаций! Но хорошо ли это на самом деле? Как реально протестировать этот метод, чтобы убедиться, что он не будет сохранять объект класса `Customer` в неправильных ситуациях?

Чтобы не углубляться в излишние детали, отметим, что этот метод может чаще работать неправильно, чем правильно. Возможно, существуют ограничения базы данных, предотвращающие сохранение полностью некорректного состояния, но для проверки этого обстоятельства нам нужно взглянуть на базу данных. Скорее всего, вы потратите какое-то время на ментальное отображение атрибутов Java в имена столбцов. Закончив эту часть работы, вы обнаружите, что ограничения базы данных отсутствуют или неполны.

Для того чтобы понять, почему этот метод реализован именно так, вам придется проверить довольно большое количество клиентов (не учитывая тех, которые были добавлены после того, как был завершен пользовательский интерфейс для автоматической работы с удаленными клиентами) и сравнить версии исходного кода. В поисках ответа вы поймете, что никто не в состоянии объяснить, почему этот метод работает именно так, а не иначе, и сколько раз его применяли правильно. Для того чтобы это понять, потребуется несколько часов или дней.

Ковбойская логика

AJ: Этот мужик сам не знает, чего хочет: то ли картошку грузить, то ли кататься на роликах в стаде бизонов.

Эксперты в предметной области не могут помочь в этом, поскольку для того, чтобы понимать код, надо быть программистом. Даже если бы один или два эксперта разбирались в программировании или по крайней мере понимали код, они так же не смогли бы сказать ничего определенного о предназначении этого кода. Учитывая все эти обстоятельства, следует ли нам смело изменять этот код, а если да, то как?

Здесь возникают по меньшей мере три большие проблемы.

1. Очень мало известно о предназначении интерфейса `saveCustomer()`.
2. Сама реализация интерфейса `saveCustomer()` увеличивает скрытую сложность.

3. Объект предметной области `Customer` на самом деле совсем не является объектом. Это простой контейнер данных.

Назовем эту незавидную ситуацию *потерей памяти, вызванной анемией* (*anemia-induced memory loss*). Она возникает каждый раз, когда используется такое нечеткое, совершенно субъективное программное "проектирование".

Минуточку!

В этом месте некоторые читатели могут подумать: "Наши проекты на самом деле никогда не покидают доску. Мы просто рисуем какую-то структуру и, достигнув согласия, переходим к ее реализации. Ужас!"

Если это так, не пытайтесь отличить проектирование от реализации. Помните, что в подходе DDD *проект — это код, а код — это проект*. Иначе говоря, диаграммы, нарисованные на доске, не являются проектом. Это просто способ обсуждения проблем, связанных с моделью.

Оставайтесь с нами, и вы узнаете, как перенести идеи с доски в проект и заставить их работать.

Испытав беспокойство по поводу такого стиля программирования, вы должны заинтересоваться, как создать более качественный проект. Спешу вас утешить: вы можете достичь успеха и выразить в своем коде четкий, тщательно продуманный проект.

Как применять DDD

Давайте на время прекратим тягостные дискуссии о реализации и рассмотрим одно из наиболее впечатляющих свойств DDD — ЕДИНЫЙ ЯЗЫК. Это один из двух основных столпов DDD, вторым является **ОГРАНИЧЕННЫЙ КОНТЕКСТ (2)**. Ни один из них не может существовать без другого.

Термины в контексте

Пока будем представлять себе ОГРАНИЧЕННЫЙ КОНТЕКСТ как концептуальную границу вокруг отдельного приложения или конечной системы. Каждое использование определенного термина, фразы или предложения предметной области, т.е. ЕДИНОГО ЯЗЫКА, внутри этой границы имеет конкретное контекстное значение. Любое использование термина за пределами этой границы может иметь и часто имеет другой смысл. Подробно ОГРАНИЧЕННЫЙ КОНТЕКСТ рассматривается в главе 2.

ЕДИНЫЙ ЯЗЫК

ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE) — это коллективный язык терминов. Он совместно используется экспертами в предметной области и разработчиками. Фактически на нем говорят все участники команды, разрабатывающей проект. Ваша роль в команде не имеет значения. Раз вы — член команды, значит, используете для описания проекта ЕДИНЫЙ ЯЗЫК.

Вы, наверное, подумали...

"ЕДИНЫЙ ЯЗЫК — *это, очевидно, язык бизнеса*".

А вот и нет!

"Разумеется, он должен следовать стандартной промышленной терминологии".

Нет, вовсе нет.

"Ясно, что это язык, на котором говорят эксперты в предметной области".

Извините, нет.

"ЕДИНЫЙ ЯЗЫК — *это коллективный язык, выработанный командой, состоящей как из экспертов предметной области, так и из разработчиков*".

Да. Вот теперь вы правы!

Естественно, эксперты в предметной области больше влияют на ЕДИНЫЙ ЯЗЫК, потому что они лучше всех знают данную часть бизнеса и следуют промышленным стандартам. Однако ЕДИНЫЙ ЯЗЫК *в основном выражает то, как сам бизнес мыслит и действует.* Кроме того, довольно часто несколько экспертов в предметной области оказываются несогласными с понятиями и терминами, причем относительно некоторых из них они могут ошибаться, потому что никто не может знать все. Итак, поскольку эксперты и разработчики совместно работают над моделью предметной области, они должны достичь консенсуса и выработать компромиссное решение о наилучшем ЕДИНОМ ЯЗЫКЕ проекта. Качество ЕДИНОГО ЯЗЫКА не должно быть предметом компромисса, в него должны быть включены только самые точные понятия, термины и значения. Однако первоначальный консенсус не вечен. С течением времени по мере появления больших и маленьких открытий ЕДИНЫЙ ЯЗЫК растет и развивается, как и любой другой естественный язык.

Это не уловка, чтобы заставить разработчиков говорить на одном языке с экспертами в предметной области. Это не бизнес-жаргон, навязанный разработчикам. Это настоящий язык, созданный целостной командой — экспертами в предметной области, разработчиками, бизнес-аналитиками и всеми, кто вовлечен в создание системы. Разработку ЕДИНОГО ЯЗЫКА можно начинать с включения в него терминов естественного языка, на котором говорят эксперты в предметной области, но этим ограничиваться нельзя, потому что язык со временем должен развиваться. Достаточно напомнить, что когда эксперты

в предметной области привлекаются к выработке ЕДИНОГО ЯЗЫКА, они часто выражают несогласие даже в отношении тех терминов и значений, которые считались общепризнанными.

В табл. 1.4 показана не только модель применения вакцины от гриппа в виде кода, но и выражения ЕДИНОГО ЯЗЫКА, которые использует команда. Когда члены команды обсуждают какой-то аспект модели, они буквально произносят фразы наподобие "Медсестры назначают пациентам вакцины от гриппа в стандартных дозах".

Таблица 1.4. Выбор оптимальной модели для бизнеса

Что лучше для бизнеса? Как спроектировать код, учитывая, что второе и третье утверждения одинаковы?	
Возможные точки зрения	**Итоговый код**
"Кого это волнует? Просто программируйте". М-да, ничего похожего на оптимальную модель	`patient.setShotType(ShotTypes.TYPE_FLU); patient.setDose(dose); patient.setNurse(nurse);`
"Мы делаем пациентам прививку от гриппа". Уже лучше, но не учтены некоторые важные концепции	`patient.giveFluShot();`
"Медсестры назначают пациентам вакцины от гриппа в стандартных дозах". Это похоже на то, к чему мы стремились, по крайней мере на данном этапе обучения	`Vaccine vaccine = vaccines.standardAdultFluDose();` `nurse.administerFluVaccine(patient, vaccine);`

Вокруг ЕДИНОГО ЯЗЫКА обязательно будут вестись споры и борьба, отображающая разные точки зрения экспертов и членов команды. Все это естественная часть развития наилучшего ЕДИНОГО ЯЗЫКА, как бы долго ни шел этот процесс. Это развитие происходит в результате открытых обсуждений, просмотра существующей документации, анализа "тайных знаний", которые в итоге проявятся, а также подключения стандартов, словарей и тезаурусов. На каком-то этапе вы обнаружите, что какие-то слова и фразы выпадают из бизнес-контекста, а какие-то соответствуют ему намного лучше.

Итак, как же овладеть этим исключительно важным ЕДИНЫМ ЯЗЫКОМ? Рассмотрим несколько способов, которые приводят от экспериментирования к точному знанию.

- Нарисуйте диаграммы физической и концептуальной предметной областей и нанесите на них метки, обозначающие имена и действия. Эти диаграммы носят неформальный характер, но могут содержать отдельные аспекты формального моделирования программного обеспечения. Даже если ваша команда применяет формальное моделирование с помощью языка Unified Modeling Language (UML), следует избегать формальностей, которые заведут вас в болото бесплодных дискуссий и подавят творческий потенциал ЕДИНОГО ЯЗЫКА.
- Создайте глоссарий с простыми определениями. Перечислите альтернативные термины, включая как перспективные, так и бесперспективные, сопровождая их комментариями, объясняющими причины такой оценки. Включив определения, вы не сможете не разработать устойчивые словосочетания ЕДИНОГО ЯЗЫКА, потому что ваша цель — создать ЕДИНЫЙ ЯЗЫК предметной области.
- Если вам не нравится идея разработать глоссарий, попробуйте создать документацию, содержащую неформальные диаграммы или важные понятия, связанные с программным обеспечением. И в этом случае ваша цель остается неизменной — выявить дополнительные термины и фразы ЕДИНОГО ЯЗЫКА.
- Поскольку лишь некоторые члены команды в состоянии сразу освоить глоссарий и другие письменные документы, обсудите итоговые фразы с остальными членами команды еще раз. Вряд ли вы достигнете полного согласия относительно всего словаря, поэтому следует быть гибкими и готовыми к интенсивному редактированию.

Мы перечислили идеальные первые шаги по выработке ЕДИНОГО ЯЗЫКА, соответствующего вашей предметной области. Однако его нельзя считать моделью, которую вы разрабатываете. Это только зародыш ЕДИНОГО ЯЗЫКА, который вскоре будет выражен в исходном коде системы. В качестве языка программирования можно выбрать Java, C#, Scala или любой другой язык. Эти диаграммы и документы также не учитывают, что ЕДИНЫЙ ЯЗЫК со временем будет расширяться и видоизменяться. Скорее всего, артефакты, которые вначале способствовали разработке полезного ЕДИНОГО ЯЗЫКА, хорошо соответствующего конкретной предметной области, со временем устареют. *Именно поэтому в речь команды и модель, выраженную в коде, в итоге войдут лишь самые проверенные и устойчивые элементы* ЕДИНОГО ЯЗЫКА.

Поскольку речь команды и код состоят из устойчивых выражений ЕДИНОГО ЯЗЫКА, будьте готовы отказаться от диаграмм, глоссария и другой документации,

которую будет трудно обновлять и согласовывать с текущим вариантом ЕДИНОГО ЯЗЫКА и исходным кодом, развивающимися очень быстро. Это не требование подхода DDD, а прагматичная рекомендация, поскольку поддерживать синхронизацию всей документации с системой непрактично.

Итак, как мы можем изменить метод `saveCustomer()` с учетом вышесказанного? Что если мы потребуем, чтобы класс `Customer` отражал все возможные цели бизнеса, которые он может поддерживать?

```
public interface Customer {
    public void changePersonalName(String firstName, String lastName);
    public void postalAddress(PostalAddress postalAddress);
    public void relocateTo(PostalAddress changedPostalAddress);
    public void changeHomeTelephone(Telephone telephone);
    public void disconnectHomeTelephone();
    public void changeMobileTelephone(Telephone telephone);
    public void disconnectMobileTelephone();
    public void primaryEmailAddress(EmailAddress emailAddress);
    public void secondaryEmailAddress(EmailAddress emailAddress);

}
```

Можно возразить, что это не наилучшая модель для объекта класса `Customer`, но при реализации подхода DDD на первом плане находятся вопросы проектирования. Команда может свободно спорить о том, какая модель является наилучшей, но прийти к согласию она может только после выработки ЕДИНОГО ЯЗЫКА. Предыдущий вариант интерфейса явно отражает разные цели бизнеса, которые должен поддерживать класс `Customer`, хотя с помощью ЕДИНОГО ЯЗЫКА его можно уточнять снова и снова.

Следует хорошо понимать, что ПРИКЛАДНУЮ СЛУЖБУ также необходимо перестраивать, чтобы отобразить ее явное соответствие насущным целям бизнеса. Каждый метод ПРИКЛАДНОЙ СЛУЖБЫ можно модифицировать для работы с отдельным пользовательским сценарием или пользовательской историей.

```
@Transactional

public void changeCustomerPersonalName(
    String customerId,
    String customerFirstName,
    String customerLastName) {

    Customer customer = customerRepository.customerOfId(customerId);

    if (customer == null) {
        throw new IllegalStateException("Customer does not exist.");
    }

    customer.changePersonalName(customerFirstName, customerLastName);
}
```

Этот вариант отличается от исходного, потому что в этом коде для выполнения разных пользовательских сценариев или пользовательских историй был использован отдельный метод. В новом примере мы ограничились тем, что отдельный метод ПРИКЛАДНОЙ СЛУЖБЫ изменяет только имя объекта класса `Customer`. Таким образом, используя подход DDD, нам достаточно лишь соответствующим образом уточнить ПРИКЛАДНУЮ СЛУЖБУ. Отсюда следует, что пользовательский интерфейс, скорее всего, будет отображать более узкую цель пользователя, как и прежде. Однако сейчас этот конкретный метод ПРИКЛАДНОЙ СЛУЖБЫ не требует от клиента передавать десять нулей, следующих после его имени и фамилии.

Удовлетворяет ли вас этот новый вариант? Вы можете легко читать и понимать код. Вы также можете тестировать его и убедиться, что он делает именно то, чего вы от него хотели, и не делает ничего, чего не должен делать.

Таким образом, ЕДИНЫЙ ЯЗЫК — это командный шаблон, который используется для описания концепций и терминов специфической смысловой области в виде модели программного обеспечения. Эта модель включает в себя существительные, прилагательные, глаголы и выражения, которые сформулированы сплоченной командой и используются ее членами. Программное обеспечение и тесты для верификации соответствия модели принципам предметной области выражаются с помощью ЕДИНОГО ЯЗЫКА, на котором говорят члены команды.

Единый, но не универсальный

Настало время прояснить вопросы, связанные с выразительностью ЕДИНОГО ЯЗЫКА. При этом следует тщательно учитывать несколько основных концепций.

- *Единый* значит "вездесущий" или "повсеместный", т.е. язык, *на котором говорят члены команды и выражается отдельная модель предметной области*, которую разрабатывает команда.
- *Единый* — это не значит "язык промышленности, компании или всего мира".
- В каждом ОГРАНИЧЕННОМ КОНТЕКСТЕ существует только один ЕДИНЫЙ ЯЗЫК.
- ОГРАНИЧЕННЫЕ КОНТЕКСТЫ являются относительно небольшими, меньше, чем может показаться на первый взгляд. ОГРАНИЧЕННЫЙ КОНТЕКСТ достаточно велик только для ЕДИНОГО ЯЗЫКА изолированной предметной области, но не больше.
- Язык является единым только в рамках команды, работающей над проектом в изолированном ОГРАНИЧЕННОМ КОНТЕКСТЕ.

- В отдельном проекте, в котором разрабатывается ОГРАНИЧЕННЫЙ КОНТЕКСТ, всегда существует один или несколько изолированных ОГРАНИЧЕННЫХ КОНТЕКСТОВ, которые объединяются с помощью **КАРТ КОНТЕКСТОВ (CONTEXT MAPS) (3)**. Каждый из этих ОГРАНИЧЕННЫХ КОНТЕКСТОВ имеет свой ЕДИНЫЙ ЯЗЫК, даже если некоторые термины совпадают.
- Попытка применить отдельный ЕДИНЫЙ ЯЗЫК в рамках всего предприятия или, что еще хуже, среди нескольких предприятий, закончится провалом.

Начиная новый проект, в котором правильно применяется подход DDD, следует идентифицировать изолированный ОГРАНИЧЕННЫЙ КОНТЕКСТ, который необходимо разработать. Он позволит провести четкую границу вокруг вашей модели предметной области. Обсуждайте, проводите исследования, формулируйте понятия, разрабатывайте проект и говорите на ЕДИНОМ ЯЗЫКЕ изолированной модели предметной области в рамках четкого ОГРАНИЧЕННОГО КОНТЕКСТА. Отказывайтесь от всех концепций, не соответствующих ЕДИНОМУ ЯЗЫКУ вашего изолированного ОГРАНИЧЕННОГО КОНТЕКСТА.

Бизнес-ценность DDD

Если ваш опыт сравним с моим, то вы знаете, что сейчас разработчики программного обеспечения уже не гонятся за модными методами и технологиями. Мы должны обосновывать все наши действия. Я думаю, что так было не всегда, но хорошо, что сейчас это так. Я считаю, что наилучшим обоснованием использования любой технологии являются ценности, которые она приносит бизнесу. Если мы можем предложить реальную, материальную ценность для бизнеса, то зачем бизнесменам отказываться от наших рекомендаций?

Если мы способны доказать, что наши рекомендации могут принести больше пользы, чем любые другие, то бизнес-ценность технологии повышается еще больше.

Является ли бизнес-ценность самой важной?

Конечно, стоило поместить этот подраздел в начало книги. Но я решил этого не делать. Возможно, точнее было бы назвать его "Как преподнести DDD вашему боссу". Пока вы не будете уверены, что существует реальная возможность действительно реализовать подход DDD в вашей компании, эта книга останется лишь теоретической. Я бы не хотел, чтобы вы читали эту книгу как теоретическое упражнение. Читайте ее как практическое руководство для реальных проектов. В этом случае вы сможете извлечь из нее реальную пользу. Так что продолжайте чтение!

Рассмотрим очень реалистичную бизнес-ценность, которую можно получить благодаря DDD. Объясните это вашему руководству, экспертам в предметной области и членам команды. Здесь ценность и выгоды перечисляются вместе, так что я уточню их. Начнем с выгод, которые носят менее технический характер.

1. Организация получит полезную модель своей предметной области.
2. Будут разработаны точное определение и описание бизнеса.
3. В разработке программного обеспечения будут принимать участие эксперты в предметной области.
4. Пользователи системы повысят свою квалификацию.
5. Модели будут иметь четкие границы.
6. Будет улучшена архитектура предприятия.
7. Будут применяться гибкие, итеративные и непрерывные методы моделирования.
8. Будут развернуты новые стратегические и тактические инструменты.

1. Организация получает полезную модель своей предметной области

Акцент на DDD означает направление усилий на то, что приносит бизнесу наибольшую. пользу. Мы не увлекаемся излишним моделированием. В центре нашего внимания находится СМЫСЛОВОЕ ЯДРО. Другие модели предназначены для поддержки СМЫСЛОВОГО ЯДРА и тоже являются важными. Однако приоритет имеют не вспомогательные модели, а СМЫСЛОВОЕ ЯДРО.

Когда вы сосредоточитесь на том, что отличает ваш бизнес от всех других, ваша задача станет хорошо понятной и вы определите параметры, которые необходимо соблюдать. В результате выполнения проекта вы получите именно то, что обеспечит вашей компании конкурентное преимущество.

2. Вырабатываются точное определение и описание бизнеса

В результате применения DDD будет достигнуто более полное понимание бизнеса и его цели. Я слышал от многих, что ЕДИНЫЙ ЯЗЫК, разработанный для СМЫСЛОВОГО ЯДРА бизнеса, нашел свое применение в маркетинговых материалах. Разумеется, он должен использоваться для визуализации документов и описания общей концепции деятельности компании.

Благодаря тому, что модель со временем уточняется, возникает более глубокое понимание принципов работы компании, которые могут стать инструментом анализа. В ходе совместной работы вам удастся выпытать подробности у экспертов в предметной области. Эти подробности помогут в оценке текущего и будущего направлений развития бизнеса, как стратегического, так и тактического.

3. В разработке программного обеспечения принимают участие эксперты в предметной области

Более глубокое понимание смысла своего бизнеса — это несомненная бизнес-ценность. Эксперты в предметной области не всегда достигают согласия относительно понятий и терминологии. Иногда несогласованность терминов возникает из-за поступления на работу новых специалистов, обладающих другим опытом. Иногда она является результатом разных подходов, которые применяются разными экспертами, работающими в одной и той же организации. Объединив усилия в рамках подхода DDD, эксперты в предметной области могут достичь консенсуса. Это усилит проект и организацию в целом.

В настоящее время разработчики используют ЕДИНЫЙ ЯЗЫК для равноправного общения с экспертами в предметной области. Передача знаний от экспертов в предметной области к другим членам команды приносит большую пользу. С учетом того, что разработчики рано или поздно уйдут либо в новое СМЫСЛОВОЕ ЯДРО, либо в другую организацию, подход DDD облегчает обучение и "передачу эстафеты". Шансы появления "тайного знания" о модели, которым обладают лишь несколько избранных специалистов, снижаются. Эксперты, оставшиеся разработчики и вновь поступившие работники будут по-прежнему обладать общим знанием, доступным каждому сотруднику организации. Это преимущество возникает благодаря тому, что цель, выраженная с помощью ЕДИНОГО ЯЗЫКА предметной области, носит постоянный характер.

4. Пользователи системы повышают свою квалификацию

Часто удается повысить квалификацию конечных пользователей, чтобы она точнее соответствовала модели предметной области. Поскольку предметная ориентированность подхода является его формальным признаком, он влияет на использование программного обеспечения людьми.

Когда программное обеспечение становится слишком сложным для понимания пользователями, их необходимо учить принимать множество решений. По существу, пользователи просто преобразовывают знания в данные, которые они вводят в формы. Затем данные записываются в хранилище. Если пользователи не совсем понимают, что им нужно, результат может оказаться неправильным. Часто пользователи вынуждены угадывать, что может делать программное обеспечение, и это снижает производительность их работы.

Если квалификация пользователя соответствует контурам базовой предметной модели, он может сделать правильные выводы. Программное обучение на самом деле учит пользователей, снижая затраты компании на обучение своих сотрудников. Повышение производительности и снижение затрат на обучение — это бизнес-ценность.

Теперь перейдем к выгодам бизнеса, которые носят более технический характер

5. Модели имеют четкие границы

Технологический персонал обычно приходит в уныние от того, что необходимость учитывать бизнес-ценности заставляет их решать более сложные программистские и алгоритмические задачи. Точность выбора направления позволяет сосредоточиться на достижении максимального результата. Для решения этой задачи необходимо понимать ОГРАНИЧЕННЫЙ КОНТЕКСТ проекта.

6. Улучшается архитектура предприятия

Благодаря тщательно проработанному ОГРАНИЧЕННОМУ КОНТЕКСТУ все команды предприятия начинают хорошо понимать, где и почему им следует выполнять интеграцию. Границы и отношения между ними определены очень точно. Команды, между моделями которых существует зависимость, развертывают КАРТЫ КОНТЕКСТОВ, чтобы установить формальные отношения и способы интеграции. Это действительно может привести к очень глубокому пониманию архитектуры всего предприятия.

7. Применяются гибкие, итеративные и непрерывные методы моделирования

Слово *проект* может вызывать негативные эмоции у руководства. Однако DDD — это не тяжеловесный, формальный процесс проектирования и программирования. Подход DDD не сводится к рисованию диаграмм. Он подразумевает уточнение ментальной модели экспертов в предметной области и ее преобразование в модель, полезную для бизнеса. Он не предназначен для создания модели, пытающейся имитировать реальность.

Команда разработчиков использует гибкий подход, имеющий итеративный и поступательный характер. В рамках подхода DDD команда может успешно применять любой процесс гибкой разработки, который для нее удобен. В результате создается модель, представляющая собой работающее программное обеспечение. Пока модель приносит бизнесу пользу, она непрерывно уточняется.

8. Развертываются новые стратегические и тактические инструменты

ОГРАНИЧЕННЫЙ КОНТЕКСТ позволяет команде очертить границы, в рамках которых вырабатывается решение конкретной задачи предметной области. В рамках отдельного ОГРАНИЧЕННОГО КОНТЕКСТА команда формулирует ЕДИНЫЙ ЯЗЫК. На нем говорят члены команды и формулируется модель программного обеспечения. Разношерстные команды, отвечающие за свои ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, используют КАРТУ КОНТЕКСТОВ для стратегического

разграничения ОГРАНИЧЕННЫХ КОНТЕКСТОВ и выяснения потребности в интеграции. Внутри границ отдельной модели команда может развертывать любое количество полезных тактических инструментов для тактического моделирования: **АГРЕГАТЫ (AGGREGATES) (10)**, **СУЩНОСТИ (ENTITIES) (5)**, **ОБЪЕКТЫ-ЗНАЧЕНИЯ (OBJECT-VALUES) (6)**, **СЛУЖБЫ (SERVICES) (7)**, **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS) (8)** и др.

Проблемы применения DDD

Реализуя подход DDD, вы сталкиваетесь с проблемами. Кто-нибудь когда-нибудь с ними уже сталкивался. Каковы типичные проблемы и как их решить с помощью DDD? Рассмотрим одну из распространенных проблем.

- Для создания ЕДИНОГО ЯЗЫКА нужны время и усилия.
- В проекте изначально и постоянно должны принимать участие эксперты в предметной области.
- Для решения проблем предметной области разработчики должны изменить способ мышления.

Одна из наиболее сложных проблем, связанных с использованием принципов DDD, состоит в том, что для размышлений о предметной области, выработки концепций и терминологии, а также для разговоров с экспертами в предметной области с целью выработки, освоения и уточнения ЕДИНОГО ЯЗЫКА, а не закодированного технического сленга, требуются время и усилия. Если вы хотите полноценно применять принципы DDD с наибольшей пользой для бизнеса, то должны больше думать и прилагать больше усилий, а для этого требуется больше времени. Так оно и есть, точка.

Иногда необходимое привлечение экспертов в предметной области оказывается затруднительным. Не важно, насколько это трудно, вы должны это сделать. Если вы не привлечете к реализации проекта хотя бы одного реального эксперта, то не получите глубоких знаний о предметной области. Если же вы включили в проект экспертов в предметной области, то ответственность за успех ложится на плечи разработчиков. Именно разработчики должны разговаривать с настоящими экспертами и внимательно их слушать, преобразовывая их разговорный язык в программное обеспечение, отражающее их ментальную модель предметной области.

Если предметная область, в которой вы работаете, действительно имеет значение для вашего бизнеса, то знание о ней надо получить именно от экспертов в предметной области. Я участвовал в проектах, в которых сложно было встретиться с экспертами в предметной области. Они часто ездили в командировки, и могли проходить недели между нашими одночасовыми совещаниями с их участием. В малом бизнесе таким экспертом может быть директор или один из вице-президентов, и они обременены обязанностями, которые им кажутся более важными.

Ковбойская логика

AJ: Если ты не в состоянии поймать на аркан большого бычка, то все время будешь голодным.

Для того чтобы заполучить эксперта в предметной области, иногда надо проявить сообразительность...

Как привлечь в проект экспертов в предметной области

Кофе. Или, выражаясь ЕДИНЫМ ЯЗЫКОМ,

"Привет, Салли, я купил тебе большой, наполовину обезжиренный, наполовину однопроцентный, экстрагорячий, две части с кофеином, две части без кофеина латте со взбитыми сливками. Найдется ли у тебя несколько минут для разговора о ...?"

Учите ЕДИНЫЙ ЯЗЫК менеджеров высшего уровня: "... прибыль ... доход ... конкурентное преимущество ... рыночное доминирование". Серьезно.

Билеты на хоккей.

Для того чтобы правильно применить подход DDD, большинство разработчиков должны *изменить образ своего мышления*. Разработчики склонны к техническому мышлению. Нам легче находить технические решения. Думать так — совсем неплохо. Просто иногда лучше думать иначе. Если за многие годы вы привыкли рассматривать программное обеспечение только с технической точки зрения, то, возможно, сейчас настало подходящее время изменить свой образ мышления. Для этого лучше всего начать разработку ЕДИНОГО ЯЗЫКА.

Ковбойская логика

LB: У этого мужика слишком маленькие сапоги. Если он не найдет себе другую пару, у него заболят пальцы ног.

AJ: Ага. Как говорится, если ты не услышал, то должен хотя бы почувствовать.

Существует другой уровень мышления, которого требует подход DDD, исходя из своего названия. Моделируя предметную область с помощью программного обеспечения, мы должны хорошенько подумать о том, из каких объектов состоит модель и что они делают. Речь идет о *проектировании поведения объектов*. Мы хотим правильно назвать поведенческие функции объектов, чтобы отобразить их сущность в ЕДИНОМ ЯЗЫКЕ. Однако при этом следует внимательно рассматривать, что именно делают объекты в рамках того или другого конкретного поведения. На этом уровне работа выходит за рамки простого создания атрибутов класса и предоставления клиентам модели открытых методов получателей и установщиков.

Рассмотрим более интересную предметную область, не такую рудиментарную, как предыдущая. Я специально повторю свои рекомендации, чтобы развить свои идеи.

Что произойдет, если мы просто включим в нашу модель методы доступа к данным? Если мы откроем объектам модели только методы доступа, мы получим модель, напоминающую модель данных. Рассмотрим два примера и попробуем понять, какой из них требует более тщательного проектирования, а какой приносит больше выгоды своим клиентам. Требования выражены в виде Scrum-модели, в которой мы должны выбрать список заданий для спринта.[1] Вероятно, вы делаете это постоянно, так что предметная область вам хорошо знакома.

В первом примере, как это сейчас принято, используются методы доступа к атрибутам.

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...
    public void setSprintId(SprintId sprintId) {
```

[1] Спринт — итерация процесса разработки на основе принципов Scrum, на которой расширяются функциональные возможности программного обеспечения; задача — функциональная возможность, выбранная заказчиком из списка заданий проекта для реализации в данном спринте; список заданий проекта — общий список требований к функциональным возможностям программного обеспечения. — *Примеч. ред.*

```
    this.sprintId = sprintId;
    }

    public void setStatus(BacklogItemStatusType status) {
        this.status = status;
    }
 ...

}
```

Запишем клиент этой модели.

```
// клиент выбирает список заданий для спринта,
// устанавливая его атрибуты sprintId и status

backlogItem.setSprintId(sprintId);
backlogItem.setStatus(BacklogItemStatusType.COMMITTED);
```

Во втором примере используется поведение объекта предметной области, которое выражает ЕДИНЫЙ ЯЗЫК предметной области.

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...

    public void commitTo(Sprint aSprint) {
         if (!this.isScheduledForRelease()) {
              throw new IllegalStateException(
                   "Must be scheduled for release to commit to sprint.");
    }

    if (this.isCommittedToSprint()) {
        if (!aSprint.sprintId().equals(this.sprintId())) {
             this.uncommitFromSprint();
       }
    }

    this.elevateStatusWith(BacklogItemStatus.COMMITTED);

    this.setSprintId(aSprint.sprintId());

    DomainEventPublisher
        .instance()
        .publish(new BacklogItemCommitted(
                this.tenant(),
                this.backlogItemId(),
                this.sprintId()));
    }
 ...
}
```

Клиент этой явной модели работает более безопасно.

```
// клиент выбирает список заданий для спринта,
// используя предметно-ориентированное поведение

backlogItem.commitTo(sprint);
```

В первом примере используется очень дата-центричный подход. Клиент обязан знать, как правильно выбрать список заданий для спринта. Модель, которая в данном случае не является предметной, не может ему помочь. Что произойдет, если клиент по ошибке изменит только атрибут `sprintId`, но не `status`, или наоборот? А что если в будущем понадобится задать другой атрибут? Для ответа на эти вопросы необходимо проанализировать код клиента и проверить, правильно ли он отображает значения данных в соответствующие атрибуты класса `BacklogItem`.

Этот подход раскрывает форму объекта класса `BacklogItem` и концентрирует внимание на его атрибутах, а не поведении. Даже если возразить, что методы `setSprintId()` и `setStatus()` описывают поведение, ничего не изменится, — дело в том, что такое "поведение" не имеет ценности для предметной области. Такое "поведение" не демонстрирует явным образом предназначение сценариев, которые должно моделировать программное обеспечение предметной области при выборе списка заданий для спринта. Если разработчик клиента попытается мысленно выбрать среди атрибутов класса `BacklogItem` атрибуты, необходимые для выбора списка заданий для спринта, это вызовет у него когнитивный диссонанс. Во многом это объясняется тем, что модель ориентируется на данные.

Рассмотрим второй пример. Вместо раскрытия клиентам атрибутов данных он раскрывает поведение, которое четко и однозначно означает, что клиент должен выбрать список заданий для спринта. Эксперты в этой предметной области обсуждают следующее требование к модели.

> Разрешить выбор каждого списка заданий для спринта. Список может быть выбран, только если он уже запланирован для выпуска. Если он уже выбран для другого спринта, то его предыдущий выбор должен быть отменен. После завершения выбора об этом следует уведомить все стороны.

Таким образом, метод во втором примере описывает ЕДИНЫЙ ЯЗЫК модели в контексте, т.е. в ОГРАНИЧЕННОМ КОНТЕКСТЕ, в котором изолирован тип `BacklogItem`. Анализируя этот сценарий, мы обнаруживаем, что первый сценарий не полон и содержит ошибки.

Во второй реализации клиенты не обязаны знать, что требуется для осуществления выбора, независимо от его сложности. Реализация этого метода именно настолько логична, насколько это необходимо. Во вторую реализацию легко

добавить защиту, чтобы предотвратить выбор списка заданий, еще не предназначенного для выпуска. Правда, вы можете включить защиту и в первую реализацию, но метод-установщик в этом случае должен понимать полный контекст состояния объектов, а не просто требования к атрибутам `sprintId` и `status`.

Кроме того, существует еще одно тонкое различие. Отметим, что если список заданий уже был выбран для другого спринта, в текущем спринте его выбор сначала будет отменен. Это важная деталь, потому что, если выбор списка заданий в спринте не отменен, для клиентов должно быть опубликовано СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ.

> Разрешите отмену выбора в спринте для каждого списка заданий. Когда выбор списка будет отменен, уведомьте об этом заинтересованные стороны.

Публикацию уведомления об отмене выбора можно осуществить без реализации метода предметной области `uncommitFrom()`. Метод `commitTo()` даже не обязан знать содержания уведомления. Достаточно, чтобы он знал, что необходимо отменить выбор для любого текущего спринта до осуществления выбора списка заданий для нового спринта. Кроме того, на заключительном этапе своей работы метод предметной области `commitTo()` уведомляет заинтересованные стороны с помощью СОБЫТИЯ. Если не поместить эту функцию в класс `BacklogItem`, то мы будем вынуждены публиковать СОБЫТИЯ с помощью клиентов. Очевидно, что это приведет к утечке логики предметной области из модели. Это плохо.

Ясно, что во втором примере по сравнению с первым необходимо тщательнее проработать метод `BacklogItem`. Хотя эта проработка не очень сложная, она дает намного больше преимуществ. Чем лучше мы научимся пользоваться этим методом проектирования, тем легче будет его применить. В результате от команды потребуется больше умственных усилий, больший объем работы, более тесное сотрудничество и более высокая организованность, но не настолько, чтобы подход DDD стал неприемлемым. Новые идеи стоят затраченных усилий.

Заметки на доске

- Анализируя конкретную предметную область, в которой вы работаете в данный момент, выделите общие термины и действия в рамках модели.
- Запишите эти термины на доске.
- Затем запишите фразы, которыми должна пользоваться ваша команда во время обсуждения проекта.
- Обсудите эти термины и фразы с реальными экспертами в предметной области и попросите уточнить их (не забудьте о кофе).

Обоснование моделирования предметной области

Тактическое моделирование обычно сложнее *стратегического*. Следовательно, если вы собираетесь разрабатывать модель предметной области с помощью тактических шаблонов DDD (АГРЕГАТОВ, СЛУЖБ, ОБЪЕКТОВ-ЗНАЧЕНИЙ, СОБЫТИЙ и т.п.), то будете вынуждены тратить больше умственных усилий и средств. Как же обосновать тактическое моделирование предметной области, если оно связано с бо́льшими затратами? Какими критериями следует пользоваться, чтобы объяснить необходимость дополнительных инвестиций для правильной и полноценной реализации подхода DDD?

Представьте себе, что вы ведете экспедицию по неизвестной территории. Для этого надо знать окружающий ландшафт и границы. Ваша группа должна изучить карты, возможно, даже самостоятельно их нарисовать и определить стратегическое направление. Вы должны выяснить особенности территории и понять, как их использовать. Независимо от того, насколько тщательным будет ваше планирование, некоторые аспекты вашего путешествия все равно будут действительно сложными.

Если ваша стратегия приведет вас к выводу, что вы должны залезть на высокую скалу, то вы должны взять соответствующие тактические средства и выполнить маневры, необходимые для подъема. Находясь внизу и смотря наверх, вы видите конкретные сложности и возможные опасности. Конечно, вы не можете видеть все детали, пока не подойдете к скале вплотную. Вы можете пойти по скользкой змеиной тропе, а можете забивать клинья в камни. Для того чтобы воспользоваться этими средствами защиты, вы возьмете с собой карабины. Вы можете сделать маршрут максимально прямым, но на самом деле будете перемещаться от одной выбранной вами точки к другой. Иногда вам даже придется возвращаться и обходить опасные места в зависимости от конкретной обстановки. Многие люди считают скалолазание опасным видом спорта, но те, кто на самом деле поднимался на скалы, скажут вам, что это безопаснее, чем водить машину или летать на самолете. Конечно, для того чтобы это было правдой, скалолазы овладевают инструментами и методами ориентации на скале.

Разрабатывая конкретную **ПРЕДМЕТНУЮ ПОДОБЛАСТЬ (SUBDOMAIN) (2)**, в которой необходимо выполнить такой сложный и даже рискованный подъем, вы будете пользоваться тактическими шаблонами DDD так, как скалолазы пользуются веревками. Бизнес-инициатива, соответствующая критериям СМЫСЛОВОГО ЯДРА, не означает немедленного отказа от использования тактических шаблонов. СМЫСЛОВОЕ ЯДРО — это неизвестная и сложная область. Используя правильную тактику, команда лучше всего защищена от опасного падения.

Приведем несколько практических рекомендаций. Я начну с первого сверху уровня и буду постепенно их уточнять.

- Если ОГРАНИЧЕННЫЙ КОНТЕКСТ разрабатывается как СМЫСЛОВОЕ ЯДРО, значит, он имеет жизненно важное значение для успеха бизнеса. Базовая модель не очень понятна и требует многочисленных экспериментов и рефакторинга. Вероятно, в нее необходимо вносить постоянные уточнения на протяжении долгого времени. Модель может выходить за пределы ваших знаний. Тем не менее, если ОГРАНИЧЕННЫЙ КОНТЕКСТ носит сложный, инновационный характер и требует непрерывных изменений, настоятельно рекомендуем рассмотреть возможность использования тактических шаблонов в качестве инвестиций в будущее вашего бизнеса. Это значит, что ваше СМЫСЛОВОЕ ЯДРО заслуживает лучших разработчиков высокой квалификации.
- Предметная область, которая может стать **НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ (GENERIC SUBDOMAIN) (2)** или СЛУЖЕБНОЙ ПОДОБЛАСТЬЮ (SUPPORTING SUBDOMAIN) для своих клиентов, на самом деле может быть СМЫСЛОВЫМ ЯДРОМ для вашего бизнеса. Не всегда следует рассматривать предметную область с точки зрения конечного пользователя. Если вы разрабатываете ОГРАНИЧЕННЫЙ КОНТЕКСТ по заданию вашего руководства, это ваше СМЫСЛОВОЕ ЯДРО не зависит от точки зрения ваших внешних клиентов. Настоятельно рекомендуем тактические шаблоны.
- Если вы разрабатываете СЛУЖЕБНУЮ ПОДОБЛАСТЬ, которая по разным причинам не может рассматриваться как сторонняя НЕСПЕЦИАЛИЗИРОВАННАЯ ПОДОБЛАСТЬ, возможно, целесообразно использовать тактические шаблоны. В этом случае оцените уровень квалификации вашей команды и новизну модели. Она является инновационной, если добавляет конкретную бизнес-ценность, и отражает специальное знание, а не просто сложная с технической точки зрения. Если команда способна правильно применять тактические шаблоны и СЛУЖЕБНАЯ ПОДОБЛАСТЬ является инновационной и будет существовать долгое время, то целесообразно разрабатывать программное обеспечение с помощью тактического проектирования. Однако это не сделает вашу модель СМЫСЛОВЫМ ЯДРОМ, поскольку с точки зрения бизнеса она является СЛУЖЕБНОЙ.

Эти рекомендации могут оказаться несколько ограниченными, если в проекте участвует много разработчиков с огромным опытом моделирования предметной области. Если команда имеет громадный опыт, а инженеры считают, что тактические шаблоны — это лучший выбор, целесообразно ей поверить. Честные разработчики, независимо от накопленного опыта, всегда укажут, имеет ли смысл разрабатывать модель предметной области в конкретном случае.

Тип предметной области бизнеса сам по себе не является определяющим фактором при выборе метода проектирования. Для того чтобы помочь вам принять

правильное окончательное решение, ваша команда должна рассмотреть важные вопросы. Рассмотрим следующий короткий список более подробных параметров, влияющих на решение, которые более или менее согласованы с предыдущими рекомендациями.

- Доступны ли эксперты в предметной области и можно ли их привлечь к работе в команде?
- Возрастает ли со временем сложность предметной области бизнеса, даже если пока она относительно простая? Существует риск применить СЦЕНАРИЙ ТРАНЗАКЦИИ (TRANSACTION SCRIPT)[2] к сложным приложениям. Если сейчас вы используете СЦЕНАРИЙ ТРАНЗАКЦИИ, то следует ли оценить практичность последующего рефакторинга поведенческой модели предметной области, если КОНТЕКСТ станет более сложным?
- Облегчают ли тактические шаблоны DDD интеграцию с другими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, разработанными сторонними организациями или поставщиками? Насколько они практичны?
- Упрощается ли разработка на самом деле и уменьшается ли объем программирования при использовании СЦЕНАРИЯ ТРАНЗАКЦИИ? (Опыт показывает, что СЦЕНАРИЙ ТРАНЗАКЦИИ не уменьшает объем программирования. Вероятно, это объясняется тем, что сложность предметной области и инновационность модели на этапе планирования проекта недооцениваются.)
- Допускают ли установленный порядок разработки проекта и сроки его сдачи дополнительные затраты, связанные с тактическим проектированием?
- Может ли тактическое проектирование СМЫСЛОВОГО ЯДРА защитить систему от изменяющихся архитектурных влияний? СЦЕНАРИЙ ТРАНЗАКЦИИ может оставить систему уязвимой к таким изменениям. (Модели предметной области часто подвергаются более разрушительным архитектурным влияниям по сравнению с другими уровнями.)
- Получат ли клиенты/заказчики выгоду от более ясного, испытанного временем подхода к проектированию и разработке и можно ли в будущем заменить их приложения готовыми решениями? Иначе говоря, можем ли мы вообще разработать приложение/службу клиента?
- Насколько трудным является тактическое предметно-ориентированное проектирование приложения/службы по сравнению, например, со СЦЕНАРИЕМ ТРАНЗАКЦИИ? (Для ответа на этот вопрос ключевыми факторами являются уровень квалификации и доступность экспертов в предметной области.)

[2] Здесь я несколько обобщил термины. В этом списке СЦЕНАРИЙ ТРАНЗАКЦИИ охватывает несколько подходов, не использующих модель предметной области.

- Можем ли мы добросовестно выбрать другой подход, если инструментарий команды уже укомплектован средствами DDD? (Некоторые инструменты применяют моделирование постоянного хранения, например используют объектно-реляционное отображение, полную сериализацию и хранение АГРЕГАТОВ, ХРАНИЛИЩЕ СОБЫТИЙ (EVENT STORE) или каркас, поддерживающий тактическое предметно-ориентированное проектирование. Существуют и другие инструменты.)

Этот список не учитывает особенности вашей предметной области, поэтому вы можете дополнить его дополнительными критериями. Просто следует понимать конкурирующие причины, чтобы выбрать самые лучшие и мощные методы. Кроме того, необходимо знать бизнес и технологии. Во главу угла следует ставить интересы бизнес-потребителя, а не разработчиков объектов и практиков. Подумайте хорошенько.

Подход DDD не сложный

Я ни в коем случае не хочу сказать, что правильное применение принципов DDD приводит к тяжелому процессу проектирования с многочисленными формальностями и тотальным документированием артефактов. К подходу DDD это не относится. Он хорошо сочетается с любыми методами гибкого проектирования, такими как Scrum, который команды любят применять. Он подразумевает быстрое уточнение реальной модели программного обеспечения с помощью тестирования. Если вам нужно разработать новый объект предметной области, например СУЩНОСТЬ или ОБЪЕКТ-ЗНАЧЕНИЕ, то разработка через тестирование (test-first approach) сведется к следующим этапам.

1. Разрабатывается тест, демонстрирующий, как клиент должен использовать новый объект предметной области.
2. Создается новый объект предметной области с кодом, достаточным для компиляции теста.
3. Выполняется рефакторинг, пока тест правильно отражает способ использования клиентом объекта предметной области, а объект предметной области имеет правильную сигнатуру поведенческих методов.
4. Реализуется поведение каждого объекта предметной области, пока они проходят тест, и выполняется рефакторинг объекта предметной области до тех пор, пока в нем существуют недопустимые дубликаты кода.
5. Код демонстрируется членам команды, включая экспертов предметной области, чтобы убедиться, что тест правильно использует объект предметной области в соответствии с текущим вариантом ЕДИНОГО ЯЗЫКА.

Может показаться, что описанная выше последовательность шагов ничем не отличается от разработки через тестирование, которое вы уже применяли. Они действительно мало отличаются, но дело в том, что в главном они совпадают. Цель этапа тестирования состоит не в том, чтобы попытаться доказать с абсолютной определенностью, что модель на данном этапе является безошибочной. Сначала мы сосредоточиваемся на том, как модель будет использоваться клиентами, а тесты влияют на проектирование модели. Хорошо, что это действительно соответствует гибкому подходу к разработке программного обеспечения. Подход DDD подразумевает облегченную разработку, без формальностей, сложностей и упреждающего проектирования. С этой точки зрения он не отличается от обычного метода гибкой разработки. Итак, даже если предыдущие этапы не дали вам четкого представления о методах гибкой разработки, я думаю, что они прояснили принципы DDD и показали, что они согласуются с гибким подходом.

Впоследствии вы также добавите тесты, верифицирующие корректность нового объекта предметной области со всех возможных (и практичных) точек зрения. Пока нас интересует правильность выражения концепции предметной области, воплощенной в новом объекте. Чтение демонстрационных тестов, ориентированных на клиентов, может выявить употребление ЕДИНОГО ЯЗЫКА. Эксперты в предметной области, которые не обязаны иметь техническое образование, должны читать коды тестов с помощью разработчиков, чтобы получить ясное представление о том, соответствует ли модель поставленным целям. Из этого следует, что тестовые данные должны быть реалистичными и типичными. В противном случае эксперты в предметной области не смогут правильно судить о реализации.

Эта методология разработки через тестирование повторяется до тех пор, пока вы не получите модель, решающую задачи, поставленные на текущей итерации. Указанные выше шаги представляют собой метод гибкой разработки и выражают первичную идею экстремального программирования. Использование методов гибкой разработки не исключает применения шаблонов и методов DDD. Вместе они работают довольно хорошо. Конечно, вы можете в полной мере применять принципы DDD без использования разработки через тестирование. Вы всегда можете разрабатывать тесты для проверки существующих объектов модели. Однако проектирование программного обеспечения с точки зрения клиента добавляет очень полезное свойство.

Правдоподобный вымысел

Когда я размышлял о том, как наилучшим образом изложить рекомендации по реализации современного подхода DDD, я хотел привести обоснование для каждого совета. Это значило ответить не только на вопрос "Как?", но и "Почему?" Оказалось, что для иллюстрации рекомендаций и демонстрации правильного использования

подхода DDD для решения основных проблем, возникающих перед разработчиками, удобно взглянуть на несколько проектов с точки зрения анализа сценариев.

Иногда проще посмотреть на проблемы, с которыми уже сталкивались другие проектные команды, и извлечь уроки из их неправильного употребления DDD, чем заниматься самоанализом. Конечно, как только вы распознаете дефекты работы других людей, вы будете в состоянии судить, идете ли вы в том же неправильном направлении или даже забрались в гущу того же болота. Зная правильный курс и свои координаты, вы можете внести точные корректировки, чтобы исправить ошибки и избежать их в будущем.

Вместо того чтобы описывать ряд реальных проектов, в которых я работал, — которые я все равно не могу обсуждать открыто, — я решил немного пофантазировать на основе реальных ситуаций, в которые я и другие разработчики уже попадали. Благодаря этому я мог создать идеальные условия, чтобы продемонстрировать причины, по которым определенный подход к реализации является наилучшим, по крайней мере для решения проблем DDD.

Таким образом, это не просто фантазия на тему анализа сценариев. Я описываю вымышленную компанию с реальным уставом, вымышленные команды, разрабатывающие и внедряющие реальное программное обеспечение в этой компании, а также реальные задачи, связанные с подходом DDD, и вытекающие из них проблемы с их реальными решениями. Я назвал такой прием "правдоподобным вымыслом". Я счел довольно эффективным писать в этом стиле и надеюсь, что вы извлечете из этого пользу.

Описывая любой набор примеров, мы должны его ограничить, чтобы сделать практичным. Иначе все попытки учить и учиться утонут в океане иллюстраций. В то же время примеры не могут быть чрезмерно упрощенными или нежизненными, иначе мы потеряем возможность извлечь из них жизненно важные уроки. Для того чтобы достичь баланса, я выбрал бизнес-ситуацию, связанную с разработкой проекта "с нуля".

Включаясь в проекты в разные моменты времени, мы увидим разные проблемы и успехи, которых достигли команды. СМЫСЛОВОЕ ЯДРО, которое лежит в центре внимания примеров, является достаточно сложным, чтобы исследовать подход DDD с разных точек зрения. Наши ОГРАНИЧЕННЫЕ КОНТЕКСТЫ используют друг друга, и это позволяет нам исследовать интеграцию с помощью подхода DDD. Однако эти три иллюстративные модели не могут продемонстрировать каждый аспект стратегического проектирования в среде "существующих производств", в которых существует множество унаследованных систем. Я не полностью избегаю этих менее интересных областей, как будто они не важны. Каждый раз, когда это будет желательным, мы будем отвлекаться от основных примеров и изучать области, где принципы DDD могут принести дополнительную выгоду.

Теперь позвольте мне представить вам компанию и немного рассказать о команде и проекте, над которым она работает.

SaaSOvation, ее продукция и применение DDD

Компания называется SaaSOvation. Как следует из названия, цель компании — разработка служебных программных продуктов (Software As a Service — SaaS). Продукты компании SaaS развернуты на ее мощностях. Они оцениваются и используются организациями-подписчиками. Бизнес-план компании включает два запланированных продукта, которые должны быть выпущены один за другим.

Основной продукт называется CollabOvation. Он представляет собой набор программ для обеспечения корпоративного сотрудничества, обладающий возможностями ведущих социальных сетей. К этим возможностям относятся форумы, общие календари, блоги, мгновенные сообщения, wiki-система, доски сообщений, система управления документооборотом, объявления и предупреждения, отслеживание рабочего процесса и ленты новостей RSS. Все инструменты для обеспечения сотрудничества фокусируются на бизнес-потребностях корпораций, помогая им резко повысить производительность небольших проектов, больших программ и подразделений. Деловое сотрудничество важно для создания и укрепления взаимовыгодной атмосферы в сегодняшней изменяющейся, иногда неустойчивой, но все же быстро развивающейся экономике. Все, что может помочь повысить производительность, позволяет передавать знания, способствует совместному использованию идей, коллективному управлению творческим процессом и достижению результатов, идет во благо общего корпоративного успеха. Система CollabOvation делает выгодное предложение клиентам, а ее разработка доставит удовольствие проектировщикам.

Вторым продуктом, названным ProjectOvation, является СМЫСЛОВОЕ ЯДРО, являющееся предметом главной заботы. Этот инструмент фокусируется на управлении быстрыми и гибкими проектами, используя методологию Scrum в качестве итеративной и инкрементной основы для управления проектами. ProjectOvation подчиняется традиционной модели управления проектами Scrum, дополненной продуктом, владельцем продукта, командой, задачами, списками заданий, запланированными выпусками и спринтами. Оценка задач обеспечивается калькуляторами бизнес-ценности, использующими анализ эффективности затрат. Представьте себе всю мощь методологии Scrum, воплощенной в продукте ProjectOvation. Но компания SaaSOvation планирует получить еще большую отдачу от вложенных средств.

Продукты CollabOvation и ProjectOvation не должны развиваться изолированно друг от друга. Компания SaaSOvation и ее штат советников придумали новшество, касающееся разработки инструментов сотрудничества на принципах гибкой разработки программного обеспечения. В соответствии с этим новшеством функции продукта CollabOvation будут предлагаться как дополнения к продукту ProjectOvation. Без сомнения, предоставление инструментов для коллективного планирования проектов, обсуждения функциональных возможностей и сценариев, а также проведения дискуссий между командами и внутри команд окажется популярной

опцией. Компания SaaSOvation прогнозирует, что более 60 процентов подписчиков ProjectOvation добавят функции CollabOvation. Такой способ продажи дополнений часто приводит к полноценной продаже самого дополнительного продукта. Как только канал сбыта будет установлен и команды разработчиков программного обеспечения увидят выгоды от сотрудничества, обеспечиваемого их комплектом управления проектами, их энтузиазм будет влиять на решение купить полный комплект инструментов для обеспечения корпоративного сотрудничества. Благодаря такому вирусному способу продаж компания SaaSOvation прогнозирует, что впоследствии как минимум 35 процентов всех продаж продукта ProjectOvation приведут к покупке продукта CollabOvation. Они считают это осторожной оценкой, но и это чрезвычайно большой успех.

Сначала была укомплектована команда разработки продукта CollabOvation. В команде есть несколько закаленных ветеранов и намного больше разработчиков среднего уровня. На первых совещаниях в качестве основного подхода к проектированию и разработке команда выбрала предметно-ориентированное проектирование. Один из двух старших разработчиков уже использовал минимальный набор шаблонов DDD в предыдущем проекте, который реализовывал его бывший работодатель. По его описаниям, понятным опытным разработчикам, использующим практику DDD, это не было полноценным использованием DDD. То, что он делал, иногда называют облегченным подходом DDD.

Облегченный подход DDD (DDD-Lite) означает идентификацию и выбор подмножества тактических шаблонов DDD без должного внимания к выработке, формулировке и уточнению ЕДИНОГО ЯЗЫКА. Кроме того, в рамках этого подхода обычно пренебрегают использованием ОГРАНИЧЕННЫХ КОНТЕКСТОВ и КАРТАМИ КОНТЕКСТОВ. Этот подход носит технический характер и ориентируется на решение технических проблем. Он может обеспечить преимущество, но обычно не такое большое, как можно было бы ожидать от его сочетания со стратегическим моделированием. Компания SaaSOvation поверила в это. В ее случае это скоро приведет к проблемам, потому что команда не понимает ПРЕДМЕТНЫХ ПОДОБЛАСТЕЙ, а также мощи и надежности явных ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Ситуация могла быть еще хуже. Компания SaaSOvation смогла избежать главных опасностей, связанных с использованием облегченного подхода DDD-Lite, просто благодаря тому, что два ее основных продукта сами образуют естественный набор ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Это привело к тому, что модели CollabOvation и ProjectOvation формально разделены. Но это простая случайность. Это не значит, что команда поняла ОГРАНИЧЕННЫЙ КОНТЕКСТ. Из-за этого возникли ее первые проблемы. Итак, вы либо учитесь, либо делаете ошибки.

Хорошо, что мы можем извлечь выгоду из исследования неполного использования DDD компанией SaaSOvation. Команда в конечном счете научится на своих ошибках, лучше освоив методы стратегического проектирования. Вы также узнаете об исправлениях, сделанных командой CollabOvation, и о том, как команда ProjectOvation извлекла выгоду из ретроспективного анализа первичных условий в аналогичных проектах. Полное описание сценария изложено в главах,

посвященных **ПРЕДМЕТНЫМ ПОДОБЛАСТЯМ (2)** и **ОГРАНИЧЕННЫМ КОНТЕКСТАМ (2)**, а также **КАРТАМ КОНТЕКСТОВ (3)**.

Резюме

Итак, начало получилось обнадеживающим. Вы, вероятно, уже почувствовали, что вы и ваша команда действительно сможете успешно справиться с современным методом разработки программного обеспечения. Я согласен в вами.

Конечно, мы не собираемся ничего упрощать. Реализация подхода DDD требует действительно совместных усилий. Если бы это было просто, то любой мог бы написать большую программу, а мы знаем, что так просто не бывает. Поэтому подготовьтесь. Это будет стоить того, потому что ваш проект будет точно описывать, как работает ваше программное обеспечение.

Перечислим то, что мы изучили к данному моменту.

- Вы обнаружили то, что подход DDD может сделать для ваших проектов и ваших команд, чтобы помочь вам справиться со сложностью предметной области.
- Вы узнали, как оценить ваш проект, чтобы понять, заслуживает ли он инвестиций в подход DDD.
- Вы рассмотрели общие альтернативы DDD и поняли, почему использование альтернативных подходов часто приводит к проблемам.
- Вы усвоили основы DDD и готовы сделать первые шаги в вашем проекте.
- Вы узнали, как преподнести принципы DDD вашему руководству, экспертам в предметной области и техническим членам команды.
- Вы теперь вооружены знанием того, как успешно справиться с проблемами DDD.

Теперь поговорим о том, куда двигаться дальше. Следующие две главы посвящены теме первостепенной важности — стратегическому проектированию, а за ними следует глава об архитектуре программного обеспечения с использованием принципов DDD. Это действительно важный материал, которым необходимо овладеть, прежде чем перейти к изучению последующих глав о тактическом моделировании.

Глава 2

Предметные области, подобласти и ограниченные контексты

Ровно столько нот, сколько необходимо.
Моцарт в фильме "Амадей" (Amadeus)
(Orion Pictures, Warner Brothers, 1984)

Вы должны четко понимать три вещи.

- Что такое **ПРЕДМЕТНАЯ ОБЛАСТЬ**
- Что такое **ПРЕДМЕТНЫЕ ПОДОБЛАСТИ**
- Что такое **ОГРАНИЧЕННЫЕ КОНТЕКСТЫ**

Из того, что все эти понятия подробно обсуждаются во второй половине книги [Эванс], совсем не следует, что они имеют вторичное значение. Для того чтобы с успехом применять подход DDD, необходимо хорошо в них разбираться.

Назначение главы

- Получить общее представление о подходе DDD с помощью изучения шаблонов ПРЕДМЕТНАЯ ОБЛАСТЬ, ПРЕДМЕТНАЯ ПОДОБЛАСТЬ и ОГРАНИЧЕННЫЙ КОНТЕКСТ.
- Понять, почему стратегическое проектирование настолько важное и почему вредно проектировать, не используя стратегические шаблоны.
- Рассмотреть практическую ПРЕДМЕТНУЮ ОБЛАСТЬ с несколькими ПРЕДМЕТНЫМИ ПОДОБЛАСТЯМИ.

- Разобраться в ОГРАНИЧЕННЫХ КОНТЕКСТАХ как концептуально, так и в техническом отношении.
- Увидеть ключевые моменты сценария SaaSOvation, когда проектировщики распознают стратегические шаблоны.

Общая картина

В широком смысле ПРЕДМЕТНАЯ ОБЛАСТЬ (DOMAIN) — это то, что делают организация и среда, в которой она это делает. Бизнесмены выбирают рынок, продают товары и предоставляют услуги. Любая организация имеет собственную систему понятий и образ действий. Эта система понятий и методы работы организации образуют ПРЕДМЕТНУЮ ОБЛАСТЬ. Разрабатывая программное обеспечение для организации, вы работаете в ее ПРЕДМЕТНОЙ ОБЛАСТИ. Вам должно быть совершенно очевидно, что представляет собой ваша ПРЕДМЕТНАЯ ОБЛАСТЬ, ведь вы в ней работаете.

Следует помнить, что термин *предметная область* может быть немного перегружен. Он может означать всю предметную область бизнеса, его смысловое ядро или служебную подобласть. Я приложу все усилия, чтобы выделить каждое использование термина. Ссылаясь на отдельную область бизнеса, я буду называть ее **СМЫСЛОВЫМ ЯДРОМ**, **ПРЕДМЕТНОЙ ПОДОБЛАСТЬЮ** и т.п.

Поскольку термин *модель предметной области* включает слово *область*, можно было бы подумать, что мы должны создать единственную, связную, комплексную модель всех организаций данной предметной области, что-то вроде модели предприятия. Однако при использовании подхода DDD это не является нашей целью. Подход DDD делает акцент на совсем противоположном. Общая ПРЕДМЕТНАЯ ОБЛАСТЬ организации состоит из ПОДОБЛАСТЕЙ. Подход DDD позволяет разрабатывать модели в ОГРАНИЧЕННЫХ КОНТЕКСТАХ. Фактически при разработке МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ мы сосредоточиваемся только на одной определенной области всей предметной области. Любая попытка определить бизнес даже умеренно сложной организации в рамках единственной, всеобъемлющей модели будет чрезвычайно трудной и обречена на неудачу. Как станет ясно из этой главы, чтобы достичь успеха, мы должны четко выделить отдельные предметные области внутри общей предметной области.

Итак, если модель предметной области не должна охватывать все, что делает организация и как она это делает, то что именно она должна содержать?

Почти каждая ПРЕДМЕТНАЯ ОБЛАСТЬ программного обеспечения имеет несколько ПОДОБЛАСТЕЙ. На самом деле совершенно не имеет значения, является

ли организация огромной и чрезвычайно сложной или в ней работают несколько сотрудников. Кроме того, не имеет значения, какое программное обеспечение она использует. Существует несколько функций, обеспечивающих успех любого бизнеса, поэтому целесообразно рассмотреть каждую их этих функций отдельно.

ПРЕДМЕТНЫЕ ПОДОБЛАСТИ и ОГРАНИЧЕННЫЕ КОНТЕКСТЫ в действии

Вот довольно простой пример, позволяющий представить, как могут использоваться ПОДОБЛАСТИ. Представьте себе компанию розничной торговли, которая продает продукты в Интернете. Продукты, которые она продает, могут быть любыми, поэтому мы не будем слишком заострять на них внимание. Для того чтобы вести бизнес в этой ПРЕДМЕТНОЙ ОБЛАСТИ, компания должна предоставить покупателям каталог товаров, позволить разместить заказы, обеспечить взимание платы за проданные товары и доставку этих товаров покупателям. На первый взгляд кажется, что ПРЕДМЕТНАЯ ОБЛАСТЬ этого интернет-магазина состоит из четырех основных ПОДОБЛАСТЕЙ: *каталога товаров*, *заказов*, *выставления счетов* и *доставки*. ПОДОБЛАСТЬ *электронной торговли* показана в верхней части рис. 2.1.

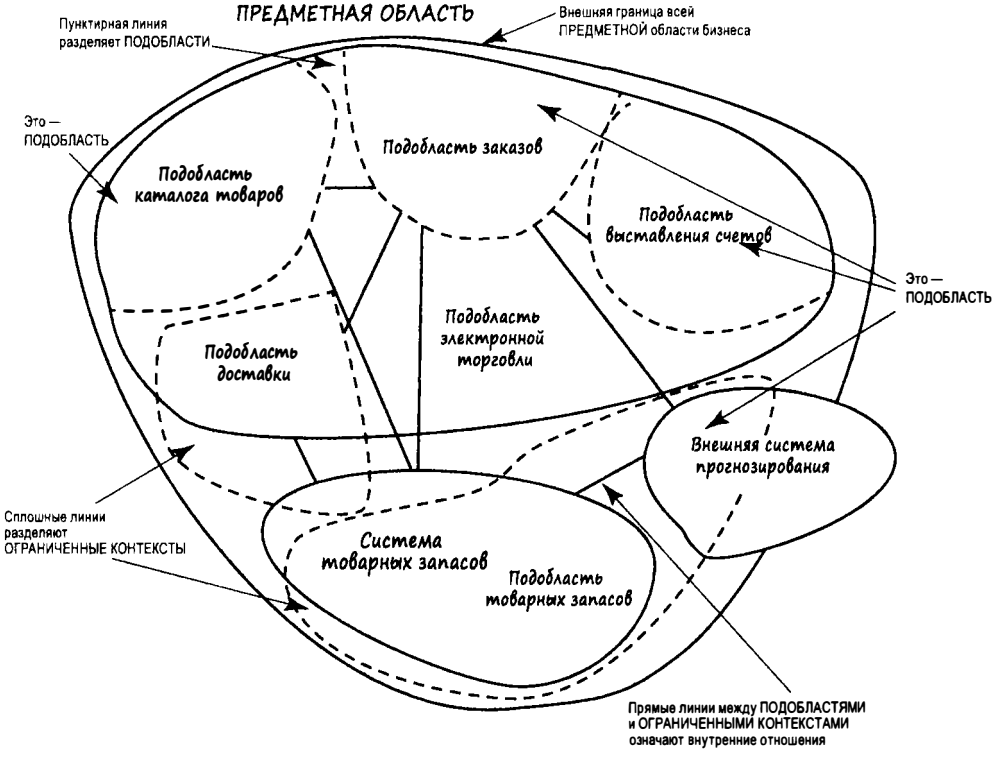


Рис. 2.1. ПРЕДМЕТНАЯ ОБЛАСТЬ с ПОДОБЛАСТЯМИ и ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ

Все это кажется довольно очевидным, и до некоторой степени это так и есть. Однако если мы введем всего одну дополнительную деталь, то сделаем наш пример более сложным. Представьте на мгновение, насколько трудно бывает работать с *товарными запасами*, дополнительной системой и ПОДОБЛАСТЬЮ, показанными на рис. 2.1. Мы возвратимся к проблеме повышенной сложности позднее, а пока рассмотрим физические подсистемы и логические подобласти на схеме.

Заметьте, что пока для того, чтобы понять ПРЕДМЕТНУЮ ОБЛАСТЬ этой розничной компании, мы рассматриваем всего три физические системы, из которых только две размещены внутри. Эти две внутренние системы можно представить в виде двух ОГРАНИЧЕННЫХ КОНТЕКСТОВ. К сожалению, в настоящее время большинство систем разрабатывается без использования подхода DDD. Это приводит к довольно типичной ситуации, в которой небольшое количество подсистем несут ответственность за многие бизнес-функции.

Внутри ОГРАНИЧЕННОГО КОНТЕКСТА *электронной торговли* на самом деле существует множество неявных моделей предметной области, причем некоторые из них невозможно точно отделить друг от друга. Если бы применялся другой подход, эти модели были бы отделены друг от друга, а пока они фактически слиты в одну модель программного обеспечения, и это очень неудачно. Розничная компания уменьшила бы количество проблем, просто купив этот ОГРАНИЧЕННЫЙ КОНТЕКСТ у стороннего поставщика, вместо того, чтобы создавать его самой, но кто бы ни обслуживал эту систему, он испытает отрицательные последствия увеличивающейся сложности, которая следует из смешивания моделей *каталога продукции*, *заказов*, *выставления счетов* и *доставки* в одну большую модель электронной торговли. Для поддержки увеличивающегося количества функций логические модели должны расти и каждая из конфликтующих проблем будет препятствовать решению других. Ситуация еще более обострится, если потребуется добавить новую логическую модель — новый набор основных функций. Вот что происходит, когда проблемы программного обеспечения не удается четко разделить.

Это особенно неудачно потому, что многие разработчики программного обеспечения считают, что впихнуть все в одну систему — это очень умное решение. Вы получаете все знающую и все умеющую систему электронной торговли, которая, конечно же, удовлетворяет все потребности. Однако это самообман, так как сколько бы функций ни было втиснуто в одну подсистему, она никогда не будет учитывать потребности каждого потенциального потребителя. Никогда. Добавьте к этому тот факт, что, не разделенные, но разные модели предметной области программного обеспечения в ПОДОБЛАСТИ намного усложняют внесение постоянных изменений, так как все будет связано друг с другом и зависеть от всего остального.

Тем не менее, используя одно из стратегических средств проектирования DDD, мы можем до некоторой степени снизить сложность, разбивая извне эти переплетенные модели на логически разделенные ПОДОБЛАСТИ согласно их фактической функциональности. Границы между ПОДОБЛАСТЯМИ на рис. 2.1 обозначены пунктирными линиями. Это не значит, что мы осуществили рефакторинг сторонних моделей, четко разграничив их. Мы просто указали, какие модели должны существовать отдельно, по крайней мере потому, что они применяются к бизнес-операциям нашей конкретной розничной компании. Мы также провели несколько соединительных линий между логическими ПОДОБЛАСТЯМИ и даже физическими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, чтобы продемонстрировать интеграцию.

Теперь перейдем от технических сложностей к бизнес-проблемам, стоящим перед нашей небольшой компанией. Ее финансы и складская площадь ограничены. По этой причине ей постоянно приходится маневрировать. Компания не должна тратить деньги на товары, которые не имеют спроса, причем некоторые товары продаются лучше в определенные периоды времени и хуже в другие. Очевидно, если некоторые товары не продаются по плану, то средства компании оказываются вложенными в товары, на которые нет спроса у клиентов, по крайней мере временно. Итак, деньги заморожены. В результате компания имеет недостаточную площадь для продажи товаров, пользующихся хорошим спросом в любой момент времени.

Это еще не все. Описанные выше сложности приводят к еще одной проблеме. Если некоторые товары продаются быстрее, чем ожидалось, то компания не будет в состоянии делать достаточно большие запасы этих товаров, чтобы удовлетворить потребительский спрос. Проблема недостаточных запасов может оттолкнуть клиентов к другим продавцам. Конечно, некоторые оптовые торговцы предпочитают делать прямые поставки от имени розничной компании, но эта возможность стоит дороже и порождает другие нежелательные последствия. Существуют также экономные стратегии, предусматривающие хранение некоторых товаров поблизости от мест их потребления и прямую поставку товаров повышенного спроса в удаленные регионы. Таким образом, прямые поставки могут способствовать успеху розничной компании, но не могут использоваться как средство срочного спасения падающего объема продаж. Это не значит, что существует дефицит товаров повышенного спроса. Просто их трудно приобрести у маленькой розничной компании, потому что она не может оптимизировать их запасы. Если клиенты постоянно будут испытывать задержки поставок, то компания, вероятно, потеряет значительную часть своего конкурентного преимущества, завоеванного ранее. В этом примере отражены проблемы заказчиков, которые обычно решает компания Lokad.[1]

[1] www.locad.com.

Уточним, что мы не исследовали масштаб проблем, связанных с товарными запасами, и эти нежелательные ситуации характерны не только для небольших розничных компаний. Любые розничные компании хотят покупать и запасать товары в точном соответствии с потребностями, минимизируя затраты и оптимизируя объем продаж согласно спросу. И все же небольшие розничные компании сильнее страдают от неоптимальных действий, чем крупные.

Интернет-магазины могли бы существенно улучшить положение, если бы определяли объем будущих запасов и продаж, основываясь на прошлых тенденциях. Если розничная компания могла бы использовать механизм прогноза, передавая ему данные о запасах и истории продаж, то она могла бы получить прогнозы спроса и конкретные объемы оптимальных запасов. На основе этой информации она могла бы изменить порядок и объем поставок каждого товара.

Чтобы добавить такую функциональную возможность для небольшой розничной компании, вероятно, пришлось бы создать новое **СМЫСЛОВОЕ ЯДРО**, потому что решение этой проблемы нетривиально, но имеет большое значение для получения нового конкурентного преимущества. Фактически третий физический ОГРАНИЧЕННЫЙ КОНТЕКСТ на рис. 2.1 — это *внешняя система прогнозирования*. ПОДОБЛАСТЬ *заказов* и ОГРАНИЧЕННЫЙ КОНТЕКСТ *товарных запасов* интегрируются с *прогнозированием*, чтобы предоставить историческую информацию о продажах и возвратах товаров. Кроме того, у нас должна быть ПОДОБЛАСТЬ *каталога товаров*, обеспечивающая глобальное распознавание штрих-кодов товаров для *прогнозирования*, чтобы можно было сравнить ассортимент товаров небольшой розничной компании со связанными и аналогичными тенденциями продаж во всем мире, давая более широкую перспективу. Это приводит к необходимости создания механизма *прогнозирования*, обладающего средствами для вычисления самых точных чисел, необходимых небольшой розничной компании для правильного снабжения.

Если бы этим новым решением было СМЫСЛОВОЕ ЯДРО, что наиболее вероятно, то команда, разрабатывающая его, извлекла бы большую выгоду из понимания окружающего бизнес-ландшафта, составленного из логических ПОДОБЛАСТЕЙ и необходимой интеграции. Таким образом, выделение уже существующей интеграции, обозначенной на рис. 2.1, является ключевым моментом для осознания ситуации в самом начале проекта.

Впрочем, ПОДОБЛАСТИ не всегда используются для выделения моделей значительного размера и функциональности. Иногда ПОДОБЛАСТЬ может быть простой, как набор алгоритмов, который важен для бизнес-решения, но не являться частью другого СМЫСЛОВОГО ЯДРА. Применяя проверенные методы DDD, такие простые ПОДОБЛАСТИ можно отделить от ЯДРА с помощью **МОДУЛЕЙ (MODULES) (9)** и не помещать их в тяжелый, архитектурно важный компонент подсистемы.

Применяя подход DDD, мы стремимся отделить каждый ОГРАНИЧЕННЫЙ КОНТЕКСТ, в котором значение каждого термина, использованного моделью предметной области, является хорошо понятным, или по крайней мере должно быть хорошо понятным, если мы правильно выполнили моделирование программного обеспечения. Границы контекстов в основном носят *лингвистический* характер. Они являются ключом к реализации DDD.

Ковбойская логика

LB: У нас с соседями не было никаких проблем, пока не упал забор.

AJ: Правильно. Чем выше заборы, тем лучше соседи.

Обратите внимание на то, что отдельный ОГРАНИЧЕННЫЙ КОНТЕКСТ не обязательно должен находиться в пределах отдельной ПОДОБЛАСТИ. На рис. 2.1 только один ОГРАНИЧЕННЫЙ КОНТЕКСТ *товарных запасов* находится в пределах только одной подобласти.[2] Довольно очевидно, что при разработке *системы электронной торговли* подход DDD не использовался. В этой системе мы идентифицировали четыре ПОДОБЛАСТИ, хотя в ней есть, вероятно, больше ПОДОБЛАСТЕЙ. С другой стороны, в *системе товарных запасов* действительно одной ПОДОБЛАСТИ соответствует один ОГРАНИЧЕННЫЙ КОНТЕКСТ, ограничивая его модель предметной области инвентаризацией товаров. Очевидно, ясность модели *системы товарных запасов* может быть следствием применения принципов DDD, хотя это может быть и случайностью. Для того чтобы выяснить этот вопрос, следовало бы разобраться в технических деталях. Тем не менее мы все еще можем применять *систему товарных запасов* для разработки нового СМЫСЛОВОГО ЯДРА.

Какой из ОГРАНИЧЕННЫХ КОНТЕКСТОВ, изображенных на рис.2.1, имеет лучшую лингвистическую структуру? Иначе говоря, какой из них содержит непротиворечивый набор терминов, специфичных для предметной области? Поскольку в *системе электронной торговли* существует не менее четырех ПОДОБЛАСТЕЙ, очень вероятно, что термины и значения в них будут совпадать. Например, термин *клиент* должен иметь несколько значений. Когда клиент просматривает **каталог**, этот термин означает одно, а когда пользователь размещает **заказ** — другое. Разберемся, почему это происходит. Когда **клиент** листает **каталог**, этот термин находится в контексте предыдущих покупок, лояльности, доступных товаров, скидок и вариантов доставки. Однако, когда **клиент** размещает **заказ**, смысл термина

[2] Правда, ПОДОБЛАСТЬ *доставки* использует ПОДОБЛАСТЬ *товарных запасов*, но это не делает ПОДОБЛАСТЬ *товарных запасов* частью *системы электронной торговли*, для которой *доставка* является контекстом.

сужается. Он сводится к нескольким деталям, таким как фамилия и адрес доставки, адрес выставления счета, сумма к оплате и время доставки. Уже эти простые рассуждения показывают, что *система электронной торговли* не содержит одного четкого определения термина *клиент*. В этой ситуации можно предположить, что и другие отдельные термины могут иметь несколько значений. Итак, это не четкий ОГРАНИЧЕННЫЙ КОНТЕКСТ с ясным смыслом каждого термина, именующего понятие предметной области.

Впрочем, у нас нет никаких гарантий, что *система товарных запасов* имеет абсолютно точную и однозначную модель. Даже в данном узкоспециализированном КОНТЕКСТЕ мы сталкиваемся с разными значениями терминов, которыми оперируют в *системе товарных запасов*. Это вызвано тем, что существуют разные способы использования **единиц хранения** (items). Есть ли явное различие между заказанной, полученной, хранящейся и выданной со склада **единицей хранения**? Заказанная **единица хранения**, которая еще не доступна для продажи, вызывается **невыданной единицей хранения** (back-ordered item). Получаемая **единица хранения** часто называется **принятым товаром** (goods received). **единица хранения**, находящаяся на складе, называется **номенклатурной позицией** (stock item). Потребляемая **единица хранения** часто называется **товаром, выдаваемым со склада** (item leaving inventory). Испорченные или поврежденные **единицы хранения** часто называются **отходами при хранении** (wasted inventory item).

Из рис. 2.1 мы не знаем, насколько хорошо смоделирован диапазон понятий, связанных с хранением товаров и их сопроводительной лингвистики. При использовании DDD мы не стали бы угадывать. Мы были бы уверены, что каждое из этих понятий хорошо понятно, упоминается явно и моделируется как таковое. Способ, которым эксперты в предметной области описывают каждое из этих понятий, мог бы привести к разделению некоторых из них в различных ОГРАНИЧЕННЫХ КОНТЕКСТАХ.

По внешнему виду можно сказать, что *система товарных запасов* лучше соответствует принципам DDD, чем *система электронной торговли*. Возможно, команда, работавшая над ее моделью, не пыталась сделать так, чтобы **единица хранения** представляла все возможные ситуации, связанные с хранением товаров. Несмотря на неопределенность, вполне возможно, что модель *системы товарных запасов* будет легче интегрировать с другими моделями, чем *систему электронной торговли*.

Говоря об интеграции, следует подчеркнуть, что на рис. 2.1 ясно продемонстрировано, что ОГРАНИЧЕННЫЕ КОНТЕКСТЫ на предприятии редко бывают полностью изолированными. Даже если представить себе стороннюю *систему электронной торговли* в виде крупной, всеохватывающей модели, она не сможет удовлетворить все потребности розничной компании. Сплошные линии, соединяющие ПОДОБЛАСТИ в *системе электронной торговли*, *системе товарных запасов* и *внешней системе прогнозирования*, показывают, как должны работать необходимые отношения

интеграции, объединяющие разные модели. В интеграции всегда существуют отношения особого вида. О возможностях интеграции вы узнаете, когда мы будем обсуждать **КАРТЫ КОНТЕКСТОВ (CONTEXTS MAPS) (3)**.

Итак, мы сделали краткий обзор одного из представлений простой предметной области. Мы вскользь упомянули о СМЫСЛОВОМ ЯДРЕ и получили представление о его важной роли в подходе. Настало время разобраться в нем получше.

Внимание на СМЫСЛОВОЕ ЯДРО

Понимая смысл ПОДОБЛАСТЕЙ и ОГРАНИЧЕННЫХ КОНТЕКСТОВ, рассмотрим абстрактное представление другой ПРЕДМЕТНОЙ ОБЛАСТИ, показанное на рис. 2.2. Это представление может относиться к любой предметной области, возможно, даже к той, в которой вы работаете. Я удалил конкретные названия; таким образом, вы можете мысленно подставить свои. Естественно, наши бизнес-цели должны постоянно уточняться и расширяться, что должно отражаться в постоянно меняющихся ПОДОБЛАСТЯХ и содержащихся в них моделях. На этой диаграмме изображена лишь общая ПРЕДМЕТНАЯ ОБЛАСТЬ в определенный момент времени и с определенной точки зрения, которая может измениться.

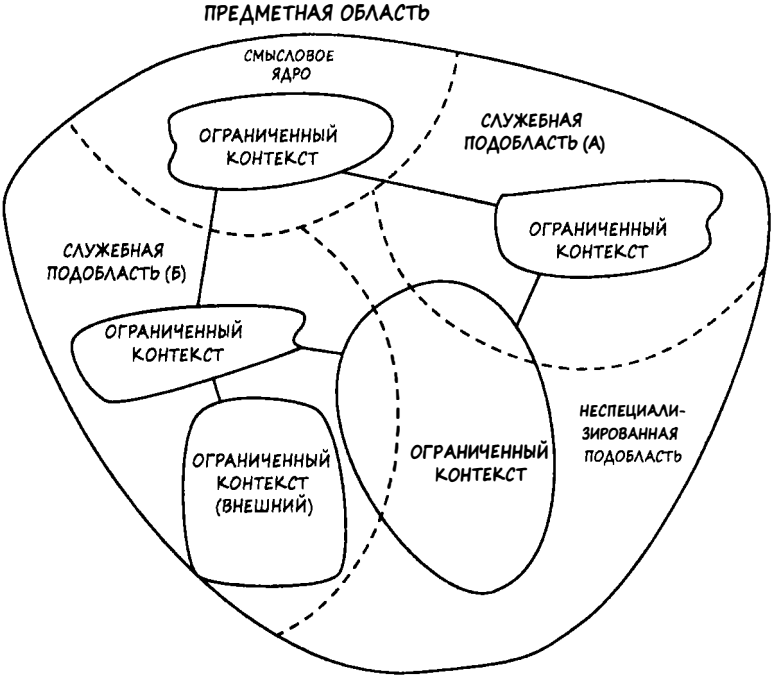


Рис. 2.2. Абстрактная ПРЕДМЕТНАЯ ОБЛАСТЬ, **содержащая** ПОДОБЛАСТИ **и** ОГРАНИЧЕННЫЕ КОНТЕКСТЫ

Заметки на доске

- В одном столбце запишите список всех ПОДОБЛАСТЕЙ, с которыми вы сталкиваетесь в вашей повседневной работе. В другом столбце перечислите ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. Пересекаются ли ПОДОБЛАСТИ с несколькими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ? Если да, то это не обязательно плохо. Возможно, это особенность программного обеспечения предприятия.
- Теперь, используя шаблон, показанный на рис. 2.2, напишите на доске несколько названий программного обеспечения, работающего на предприятии вместе с ПОДОБЛАСТЯМИ, ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, и нарисуйте линии интеграционных отношений между ними.

Это было трудно? Возможно, потому что шаблон, изображенный на рис. 2.2, скорее всего, не очень точно отображает существующие границы в вашей ПРЕДМЕТНОЙ ОБЛАСТИ.

- Начните сначала. На этот раз вы должны нарисовать диаграмму, соответствующую *вашим* ПРЕДМЕТНОЙ ОБЛАСТИ, ПОДОБЛАСТЯМ и ОГРАНИЧЕННЫМ КОНТЕКСТАМ. Используйте прием, показанный на рис. 2.2, но не останавливайтесь на этом и приспосабливайте его к своим потребностям.

Разумеется, вы можете не знать о существовании каждой ПОДОБЛАСТИ и ОГРАНИЧЕННОГО КОНТЕКСТА на всем вашем предприятии, особенно если ваша ПРЕДМЕТНАЯ ОБЛАСТЬ действительно большая и сложная. Однако вы можете выделить те из них, с которыми вам приходится сталкиваться в повседневной работе. В любом случае попытайтесь. Не бойтесь ошибиться. Вы получите небольшой опыт работы с КАРТАМИ КОНТЕКСТОВ, который вы сможете уточнить в следующей главе. Если вы захотите после этого перейти к изучению следующей главы, на здоровье. А пока мы не будем углубляться в детали и сначала усвоим основные идеи.

Теперь посмотрите на верхнюю границу предметной области на рис. 2.2. Вы увидите ПОДОБЛАСТЬ с надписью СМЫСЛОВОЕ ЯДРО. Это еще один очень важный аспект подхода DDD. **СМЫСЛОВОЕ ЯДРО** — это часть ПРЕДМЕТНОЙ ОБЛАСТИ, имеющая первостепенное значение для организации. Со стратегической точки зрения бизнес должен выделяться своим СМЫСЛОВЫМ ЯДРОМ. Это ядро имеет исключительную важность для успеха бизнеса. Этот проект получает наивысший приоритет, в него включаются один или несколько экспертов в предметной области с глубокими знаниями данной ПОДОБЛАСТИ, лучшие разработчики и столько ресурсов и средств, сколько возможно, чтобы сплоченная команда беспрепятственно достигла успеха. Большинство ваших DDD-проектов будет сосредоточено на СМЫСЛОВОМ ЯДРЕ.

На рис. 2.2 изображены еще два вида ПОДОБЛАСТЕЙ — СЛУЖЕБНАЯ и НЕСПЕЦИАЛИЗИРОВАННАЯ. Иногда ОГРАНИЧЕННЫЙ КОНТЕКСТ создается или приобретается для поддержки бизнеса. Если он моделирует некий аспект бизнеса, который важен, но не является СМЫСЛОВЫМ ЯДРОМ, то он относится к **СЛУЖЕБНОЙ ПОДОБЛАСТИ**. Бизнес создает СЛУЖЕБНУЮ ПОДОБЛАСТЬ, потому что она имеет специализацию. Если же она не имеет специального предназначения для бизнеса, а требуется для всего бизнеса в целом, то ее называют **НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ**. Если ПОДОБЛАСТЬ является СЛУЖЕБНОЙ или НЕСПЕЦИАЛИЗИРОВАННОЙ, это еще не значит, что она не имеет важности. Эти виды ПОДОБЛАСТЕЙ важны для успеха бизнеса, хотя и не имеют первоочередного значения. Именно СМЫСЛОВОЕ ЯДРО должно быть реализовано идеально, поскольку оно обеспечивает преимущества для бизнеса.

Заметки на доске

- Для того чтобы убедиться, что вы осознали важность концепций СМЫСЛОВОГО ЯДРА, вы должны стереть доску и попытаться выделить СМЫСЛОВОЕ ЯДРО вашей организации.
- Затем проверьте, можете ли вы идентифицировать СЛУЖЕБНЫЕ ПОДОБЛАСТИ и НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ в вашей ПРЕДМЕТНОЙ ОБЛАСТИ.

Не забывайте задавать вопросы экспертам в предметной области!

Даже если у вас нет возможности задать вопросы экспертам в предметной области прямо сейчас, это упражнение поможет вам тщательно продумать, какое программное обеспечение имеет первостепенное значение для вашего бизнеса, какое программное обеспечение носит служебный характер, а какое вообще не влияет на успех вашего бизнеса. Проработав этот вопрос, вы будете увереннее размышлять о процессах и методах.

Обсудите каждую ПОДОБЛАСТЬ и ОГРАНИЧЕННЫЙ КОНТЕКСТ, нарисованные вами на диаграмме, с несколькими экспертами в предметной области, специализирующимися в разных сферах.

Вы не только многое от них узнаете, но и получите ценный опыт, относящийся к *умению слушать экспертов*. Это отличительная особенность правильной реализации принципов DDD.

Все, что вы сейчас узнали, является основой для стратегического проектирования.

Чем объяснить невероятную важность стратегического проектирования

Итак, вы уже усвоили часть терминологии DDD и ее смысл, но пока не знаете, *почему* она является настолько важной. Я просто утверждал, что она очень важная, и надеялся, что вы мне поверите. Но я все же предпочел обосновать свою точку зрения. Давайте вернемся к нашему примеру, в котором описываются проекты компании SaaSOvation. Как мы видели в предыдущей главе, наши разработчики зашли в тупик.

На первом этапе проектирования на основе подхода DDD команда разработчиков стала отклоняться от правильного курса, ведущего к разработке ясной модели. Это произошло потому, что она не поняла принципов стратегического проектирования даже на базовом уровне. Как и большинство разработчиков, она сосредоточилась на деталях **СУЩНОСТЕЙ (ENTITIES)** (5) и **ОБЪЕКТОВ-ЗНАЧЕНИЙ (VALUE OBJECTS)** (6). Это затуманило всю картину. Они смешали концепции смыслового ядра со служебными, что привело к слиянию двух моделей в одну. Вскоре разработчики стали испытывать сложности проектирования, показанные на рис. 2.3. Каков итог? Они не полностью достигли цели, связанной с реализацией подхода DDD.

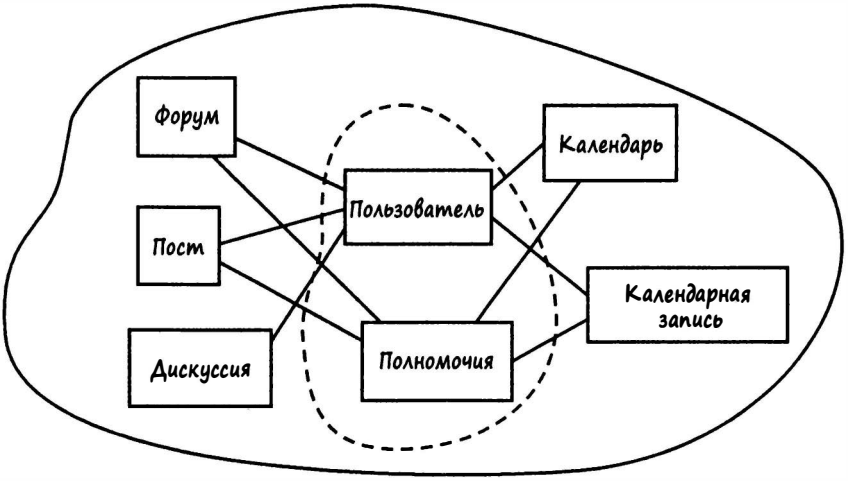


Рис. 2.3. Команда не понимает основ стратегического проектирования. Это ведет к неправильным концепциям в модели сотрудничества. Проблемные элементы обведены пунктирной линией

Некоторые из сотрудников команды SaaSOvation говорили: "Что если концепции корпоративного сотрудничества тесно связаны с пользователями и полномочиями? Мы должны отслеживать, кто и что делает!" Старший разработчик указал, что эти вопросы на самом деле не должны интересовать команду. "В конце концов, форум, пост, дискуссия, календарь и календарная запись будут объединены в некие объекты *сотрудничества* между людьми. *В этом-то и дело. Здесь используется неправильная лингвистика".* Развивая свою аргументацию, он показал, что форум, пост, дискуссия и тому подобные объекты связаны с неправильной лингвистической концепцией. Термины пользователь и полномочия *не имеют никакого отношения к взаимодействию и никак не гармонизируют* *ЕДИНЫЙ ЯЗЫК СОТРУДНИЧЕСТВА.* пользователи и полномочия — это сущности и концепции доступа, относящиеся к вопросам безопасности. Каждая концепция, моделируемая в *контексте сотрудничества* (*collaboration context*) — т.е. в ОГРАНИЧЕННОМ КОНТЕКСТЕ, охватывающем модель предметной области сотрудничества, — должна иметь лингвистическую ассоциацию с сотрудничеством, а в данный момент этого нет. "Мы должны сосредоточиться на концепциях сотрудничества, таких как автор и модератор. Это правильные концепции и лингвистические термины, имеющие отношение к сотрудничеству".

Как назвать ОГРАНИЧЕННЫЙ КОНТЕКСТ

Вы обратили внимание на название *контекст сотрудничества*, который мы использовали выше? Так мы назвали ОГРАНИЧЕННЫЙ КОНТЕКСТ, использовав формат *контекст название-модели*. В данном случае выбрано название *контекст сотрудничества*, потому что этот ОГРАНИЧЕННЫЙ КОНТЕКСТ содержит модель предметной области проекта *сотрудничество*. Существует также *контекст идентификации и доступа* (*identity and access context*), представляющий собой ОГРАНИЧЕННЫЙ КОНТЕКСТ, содержащий модель проекта *идентификация и доступ*, и *контекст управления гибким проектированием (agile project management (PM) context)*, представляющий собой ОГРАНИЧЕННЫЙ КОНТЕКСТ, содержащий модель *управление гибким проектированием*.

Напомним, что сначала разработчики компании SaaSOvation не понимали, что пользователи и полномочия не имеют никакого отношения к инструментам сотрудничества. Конечно, существуют пользователи их программного обеспечения и их необходимо отличать друг от друга, чтобы определить задачи, которые они могут выполнять. Но для инструментов сотрудничества важны роли пользователей, а не их личности и полномочия. В то же время сейчас модель сотрудничества содержит подробные описания пользователей и полномочий, тесно переплетенные друг с другом. Если атрибуты пользователей и/или полномочия изменятся, это повлияет на большую часть модели или даже на всю модель. Фактически эта проблема возникла уже в самом начале. Команда хотела отказаться от подхода, основанного на полномочиях, и переключиться на управление доступом на основе ролей. Когда она решила выполнить это переключение, перед ней встали проблемы стратегического моделирования.

Теперь они поняли, что форум не должно интересовать, кто разместил на нем пост и почему ему это разрешено. форум должен просто знать, что сейчас на нем действует или недавно действовал некий автор. Команда начинает понимать, что определение сущностей, которые могут что-то делать, — это задача совсем другой модели, а основная модель сотрудничества должна знать лишь, что любой вопрос о том, кто и что может делать, уже решен. форум должен просто знать автора, желающего разместить пост или открыть дискуссию. Форум и автор — это ясные концепции ЕДИНОГО ЯЗЫКА модели сотрудничества и ОГРАНИЧЕННОГО КОНТЕКСТА под названием *контекст сотрудничества.* Пользователь и полномочия или подобные им концепции, например роль, относятся к совершенно другим областям. Их следует изолировать от *контекста сотрудничества.*

Команда могла бы легко прийти к выводу, что достаточно изменить тесную связь между пользователем и полномочиями. Кроме всего прочего, нет ничего плохого в том, что пользователь и полномочия/роль будут выделены в отдельный модуль. Это помогло бы выделить эти концепции в отдельную логическую *подобласть безопасности* внутри того же ОГРАНИЧЕННОГО КОНТЕКСТА. Однако для правильного выбора методов моделирования следует помнить, что следующий проект команды может содержать такие же или похожие инструменты доступа на основе ролей и их предметно-ориентированных характеристик. Очевидно, что пользователи и роли действительно являются частью СЛУЖЕБНОЙ или НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТИ, охватывающих сотрудников предприятия и даже (возможно, в будущем) клиентов.

Более решительный подход к четкому моделированию мог бы помочь команде избежать еще большей проблемы. Похоже, что они уже прокладывали себе путь в **БОЛЬШОЙ КОМОК ГРЯЗИ (BIG BALL OF MUD) (3)**. Это возможно не просто потому, что их концепции пользователя и полномочия не были должным образом агрегированы. Хотя модульное проектирование — это важный инструмент моделирования DDD, оно не фиксирует лингвистические несоответствия.

Старшего разработчика очень беспокоило, что без контроля эта ситуация могла бы легко привести к *недисциплинированному мышлению, которое привело бы к еще большей путанице и в конечном счете завело бы команду в тупик.* Если бы перед командой была поставлена задача построить модель другого набора концепций, не связанных с сотрудничеством, то СМЫСЛОВОЕ ЯДРО могло бы стать еще менее четким. Она могла бы создать только неявную модель с исходным кодом, который не отражал бы выразительный ЕДИНЫЙ ЯЗЫК СОТРУДНИЧЕСТВА. Команда должна была понять свою ПРЕДМЕТНУЮ ОБЛАСТЬ, *ее* ПОДОБЛАСТИ и ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. Это позволило бы избежать коварной ловушки, подстерегающей разработчиков на пути стратегического проектирования — БОЛЬШОГО КОМКА ГРЯЗИ. Таким образом, *команде необходимо научиться методам стратегического моделирования.*

О, нет! Снова это *проектирование*!

Если вы думаете, что *проектирование* — это бранное слово в среде гибкой разработки, то к подходу DDD это не относится. Сочетание принципов DDD и гибкой разработки вполне естественно. Всегда держите проектирование под контролем с помощью методов гибкой разработки. Проектирование не обязательно должно быть трудным.

Да, это был важный урок. Команда хорошо поработала и в результате выделила свои ПРЕДМЕТНУЮ ОБЛАСТЬ и ПОДОБЛАСТИ. Как она это сделала, мы вскоре расскажем.

Комментарии для сообщества DDD

Сценарии, рассматриваемые в книге, содержат три ОГРАНИЧЕННЫХ КОНТЕКСТА. Эти ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, скорее всего, отличаются от тех контекстов, в которых работаете вы. Эти примеры иллюстрируют типичные ситуации моделирования. Однако не все согласятся, что **пользователи** и **полномочия** должны быть отделены от СМЫСЛОВОГО ЯДРА. В некоторых случаях имеет смысл объединить их с МОДЕЛЬЮ СМЫСЛОГО ЯДРА. Как всегда, этот выбор должна делать конкретная команда. Однако мой опыт подсказывает, что это одна из основных проблем, с которыми сталкиваются специалисты, впервые использующие принципы DDD, и эта проблема может обесценить их попытки реализации, приведя к запутанному результату. Другой распространенной ошибкой является смешивание моделей сотрудничества и гибкого проектирования в одну модель. Эти лишь некоторые из общих проблем. Другие ошибки, связанные с моделированием, обсуждаются в каждой главе.

По крайней мере, описанные здесь проблемы являются следствием *типичных* ошибок моделирования, которые совершают команды, плохо понимающие важность лингвистических драйверов и ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Таким образом, даже если вам не нравятся конкретные примеры, проблемы и решения, описанные в них, все же относятся ко всем DDD-проектам, потому что все они все сосредоточены на лингвистике данного ОГРАНИЧЕННОГО КОНТЕКСТА.

Моя цель — изложить принципы реализации DDD с помощью самых простых, но все же нетривиальных, возможных примеров. Я не могу позволить примерам помешать обучению. Если я показываю, что управление идентификацией и доступом, сотрудничество и управление гибким проектированием имеют свою лингвистику, то примеры должны это подчеркивать. Поскольку каждая *команда* сама выявляет лингвистические драйверы, которые помогают *ей* понимать точку зрения ее экспертов в предметной области, будем предполагать, что команда SaaSOvation сделала правильные "окончательные" выводы и решения, связанные с моделированием на основе принципов DDD.

Все мои рекомендации относительно ПОДОБЛАСТЕЙ и ОГРАНИЧЕННЫХ КОНТЕКСТОВ соответствуют общей точке зрения более широкого сообщества DDD, поскольку они отражают мой собственный опыт. У других лидеров сообщества DDD может быть немного отличающиеся мнение. Однако мои объяснения, определенно, закладывают твердую основу для любой команды, чтобы избежать неоднозначности. Разъяснение темных областей DDD является самой важной услугой сообществу, и в этом заключается моя основная цель. Для того чтобы эти рекомендации принесли пользу вашему проекту, вы должны широко использовать их на практике.

Реальные предметные области и подобласти

Я хочу сказать о предметных областях еще кое-что. Они состоят из *пространства задач* (problem space) и *пространства решений* (solution space). Пространство задач позволяет нам думать о стратегической бизнес-проблеме, которая должна быть решена, в то время как пространство решений позволяет сосредоточиться на том, как мы реализуем программное обеспечение, чтобы решить бизнес-проблему. Вот то, как это согласуется с тем, что вы уже изучили.

- *Пространство задач* — это части ПРЕДМЕТНОЙ ОБЛАСТИ, которые необходимо выделить, чтобы создать новое СМЫСЛОВОЕ ЯДРО. Исследование пространства задач подразумевает изучение ПОДОБЛАСТЕЙ, *которые уже существуют и которые должны существовать*. Таким образом, ваше пространство задач — это комбинация СМЫСЛОВОГО ЯДРА и ПОДОБЛАСТЕЙ, которые это ядро должно использовать. ПОДОБЛАСТИ в пространстве задач обычно уникальны для каждого проекта, потому что используются при исследовании текущей стратегической бизнес-проблемы. По этой причине ПОДОБЛАСТИ представляют собой очень полезный инструмент для исследования пространства задач. ПОДОБЛАСТИ позволяют быстро увидеть разные части ПРЕДМЕТНОЙ ОБЛАСТИ, необходимые для решения конкретной задачи.
- Пространство решений состоит из одного или нескольких ОГРАНИЧЕННЫХ КОНТЕКСТОВ, набора конкретных моделей программного обеспечения. Это объясняется тем, что разработанный ОГРАНИЧЕННЫЙ КОНТЕКСТ — это *конкретное решение*, т.е. *представление реализации*. ОГРАНИЧЕННЫЙ КОНТЕКСТ используется для реализации решения в виде программного обеспечения.

Желательно обеспечить однозначное соответствие между ПОДОБЛАСТЯМИ и ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Это позволит выделить модели предметной области в четко определенные области в зависимости от целей, объединив пространство задач с пространством решений. Практически это не всегда возможно, за исключением проектов, которые разрабатываются "с нуля". Однако если система унаследована и, вероятно, представляет собой БОЛЬШОЙ КОМОК ГРЯЗИ, то ПОДОБЛАСТИ часто пересекаются с ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, как показано на рис. 2.1. На крупном и сложном предприятии мы можем использовать *представление оценки* (assessment view), для того чтобы понять наше пространство задач, и это может спасти нас от дорогостоящих ошибок. Мы можем концептуально разделить один большой ОГРАНИЧЕННЫЙ КОНТЕКСТ на две или больше ПОДОБЛАСТЕЙ или несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, являющихся частью отдельной ПОДОБЛАСТИ. Придумайте пример, помогающий выявить различие между пространством задач и пространством решений.

Представьте себе большую, монолитную систему, классифицированную как ERP-приложение. Строго говоря, ERP может считаться единственным ОГРАНИЧЕННЫМ КОНТЕКСТОМ. Однако, так как системы ERP обеспечивают много модульных бизнес-услуг, удобно думать о разных модулях как о разных ПОДОБЛАСТЯХ. Например, мы могли бы разделить *модуль товарных запасов* и *модуль закупок* на отдельные логические ПОДОБЛАСТИ. Правда, эти модули не находятся в абсолютно разных системах. Оба они являются частями одной и той же системы ERP. Однако каждый из них обеспечивает совершенно другой набор услуг для деловой предметной области. Для проведения аналитических дискуссий назовем эти отдельные подобласти *подобластью товарных запасов* и *подобластью закупок*. Разбирая пример, мы увидим, насколько это полезно.

В качестве основной деловой инициативы организация, ПРЕДМЕТНАЯ ОБЛАСТЬ которой представлена на рис. 2.4 (это конкретный пример на основе шаблона, показанного на рис. 2.2), приступает к планированию проектирования и разработки специализированной модели предметной области, чтобы уменьшить затраты на эксплуатацию предприятия. Модель позволит создать инструменты принятия решений, которые будут использоваться агентами по закупкам. Алгоритмы, выработанные людьми за годы руководства, теперь необходимо автоматизировать с помощью программного обеспечения, чтобы гарантировать, что они всегда будут правильно использоваться всеми агентами по закупкам. Это новое СМЫСЛОВОЕ ЯДРО *сделает организацию более конкурентоспособной*, способной более быстро идентифицировать выгодные сделки и гарантировать необходимый уровень запасов. Для аккуратного учета товарных запасов целесообразно использовать *систему прогнозирования* (см. рис. 2.1).

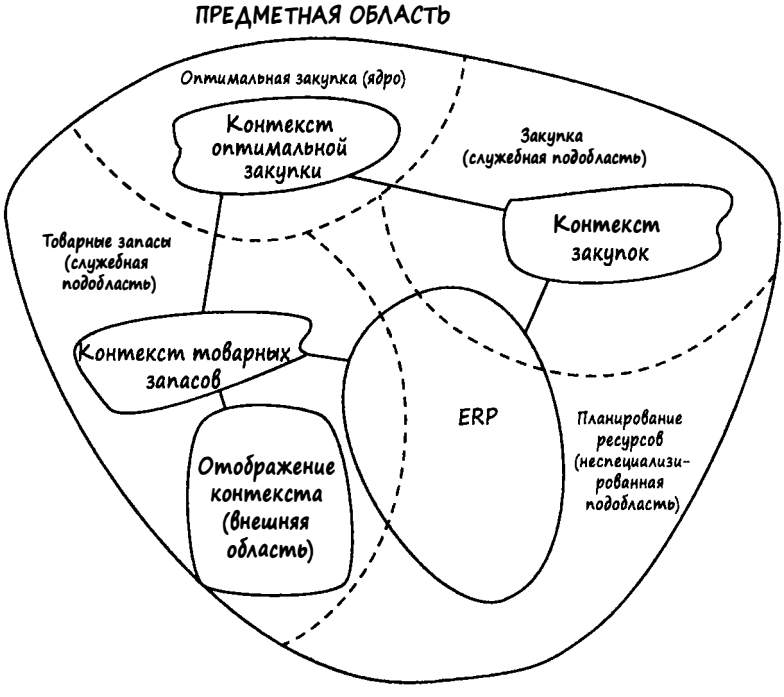


Рис. 2.4. СМЫСЛОВОЕ ЯДРО **и другие** ПОДОБЛАСТИ, **связанные с закупками и запасами. В этом представлении показаны лишь** ПОДОБЛАСТИ, **связанные с анализом конкретного пространства задач, а не вся** ПРЕДМЕТНАЯ ОБЛАСТЬ

Прежде чем принять определенное решение, мы должны сделать оценку пространства задач и пространства решений. Вот некоторые вопросы, на которые нужно ответить, чтобы сориентировать ваш проект в правильном направлении.

- Как называется и выглядит стратегическое СМЫСЛОВОЕ ЯДРО?
- Какие концепции должны рассматриваться как часть стратегического СМЫСЛОВОГО ЯДРА?
- Перечислите необходимые СЛУЖЕБНЫЕ ПОДОБЛАСТИ и НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ.
- Кто должен работать в каждой области предметной области?
- Можно ли собрать правильные команды?

Если мы не поймем концепцию и цели СМЫСЛОВОГО ЯДРА и не выделим части ПРЕДМЕТНОЙ ОБЛАСТИ, которые необходимы для его поддержки, то не будем в состоянии стратегически использовать их в своих интересах и избежать ловушек. Поддерживайте высокий уровень оценки пространства задач, но делайте это

аккуратно. Убедитесь, что все заинтересованные лица достигли согласия и нацелены на успешное воплощение концепции.

Заметки на доске

Посмотрите на доску и подумайте: "Каково наше пространство задач?" Напоминаем, что это — комбинация стратегической ПРЕДМЕТНОЙ ОБЛАСТИ и поддерживающих ее ПОДОБЛАСТЕЙ.

Когда вы хорошо поймете свое пространство задач, вернитесь в пространство решений. Оценка первого пространства влияет на оценку второго. Пространство решений сильно зависит от существующих и вновь созданных систем и технологий. Здесь мы действительно должны думать в терминах четко отделимых ОГРАНИЧЕННЫХ КОНТЕКСТОВ, потому что для каждого из них мы ищем ЕДИНЫЙ ЯЗЫК. Рассмотрим следующие чрезвычайно важные вопросы.

- Какое программное обеспечение уже существует и может использоваться повторно?
- Какие средства следует приобрести или создать?
- Как все они связаны друг с другом, т.е. интегрированы?
- Какая требуется дополнительная интеграция?
- Какие действия следует выполнить, имея необходимые существующие и созданные средства?
- Насколько высока вероятность успеха стратегической инициативы и всех вспомогательных проектов и может ли один из них вызвать задержку или даже крах всей программы?
- Являются ли используемые термины ЕДИНОГО ЯЗЫКА совершенно разными?
- Как перекрываются концепции и данные разных ОГРАНИЧЕННЫХ КОНТЕКСТОВ?
- Как перекрывающиеся термины и концепции отображаются и переводятся в ОГРАНИЧЕННЫХ КОНТЕКСТАХ?
- Какой ОГРАНИЧЕННЫЙ КОНТЕКСТ содержит концепции, относящиеся к СМЫСЛОВОМУ ЯДРУ, и какие тактические шаблоны [Эванс] используются для его моделирования?

Напоминаем, что усилия, направленные на разработку решений в СМЫСЛОВОМ ЯДРЕ, являются основными бизнес-инвестициями!

Специализированная модель закупок, описанная ранее и изображенная на рис. 2.4 (на котором изображены инструменты принятия решений и алгоритмы), представляет собой решение для СМЫСЛОВОГО ЯДРА. Модель предметной области будет реализована в явном ОГРАНИЧЕННОМ КОНТЕКСТЕ: *контексте оптимальных закупок*. Этот ОГРАНИЧЕННЫЙ КОНТЕКСТ однозначно соответствует отдельной ПОДОБЛАСТИ — *смысловому ядру оптимальных закупок*. Благодаря тому, что он однозначно соответствует одной ПОДОБЛАСТИ, а его модель предметной области тщательно продумана, он является одним из лучших ОГРАНИЧЕННЫХ КОНТЕКСТОВ в этой деловой ПРЕДМЕТНОЙ ОБЛАСТИ. Модель предметной области будет реализована в явном ОГРАНИЧЕННОМ КОНТЕКСТЕ: *контексте оптимальных закупок*. Этот ОГРАНИЧЕННЫЙ КОНТЕКСТ однозначно соответствует ПОДОБЛАСТИ под названием *смысловое ядро оптимальных закупок*. Поскольку модель однозначно соответствует одной предметной подобласти и тщательно продумана, она станет одним из лучших ОГРАНИЧЕННЫХ КОНТЕКСТОВ в данной предметной области.

Еще один ОГРАНИЧЕННЫЙ КОНТЕКСТ под названием *контекст закупок* будет разработан для уточнения некоторых технических аспектов процесса закупок и будет играть вспомогательную роль по отношению к *контексту оптимальных закупок*. Эти уточнения не раскрывают никаких специальных знаний об оптимизации закупок. Они должны просто облегчить взаимодействие *контекста оптимальных закупок* с системой ERP в пределах досягаемости. Это просто удобная модель, взаимодействующая с открытым интерфейсом системы ERP. Новый *контекст закупок* и ранее существовавший модуль закупок ERP объединяются в *подобласть (служебную) закупок*.

Модуль закупок ERP представляет собой НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ. Это объясняется тем, что эту ПОДОБЛАСТЬ можно заменить любой коммерческой системой закупок, соответствующей основным потребностям бизнеса. Однако в сочетании с новым *контекстом закупок* в *подобласти закупок* она становится *служебной подобластью*.

Вы не сможете изменить мир плохого проектирования программного обеспечения

На обычном зрелом предприятии возникают нежелательные ситуации, показанные на рис. 2.1 и 2.4. Это значит, что между ПОДОБЛАСТЯМИ и ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ в плохо спроектированном программном обеспечении нет идеального, однозначного соответствия. Вы не сможете изменить мир плохого проектирования программного обеспечения. Вы можете надеяться лишь на правильную реализацию подхода DDD в своих проектах. В результате вы столкнетесь с необходимостью интегрировать свой проект с существующими подобластями, так что приготовьтесь применить технологии, изложенные в первой трети этой главы, и анализировать многочисленные неявные модели в рамках одного продуманного ОГРАНИЧЕННОГО КОНТЕКСТА.

В соответствии с рис. 2.4 *контекст оптимальных закупок* должен также взаимодействовать с *контекстом товарных запасов*, который управляет единицами хранения. Он использует модуль товарных запасов ERP, который находится в пределах *подобласти (служебной) товарных запасов*. Для удобства поставщиков *контекст товарных запасов* может предоставить карты и маршруты к каждому из его складов от источника с помощью внешней картографической службы. С точки зрения *контекста товарных запасов* в картах нет ничего особенного. Можно выбрать одну из существующих картографических служб и использовать преимущества выбранной картографической системы в течение долгого времени. Картографическая служба — это самостоятельная НЕСПЕЦИАЛИЗИРОВАННАЯ ПОДОБЛАСТЬ, но она используется СЛУЖЕБНОЙ ПОДОБЛАСТЬЮ.

Отметим ключевые пункты с точки зрения компании, разрабатывающей *контекст оптимальных закупок*. В пространстве решений картографическая служба не является частью *контекста товарных запасов*, несмотря на то что в пространстве задач она рассматривается как часть *подобласти товарных запасов*. Даже если картографические услуги предоставлены простым компонентно-ориентированным API, в пространстве решений картографическая служба находится в другом ОГРАНИЧЕННОМ КОНТЕКСТЕ. ЕДИНЫЕ ЯЗЫКИ *подобласти товарных запасов* и *картографической подобласти* не пересекаются. Это значит, что они находятся в разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ. Когда *контекст товарных запасов* использует что-то от внешнего *картографического контекста*, данные могут пройти через по крайней мере минимальный перевод, который будет правильно интерпретирован.

С другой стороны, с точки зрения внешней бизнес-организации, которая разрабатывает и предлагает картографические услуги по подписке, картография — это СМЫСЛОВОЕ ЯДРО. У этой внешней организации есть собственная предметная область, или сфера бизнес-операций. Она должна оставаться конкурентоспособной, постоянно совершенствуя свою модель предметной области, чтобы сохранить подписчиков и привлечь новых. Если бы вы были руководителем картографической организации, то удостоверились бы, что дали клиентам, включая одного рассматриваемого подписчика, все основания пользоваться вашими услугами, а не переходить к конкурентам. Однако это не влияет на точку зрения подписчика, который разрабатывает свою систему управления товарными запасами. Для системы товарных запасов картографическая служба остается НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ. Она могла бы подписаться на услуги другой картографической службы, если бы это давало преимущества.

Заметки на доске

Какие ОГРАНИЧЕННЫЕ КОНТЕКСТЫ есть в вашем пространстве решений? Сейчас вы должны вернуться к доске в поисках хорошей идеи. Однако вас может удивить,

что мы слишком глубоко разбираемся в том, как правильно использовать ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. Так что будьте готовы к возможным усовершенствованиям. Мы же выполняем быструю и гибкую разработку, в конце концов.

Итак, в оставшейся части этой главы мы собираемся рассмотреть важность ОГРАНИЧЕННЫХ КОНТЕКСТОВ как существенного инструмента моделирования пространства решений для DDD. Обсуждение **КАРТ КОНТЕКСТОВ (3)** в основном сосредоточено на вопросе, как работать с отображением разных, но связанных ЕДИНЫХ ЯЗЫКОВ, интегрируя их ОГРАНИЧЕННЫЕ КОНТЕКСТЫ.

Осмысление ограниченных контекстов

Не забывайте, что ОГРАНИЧЕННЫЙ КОНТЕКСТ — это явная граница, внутри которой существует модель предметной области, которая отображает ЕДИНЫЙ ЯЗЫК в модель программного обеспечения. Эта граница проведена потому, что каждая концепция модели, находящаяся внутри вместе со своими свойствами и операциями, имеет особый смысл. Если вы — член команды моделирования, то должны точно знать смысл каждой концепции в вашем КОНТЕКСТЕ.

> **ОГРАНИЧЕННЫЙ КОНТЕКСТ как явная лингвистическая граница**
>
> ОГРАНИЧЕННЫЙ КОНТЕКСТ — это явная граница, внутри которой существует модель предметной области. Внутри этой границы все термины и фразы ЕДИНОГО ЯЗЫКА имеют специфическое значение, а модель точно отражает ЯЗЫК.

Часто в двух явно разных моделях у объектов с одними и теми же или аналогичными именами есть разные значения. Если явная граница проведена вокруг каждой из этих двух моделей отдельно, значение каждого понятия в каждом КОНТЕКСТЕ является вполне определенным. Таким образом, ОГРАНИЧЕННЫЙ КОНТЕКСТ — это преимущественно *лингвистическая граница*. Вы должны использовать эти аргументы как пробный камень, чтобы определить, правильно ли вы используете ОГРАНИЧЕННЫЕ КОНТЕКСТЫ.

Некоторые проекты попадают в ловушку, связанную с попыткой создать комплексную модель, цель которой состоит в том, чтобы заставить всю организацию договариваться о системе имен, у которых есть только одно глобальное значение. Такое моделирование является ошибкой. Во-первых, будет почти невозможно достичь соглашения между всеми заинтересованными лицами о том, что у всех понятий есть единственное, ясное и уникальное глобальное значение. Некоторые организации настолько крупные и сложные, что вы никогда не сможете собрать всех заинтересованных лиц, не говоря уже о достижении общего соглашения о

смысле терминов между ними. Даже если вы работаете в меньшей компании с относительно немногими заинтересованными лицами, добиться устойчивого определения единственного глобального понятия все еще маловероятно. Таким образом, лучше всего признать, что различия всегда существуют, и применить ОГРАНИЧЕННЫЙ КОНТЕКСТ, чтобы очертить границы каждой модели предметной области, в которой различия явные и хорошо поняты.

ОГРАНИЧЕННЫЙ КОНТЕКСТ не означает обязательного создания отдельного типа проектного артефакта. Это не отдельный компонент, документ или схема.[3] Таким образом, это не JAR- или DLL-файл, но они могут использоваться для развертывания ОГРАНИЧЕННОГО КОНТЕКСТА, как будет описано далее в главе.

Рассмотрим контраст между терминами сводка (account) в *контексте банковских услуг* и сводка в *литературном контексте*, как показано в табл. 2.1.

Таблица 2.1. Различие между значениями термина *сводка*

Контекст	Значение	Пример
Контекст банковских услуг	*Сводка* поддерживает запись о дебиторских и кредиторских транзакциях, отображающих текущее финансовое состояние клиента с точки зрения банка	*Сводка о текущих счетах* или *сводка о сберегательных счетах*
Литературный контекст	*Сводка* — это совокупность литературных выражений об одном или нескольких событиях, произошедших за определенный период времени	На сайте amazon.com продается книга *Into Thin Air: A Personal Account of the Mt. Everest Disaster*

На рис. 2.5 невозможно различить типы *сводок* по именам. Различие между ними можно обнаружить только по названиям их концептуальных контейнеров, т.е. соответствующих ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

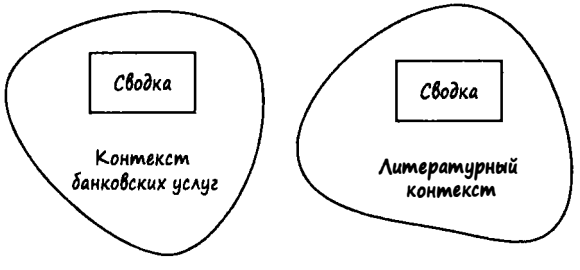


Рис. 2.5. Объекты *сводка* в двух разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ имеют совершенно разный смысл, но обнаружить это можно только по названиям ОГРАНИЧЕННЫХ КОНТЕКСТОВ

[3] Вы можете нарисовать диаграмму одного или нескольких ОГРАНИЧЕННЫХ КОНТЕКСТОВ на КАРТЕ КОНТЕКСТОВ, однако диаграмма — это не ограниченный контекст.

Эти ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, скорее всего, относятся к разным ПРЕДМЕТНЫМ ОБЛАСТЯМ. Цель этого примера — показать, что главную роль играет ОГРАНИЧЕННЫЙ КОНТЕКСТ.

Контекст играет главную роль

Контекст играет главную роль, особенно при реализации подхода DDD.

В сфере финансов часто используют термин *ценная бумага*. Комиссия по ценным бумагам и биржам (Securities and Exchange Commission — SEC) применяет термин *ценная бумага* к акциям. В то же время фьючерсные контракты — это товары и не относятся к юрисдикции Комиссии SEC. Однако некоторые финансовые фирмы называют фьючерсы *ценными бумагами*, но помечают их **СТАНДАРТНЫМ ТИПОМ (STANDARD TYPE) (6)** Фьючерс.

Существует ли оптимальный ЯЗЫК для *фьючерсов*? Это зависит от ПРЕДМЕТНОЙ ОБЛАСТИ, в которой он будет использоваться. Одни могут соглашаться, что такой язык существует, а другие это отрицать. Контекст также зависит от *культурной среды*. В конкретной фирме, торгующей фьючерсами, этот контекст может отлично соответствовать культуре использования термина *ценная бумага* в конкретном ЕДИНОМ ЯЗЫКЕ.

Мы описали очень тонкое различие между значениями одного и того же слова, которое часто возникает на предприятиях. Каждая команда в каждом КОНТЕКСТЕ выбирает имя с учетом ЕДИНОГО ЯЗЫКА. Никто не выбирает имя произвольным образом. Оно должно четко отличаться от терминов в другом КОНТЕКСТЕ. Рассмотрим два *контекста банковских услуг* — контекст сводок о текущих счетах и контекст сводок о сберегательных счетах.[4] Нет никакой необходимости давать имя сводка о текущих счетах объекту в *контексте текущих счетов* или имя сводка о сберегательных счетах — объекту в *контексте сберегательных счетов*. Обе концепции можно совершенно безопасно называть сводками, потому что в каждом из ОГРАНИЧЕННЫХ КОНТЕКСТОВ они имеют немного разные значения. Разумеется, нет никаких правил, запрещающих уточнять эти имена. Каждая команда может самостоятельно принимать такие решения.

Если требуется интеграция, то между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ необходимо установить соответствие. Это может оказаться сложной задачей в рамках подхода DDD и потребовать соответствующего внимания. Обычно экземпляр объекта не используется за пределами его границ, но связанные объекты в разных контекстах могут частично разделять общее состояние.

Рассмотрим еще один пример использования одного и того же имени в разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ, но на этот раз в пределах одной и той же

[4] Это значит, что ПРЕДМЕТНАЯ ОБЛАСТЬ разделена на ОГРАНИЧЕННЫЕ КОНТЕКСТЫ текущих и сберегательных счетов.

ПРЕДМЕТНОЙ ОБЛАСТИ. Рассмотрим вопросы моделирования работы издательства, в котором книги проходят разные этапы жизненного цикла. Грубо говоря, книги в издательстве проходят через разные КОНТЕКСТЫ.

- Разработка концепции и предложения к изданию
- Контакт с автором
- Управление писательской деятельностью и редакционным процессом
- Разработка макета книги, включая иллюстрации
- Перевод книги на другие языки
- Выпуск бумажных и/или электронных версий книги
- Маркетинг
- Продажа книги сбытовикам и/или непосредственно покупателям
- Поставка экземпляров книги сбытовикам или покупателям

Существует ли на каждом этапе единственный способ разработки правильной модели книги? Конечно, нет. На каждом из этих этапов книга имеет разные определения. Пока не заключен контракт, название книги остается неопределенным и может измениться в процессе редактирования. На этапах писательской деятельности и редакционного процесса существует несколько рукописей книги с комментариями и исправлениями, а также чистовой экземпляр. Художники-оформители создают макеты страниц. Производственный отдел использует эти макеты для создания пробных оттисков, "синек" и окончательных печатных форм. Отделу маркетинга не нужны большинство из артефактов литературной и технической редакций, за исключением обложек и аннотаций. Для продажи книги имеют значение лишь ее название, расположение склада, доступное количество экземпляров, размер и вес.

Что произойдет, если мы попытаемся разработать централизованную модель книги, охватывающую все этапы ее жизненного цикла? Возникнет множество недоразумений, разногласий и в результате — противоречивое и никому не нужное программное обеспечение. Даже если время от времени удастся разработать правильную обобщенную модель, она лишь случайно сможет удовлетворить потребности всех клиентов, и то частично.

Для того чтобы противостоять такой нежелательной путанице, издатель, моделирующий с помощью подхода DDD, использовал бы отдельные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ для каждой из стадий жизненного цикла. В каждом из многочисленных ОГРАНИЧЕННЫХ КОНТЕКСТОВ есть тип книги. Различные объекты книги разделили бы идентичность между всеми или большинством КОНТЕКСТОВ, возможно, сначала установленных на этапе осмысления. Однако модель книги в каждом контексте отличалась бы от всех других. Это прекрасно, и фактически так

и должно быть. Когда команда данного ОГРАНИЧЕННОГО КОНТЕКСТА говорит о **книге**, она знает точно, чего хочет для своего КОНТЕКСТА. Организация учитывает естественную потребность в различиях. Нельзя сказать, что таких положительных результатов легко достичь. Тем не менее, используя явные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, удается регулярно поставлять программное обеспечение с постепенно нарастающими улучшениями, которые предназначены для удовлетворения определенных потребностей бизнеса.

Теперь давайте кратко рассмотрим решение, используемое командой SaaSOvation для моделирования (рис. 2.3).

Как указывалось ранее, в ПРЕДМЕТНОЙ ОБЛАСТИ *контекста сотрудничества* эксперты не описывают людей, которые используют средства сотрудничества как **пользователи** с **полномочиями**. Скорее они говорят об этих сотрудниках с точки зрения ролей, которые они играют в КОНТЕКСТЕ, как **авторы**, **владельцы**, **участники** и **модераторы**. Там может существовать немного контактной информации, но, вероятно, это далеко не все. С другой стороны, именно в *контексте идентификации и доступа* мы говорим о **пользователях**. В этом КОНТЕКСТЕ у объектов **пользователя** есть имена клиентов и подробная информация об отдельном человеке, включая конкретные указания, как с ним связаться.

И все же мы не создаем объект **автора** из ничего. Каждый сотрудник должен быть предварительно квалифицирован. Мы подтверждаем существование **пользователя**, играющего соответствующую **роль** в пределах *контекста идентификации и доступа*. Атрибуты дескриптора аутентификации передаются вместе с запросами в *контекст идентификации и доступа*. Для того чтобы создать новый объект сотрудника, например **модератора**, мы используем подмножество атрибутов **пользователя** и имя **роли**. Точные детали того, как мы получаем состояние объекта из отдельного ОГРАНИЧЕННОГО КОНТЕКСТА, не важны (хотя позже все это будет подробно разъясняться). Пока важно то, что эти два разных понятия подобны и в то же самое время отличаются друг от друга и что эти различия определяются ОГРАНИЧЕННЫМ КОНТЕКСТОМ. На рис. 2.6 показано, как **пользователь** и **роль** в их собственном КОНТЕКСТЕ используются для того, чтобы создать **модератора** в другом КОНТЕКСТЕ.

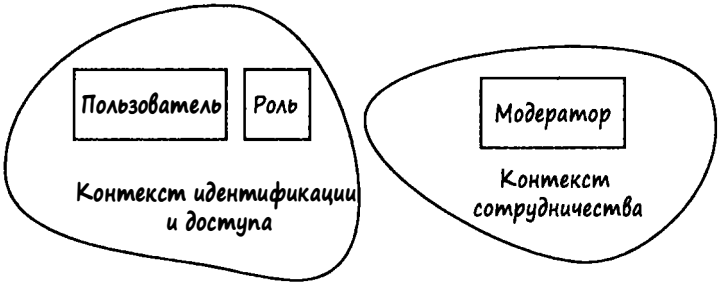


Рис. 2.6. Объект *модератора* в своем КОНТЕКСТЕ основан на атрибутах *пользователя* и *роли* в другом контексте

Заметки на доске

- Выясните, можно ли идентифицировать несколько слегка различающихся концепций, существующих в разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ вашей ПРЕДМЕТНОЙ ОБЛАСТИ.
- Выясните, правильно ли разделены эти концепции или разработчики просто скопировали один и тот же код во всех КОНТЕКСТЫ.

В принципе, определить надлежащее разделение вполне возможно, потому что у подобных объектов есть разные свойства и операции. В этом случае граница правильно отделяет понятия. Однако, если вы видите одни и те же объекты во многих контекстах, это, вероятно, означает, что сделана определенная ошибка моделирования, за исключением ситуаций, когда два ОГРАНИЧЕННЫХ КОНТЕКСТА используют **ОБЩЕЕ ЯДРО (SHARED KERNEL) (3)**.

Пространство не только для модели

ОГРАНИЧЕННЫЙ КОНТЕКСТ не обязательно охватывает только модель ПРЕДМЕТНОЙ ОБЛАСТИ. Правда, модель — основной компонент этого концептуального контейнера. Однако ОГРАНИЧЕННЫЙ КОНТЕКСТ содержит не только модель. Он часто охватывает систему, приложение, или бизнес-службу.[5] Иногда ОГРАНИЧЕННЫЙ КОНТЕКСТ содержит меньше элементов, если, например, НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ можно выделить с помощью модели предметной области. Рассмотрим элементы системы, которые обычно являются частями ОГРАНИЧЕННОГО КОНТЕКСТА.

Когда модель приводит к созданию схемы базы данных постоянного хранения, эта схема оказывается внутри границы. Это происходит потому, что данная схема спроектирована, разработана и поддерживается командой моделирования. Это значит, что имена таблиц базы данных и имена столбцов, например, непосредственно отражают имена, используемые в модели, а не имена, переведенные в другой стиль. Например, допустим, что у нашей модели есть класс под названием `BacklogItem`, и у этого класса есть именованные свойства ОБЪЕКТА-ЗНАЧЕНИЯ с именами `backlogItemId` и `businessPriority`.

[5] По общему признанию, значения терминов *система*, *приложение* и *бизнес-служба* не всегда согласуются. Однако в общем смысле я использую их, чтобы обозначить сложный набор компонентов, которые взаимодействуют друг с другом, чтобы описать ряд важных бизнес-сценариев.

```
public class BacklogItem extends Entity {
    ...
    private BacklogItemId backlogItemId;
    private BusinessPriority businessPriority;
    ...
}
```

Можно было бы ожидать, что эти атрибуты будут отображены в базу данных следующим образом.

```
CREATE TABLE 'tbl_backlog_item' (
    ...
    'backlog_item_id_id' varchar(36) NOT NULL,
    'business_priority_ratings_benefit' int NOT NULL,
    'business_priority_ratings_cost' int NOT NULL,
    'business_priority_ratings_penalty' int NOT NULL,
    'business_priority_ratings_risk' int NOT NULL,
    ...
) ENGINE=InnoDB;
```

С другой стороны, если схема базы данных существовала ранее или если другая команда средств моделирования данных нарушила проект схемы базы данных, то такая схема не относится к ОГРАНИЧЕННОМУ КОНТЕКСТУ, заполненному моделью предметной области.

Если существует **ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС (USER INTERFACE) (14)** представления, выражающий модель и управляющий реализацией ее поведения, то он также существует в ОГРАНИЧЕННОМ КОНТЕКСТЕ. Однако это не означает, что мы моделируем ПРЕДМЕТНУЮ ОБЛАСТЬ с помощью пользовательского интерфейса, вызывая анемию модели предметной области. Мы хотим отказаться от **АНТИШАБЛОНА ИНТЕЛЛЕКТУАЛЬНОГО ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (SMART UI ANTI-PATTERN)** и любого искушения перетащить в другие области системы понятия предметной области, которые принадлежат модели.

Пользователями системы или приложения не всегда бывают люди. Иногда пользователями могут быть компьютерные системы. Например, могут существовать такие компоненты, как веб-службы. Для взаимодействия с такой моделью, как **СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ (OPEN HOST SERVICE) (3, 13)**, мы можем использовать ресурсы RESTful. Вместо нее можно также развернуть протокол Simple Object Access Protocol (SOAP) или конечные точки службы сообщений. Во всех этих случаях внутри границы оказываются сервис-ориентированные компоненты.

И компоненты пользовательского интерфейса, и сервис-ориентированные конечные точки делегируются **ПРИКЛАДНЫМ СЛУЖБАМ (APPLICATION SERVICES) (14)**. Это различные виды служб, обычно обеспечивающие управление безопасностью и транзакциями, и действующие как **ФАСАД (FACADE)** [Гамма и др.] по отношению к модели. Они действуют как диспетчеры задач,

преобразовывая запросы сценария использования в выполнение операций логики предметной области. ПРИКЛАДНЫЕ СЛУЖБЫ также находятся внутри границы.

Кое-что об архитектурных и прикладных аспектах

Если вы хотите узнать, как подход DDD согласуется с разными архитектурными стилями, читайте главу, посвященную шаблону **АРХИТЕКТУРА (ARCHITECTURE) (4)**. Кроме того, **ПРИКЛАДНЫЕ СЛУЖБЫ** специально рассматриваются в главе, посвященной шаблону **ПРИЛОЖЕНИЕ (APPLICATION) (14)**. В обеих главах есть полезные диаграммы и фрагменты программ.

ОГРАНИЧЕННЫЙ КОНТЕКСТ в основном инкапсулирует ЕДИНЫЙ ЯЗЫК и его динамическую модель, но он включает в себя также то, что предназначено для обеспечения взаимодействия и поддержки модели предметной области. Позаботьтесь также о том, чтобы каждый архитектурный аспект находился на своем месте.

Заметки на доске

- Посмотрите на каждый из ОГРАНИЧЕННЫХ КОНТЕКСТОВ, изображенных на вашей доске. Можете ли вы представить себе другие компоненты, кроме модели предметной области, внутри границ КОНТЕКСТОВ?
- Если существуют пользовательский интерфейс и набор ПРИКЛАДНЫХ СЛУЖБ, убедитесь, что они находятся внутри границ контекста. (Для этого следует проявить гибкость. Некоторые идеи о представлении различных компонентов показаны на рис. 2.8–2.10.)
- Если в рамках вашей модели была разработана схема базы данных или другого постоянного хранилища, убедитесь, что они находятся внутри границы. (Способ представления схемы базы данных показан на рис. 2.8–2.10.)

Размер ограниченных контекстов

Сколько **МОДУЛЕЙ (MODULES) (9)**, **АГРЕГАТОВ (AGGREGATES) (10)**, **СОБЫТИЙ (EVENTS) (8)** и **СЛУЖБ (SERVICES) (7)** — основных структурных элементов модели предметной области, созданной с помощью подхода DDD — должно содержаться в ОГРАНИЧЕННОМ КОНТЕКСТЕ? Это все равно что спросить "Какой должна быть длина фрагмента строки?" ОГРАНИЧЕННЫЙ КОНТЕКСТ должен быть настолько большим, насколько это необходимо для полноценного выражения законченного ЕДИНОГО ЯЗЫКА.

Внешние понятия, которые не являются действительной частью СМЫСЛОВОГО ЯДРА, должны быть отброшены. Если понятие не выражается с помощью ЕДИНОГО ЯЗЫКА, то оно не должно быть представлено в вашей модели. Если же одно

или несколько внешних понятий все же закрались в модель, то от них следует избавиться. Вероятно, они принадлежат отдельной СЛУЖЕБНОЙ или НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТИ, или никакой модели вообще.

Будьте осторожны, чтобы по ошибке не отбросить понятия, которые действительно принадлежат ПРЕДМЕТНОЙ ОБЛАСТИ. Ваша модель должна полностью продемонстрировать богатство ЕДИНОГО ЯЗЫКА в КОНТЕКСТЕ, не пропустив ничего существенного. Необходим четкий, хороший критерий. Помочь принять правильное решение вашей команде помогут такие инструменты, как **КАРТЫ КОНТЕКСТОВ (3)**.

В фильме *"Амадей"* (*Amadeus*)[6] есть сцена, в которой австрийский император Иосиф II говорит Моцарту, что его музыка прекрасна, но "содержит слишком много нот". Моцарт мгновенно ответил императору: "Ровно столько нот, сколько необходимо". Этот ответ хорошо иллюстрирует подход к выбору контекстных границ в наших моделях. В модели существует столько концепций предметной области, сколько необходимо для моделирования в пределах ОГРАНИЧЕННОГО КОНТЕКСТА, ни больше, ни меньше.

Конечно, нам редко удается достичь легкости, с которой Моцарт писал симфонии. Всегда существует опасность, что мы пропустили возможность уточнить модель предметной области. На каждой итерации мы проверяем свои предположения о модели, что вынуждает нас добавлять или удалять то или иное понятие или изменять способ, которым эти понятия ведут себя и сотрудничают друг с другом. Но дело в том, что мы *оказываемся перед этой проблемой снова и снова* и, используя принципы DDD, *уделяем серьезное внимание вопросу, что чему принадлежит и чему не принадлежит*. Мы используем ОГРАНИЧЕННЫЙ КОНТЕКСТ и инструменты, такие как КАРТЫ КОНТЕКСТОВ, чтобы анализировать действительные части СМЫСЛОВОГО ЯДРА. Мы не используем произвольные правила сегрегации, не относящиеся к подходу DDD.

Прекрасное звучание моделей предметной области

Если бы наши модели были музыкальными произведениями, то они должны были бы звучать чисто и мощно, а возможно, даже элегантно и прекрасно.

Если данный ОГРАНИЧЕННЫЙ КОНТЕКСТ имеет слишком жесткие границы, то в нем возникнут зияющие дыры из-за отсутствия жизненно важных контекстных понятий. Если мы продолжим включать в модель новые и новые понятия, которые не выражают смысловое ядро решаемой бизнес-проблемы, то замутим воду так, что ничего не увидим и не поймем, что важно, а что нет. Какова наша цель? Если бы наши модели были музыкой, то хотелось бы, чтобы у них был безошибочный полный, чистый и, возможно, даже элегантный и красивый звук. Количества

[6] Orion Pictures, Warner Brothers, 1984.

нот — МОДУЛИ, АГРЕГАТЫ, СОБЫТИЯ и СЛУЖБЫ внутри контекста — не было бы ни больше, ни меньше, чем требует проект. Люди, "слушающие" нашу модель, никогда не стали бы спрашивать, что за странный "звук" слышится в середине гармоничной симфонии. При этом их не отвлекали бы моменты полной тишины, вызванной недостающей страницей музыкальных нот.

Что могло бы привести к созданию ОГРАНИЧЕННОГО КОНТЕКСТА, имеющего неправильный размер? Мы могли по ошибке поддаться архитектурным влияниям, а не руководствоваться ЕДИНЫМ ЯЗЫКОМ. Возможно, способ, которым обычно используется платформа, каркас или другая инфраструктура для упаковки и развертывания компонентов, мог некорректно повлиять на образ наших мыслей об ОГРАНИЧЕННЫХ КОНТЕКСТАХ, заставив нас рассматривать их как технические, а не лингвистические границы.

Другая возможная ловушка состоит в разделении ОГРАНИЧЕННЫХ КОНТЕКСТОВ для распределения задач в соответствии с доступными ресурсами разработчика. Технические руководители и менеджеры проекта могли бы подумать, что разработчикам проще управлять меньшими задачами. Несмотря на то что это вполне возможно, установление границ с целью распределения задач является фиктивным и соответствует лингвистическим мотивациям контекстного моделирования. Фактически нет никакой потребности накладывать фиктивные границы, чтобы управлять техническими ресурсами.

Важный вопрос: "Как ЕДИНЫЙ ЯЗЫК экспертов в предметной области задает реальные границы контекстов?"

Если сформирован фиктивный КОНТЕКСТ, чтобы учесть архитектурный компонент или ресурсы разработчика, то ЯЗЫК становится фрагментированным и теряет выразительность. Следовательно, необходимо сосредоточиться на СМЫСЛОВОМ ЯДРЕ с понятиями, которые естественно совмещаются в единственном ОГРАНИЧЕННОМ КОНТЕКСТЕ согласно ЯЗЫКУ, на котором говорят эксперты в проблемной области. Сделав это, вы сможете идентифицировать компоненты, которые естественно включаются в единую, связную модель. Сохраните все такие компоненты в ОГРАНИЧЕННОМ КОНТЕКСТЕ.

Иногда создания миниатюрных ОГРАНИЧЕННЫХ КОНТЕКСТОВ можно избежать с помощью тщательного выбора МОДУЛЕЙ. Анализируя службы, действующие в нескольких "ОГРАНИЧЕННЫХ КОНТЕКСТАХ", вы обнаружите, что разумное использование МОДУЛЕЙ могло бы уменьшить общее количество фактических ОГРАНИЧЕННЫХ КОНТЕКСТОВ до одного. Модули могут также использоваться для разделения обязанностей разработчика, управляя распределением задачи и используя более точный тактический подход.

Заметки на доске

- Нарисуйте ОГРАНИЧЕННЫЙ КОНТЕКСТ вашей текущей модели в виде большой выпуклой фигуры неправильной формы.

Даже если у вас еще нет явной модели, продолжайте думать о ЕДИНОМ ЯЗЫКЕ.

- Внутри выпуклой фигуры запишите имена основных концепций, которые должна реализовать ваша программа. Выясните, нет ли лишних или пропущенных концепций. Как решить эти проблемы?

Осторожно используйте лингвистические драйверы DDD

Важное обстоятельство: если вы не подчиняетесь лингвистическим драйверам, значит, вы не работаете с экспертами в предметной области и не слушаете их при создании ОГРАНИЧЕННОГО КОНТЕКСТА. Тщательно обдумайте размер ваших ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Не спешите уменьшать их размеры.

Согласования с техническими компонентами

Не вредно подумать об ОГРАНИЧЕННОМ КОНТЕКСТЕ с точки зрения технических компонентов, которые его содержат. Просто имейте в виду, что технические компоненты не определяют КОНТЕКСТ. Давайте рассмотрим некоторые распространенные способы их составления и развертывания.

В интегрированных средах разработки, таких как Eclipse или Idea IntelliJ, ОГРАНИЧЕННЫЙ КОНТЕКСТ часто заключается в единственном проекте. Используя среды Visual Studio и платформы .NET, вы можете разделить свой пользовательский интерфейс, ПРИКЛАДНЫЕ СЛУЖБЫ и модель предметной области на отдельные проекты в рамках одного и того же решения или выбрать другое разделение. Исходное дерево проекта может быть ограничено самой моделью предметной области или может содержаться в окружающих ее областях в рамках **МНОГОУРОВНЕВОЙ (LAYERS) (4)** или **ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ (HEXAGONAL) (4)**. Это обеспечивает большую гибкость. На языке Java высокоуровневый пакет обычно определяет имя МОДУЛЯ высшего уровня для ОГРАНИЧЕННОГО КОНТЕКСТА. С помощью одного из предыдущих примеров это можно сделать примерно так.

```
com.mycompany.optimalpurchasing
```

Дерево исходного кода в этом ОГРАНИЧЕННОМ КОНТЕКСТЕ можно разделить далее в соответствии с архитектурными аспектами. Ниже показано, как могут называться пакеты второго уровня.

```
com.mycompany.optimalpurchasing.presentation
com.mycompany.optimalpurchasing.application
com.mycompany.optimalpurchasing.domain.model
com.mycompany.optimalpurchasing.infrastructure
```

Даже при таком модулярном разделении в одном ОГРАНИЧЕННОМ КОНТЕКСТЕ должна работать только одна команда.

Одна команда в одном ОГРАНИЧЕННОМ КОНТЕКСТЕ

Назначение единственной команды для работы в единственном ОГРАНИЧЕННОМ КОНТЕКСТЕ не является попыткой ограничить гибкость командной организации. Это не означает, что команды не могут быть реорганизованы по мере необходимости или что отдельные члены одной команды не могут работать в других проектах. Компания должна использовать людей так, как ей наиболее удобно. Данный принцип просто утверждает, что одной четко определенной, сплоченной команде экспертов в предметной области и разработчиков лучше сосредоточиться на одном ЕДИНОМ ЯЗЫКЕ, смоделированном в явном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Если вы назначите две или больше разных команд в один и тот же ОГРАНИЧЕННЫЙ КОНТЕКСТ, то каждая команда внесет свой вклад в создание противоречивого и неточного ЕДИНОГО ЯЗЫКА.

Возможно также, что две команды будут сотрудничать при разработке ОБЩЕГО ЯДРА, который фактически не является типичным ОГРАНИЧЕННЫМ КОНТЕКСТОМ. Шаблон КАРТА КОНТЕКСТОВ формирует тесные отношения между двумя командами и требует длительных консультаций о необходимости модели. Этот подход к моделированию менее распространен и обычно по возможности избегается.

При использовании языка Java мы можем технически хранить ОГРАНИЧЕННЫЙ КОНТЕКСТ в одном или более файлах JAR, включая файлы WAR или EAR. На это может влиять желание обеспечить модульность. Слабо связанные части модели предметной области можно было бы разместить в отдельных файлах JAR, позволяя разворачивать их независимо от версий. Это было бы особенно полезно при работе с большими моделями. Создание многочисленных файлов JAR для единственной модели обеспечило бы преимущество контроля версий ее элементов с помощью пакетов OSGi или модулей Java 8 Jigsaw. Таким образом, различными высокоуровневыми модулями, их версиями и их зависимостями можно было бы управлять как пакетами/модулями. Есть по крайней мере четыре таких пакета/модуля, представленных предыдущими МОДУЛЯМИ второго уровня, основанными на принципах DDD. Впрочем, их может быть и больше.

Для естественного ОГРАНИЧЕННОГО КОНТЕКСТА в среде Windows, например платформы .NET, развертывание можно было бы осуществить с помощью отдельных сборок в файлах DLL. С точки зрения процесса развертывания, файлы DLL

можно считать аналогами файлов JAR, описанных выше. Эту модель можно было бы разделить для развертывания похожими способами. Обеспечение модульности общеязыковой исполняющей среды (CLR) осуществляется на основе сборок. Определенная версия сборки и версии зависимых сборок записываются в манифест сборки. См. [MSDN Assemblies].

Примеры контекстов

Поскольку указанные примеры представляют среды разработки "с нуля", между тремя выбранными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ и соответствующими ПОДОБЛАСТЯМИ в конечном счете устанавливается желательное однозначное соответствие. Нашей команде не удалось установить это соответствие с самого начала, что стало для нее важным уроком. Окончательный результат показан в рис. 2.7.

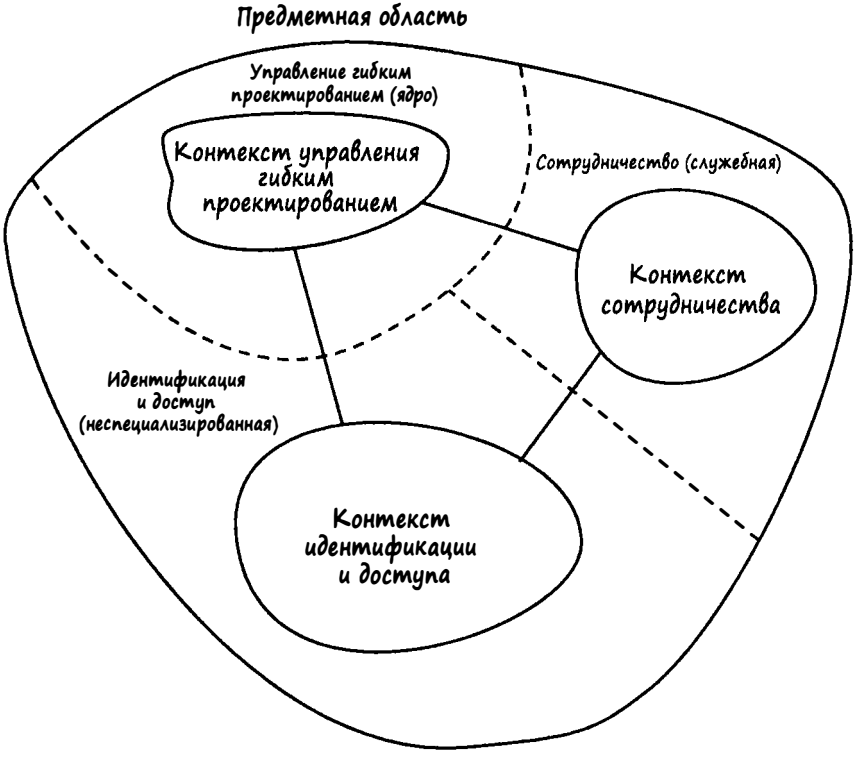


Рис. 2.7. Примерный вид ОГРАНИЧЕННЫХ КОНТЕКСТОВ из примеров в полностью согласованных ПОДОБЛАСТЯХ

Следующий материал демонстрирует, как эти три модели формируют реалистическое, современное решение для предприятия. В любом реальном проекте всегда существуют многочисленные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. Их интеграция — важный сценарий на современном предприятии. В дополнение к ОГРАНИЧЕННОМУ КОНТЕКСТУ и ПОДОБЛАСТЯМ мы должны также освоить КАРТЫ КОНТЕКСТОВ с **ИНТЕГРАЦИЕЙ (INTEGRATION) (13)**.

Рассмотрим три ОГРАНИЧЕННЫХ КОНТЕКСТА как пример реализации DDD[7]: *контекст сотрудничества, контекст идентификации и доступа* и *контекст управления гибким проектированием*.

Контекст сотрудничества

Инструменты делового сотрудничества — одна из самых важных областей для создания и усовершенствования синергетического рабочего места в быстро изменяющейся экономике. Все, что может способствовать повышению производительности, передаче знаний, совместному использованию идей и гибкому управлению творческим процессом, идет во благо корпорации. Независимо от того, кому именно программные инструменты предлагают полезные функции, предназначенные для ежедневной работы — широким массам или узким сообществам, — корпоративный опыт сосредоточивается в лучших онлайновых инструментах, и компания SaaSOvation хочет получить долю этого рынка.

Рабочая группа, перед которой поставили задачу разработать и реализовать *контекст сотрудничества*, получила разрешение на выпуск первой версии, поддерживающей следующий минимальный комплект инструментов: форумы, совместно используемые календари, блоги, мгновенный обмен сообщениями, Wiki, доски объявлений, управление документооборотом, объявления и предупреждения, отслеживание действий и каналы RSS. Помимо поддержки широкого спектра функций, каждый из отдельных инструментов сотрудничества в комплекте может также поддерживать специализированные, узкие среды, оставаясь в одном и том же ОГРАНИЧЕННОМ КОНТЕКСТЕ, потому что все они — часть сотрудничества. К сожалению, в этой книге невозможно описать весь комплект сотрудничества. Однако мы исследуем части модели предметной области для инструментов, представленных в рис. 2.8, а именно — для форумов и совместно используемых календарей.

[7] Карты контекстов содержат больше информации о трех ОГРАНИЧЕННЫХ КОНТЕКСТАХ, в частности о том, как они связаны друг с другом и как они интегрируются. Таким образом, основная информация сосредоточена в СМЫСЛОВОМ ЯДРЕ.

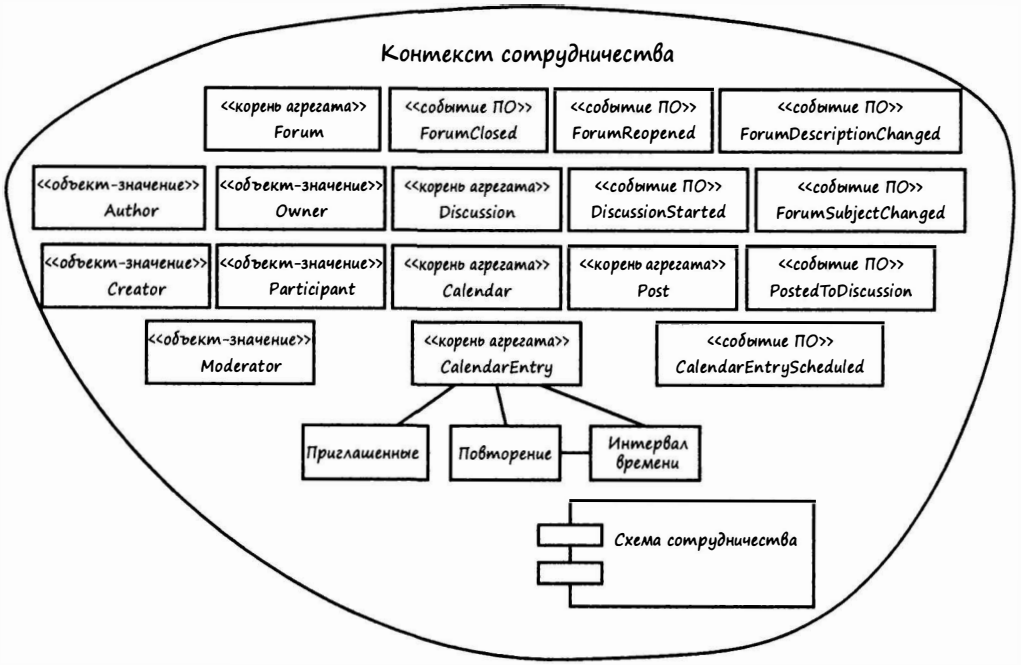


Рис. 2.8. *Контекст сотрудничества.* Его ЕДИНЫЙ ЯЗЫК определяет то, что находится в его границах. Для удобства представления некоторые элементы модели не показаны, в частности пользовательский интерфейс и компоненты ПРИКЛАДНОЙ СЛУЖБЫ

Итак, вернемся к нашей команде...

Команда с самого начала применяла тактический подход DDD, но все еще не освоила некоторые его тонкости. Фактически она использовала так называемый облегченный DDD, используя тактические шаблоны главным образом для решения технических задач. Несомненно, члены команды пытались выработать ЕДИНЫЙ ЯЗЫК сотрудничества, но не понимали, что у модели были четкие границы, которые не могут быть расширены слишком далеко. В результате они сделали ошибку, включив в модель сотрудничества систему безопасности и полномочий. Команда внимательно проанализировала проект и поняла, что разработка системы безопасности и полномочий как части их модели не так желательна, как она когда-то думала.

Сначала члены команды не были слишком обеспокоены или не полностью осознавали опасность создания смеси мелких приложений. Все же, если бы они не

использовали централизованный провайдер безопасности, это могло бы произойти. Они смешали две модели в одну. Достаточно скоро они поняли, что путаница, которая последовала из смешения проблем безопасности с их СМЫСЛОВЫМ ЯДРОМ, имела неприятные последствия. Прямо в середине логики основного бизнеса, в поведенческих методах, разработчики должны были выполнять проверку клиентских полномочий на выполнение запроса.

```
public class Forum extends Entity {
    ...
    public Discussion startDiscussion(
           String aUsername, String aSubject) {
       if (this.isClosed()) {
            throw new IllegalStateException("Forum is closed.");
       }

       User user = userRepository.userFor(this.tenantId(), aUsername);

       if (!user.hasPermissionTo(Permission.Forum.StartDiscussion)) {
            throw new IllegalStateException(
              "User may not start forum discussion.");
       }

       String authorUser = user.username();
       String authorName = user.person().name().asFormattedName();
       String authorEmailAddress = user.person().emailAddress();

       Discussion discussion = new Discussion(
           this.tenant(), this.forumId(),
           DomainRegistry.discussionRepository().nextIdentity(),
           authorUser, authorName, authorEmailAddress,
           aSubject);

           return discussion;
    }
    ...
}
```

Это "столкновение поездов"?

Одни разработчики называют сцепление нескольких выражений в одну строку, например `user.person().name().asFormattedName()`, "столкновением поездов" ("train wreck"). Другие считают, что это повышает выразительность кода. Я не рассматриваю ни одну из этих точек зрения. В центре моего внимания находится смешанная модель. "Столкновение поездов" — это совершенно другой вопрос.

Это был действительно плохой проект. Разработчики не должны были вообще ссылаться на объект `User` здесь, оставив в покое запросы, посылаемые в **ХРАНИЛИЩЕ (REPOSITORY) (12)**. Даже объект `Permission` должен был оставаться вне досягаемости. Это стало возможным, потому что эти объекты были неправильно разработаны как часть модели сотрудничества. Что еще хуже, это искажение

привело к тому, что команда пропустила понятие, которое она должна была смоделировать, а именно Author. Вместо того чтобы собрать три связанных атрибута в явный ОБЪЕКТ-ЗНАЧЕНИЕ, разработчики, казалось, удовлетворились работой с элементами данных по отдельности. У них на уме была безопасность, а не сотрудничество.

Это не было единичным случаем. У каждого объекта сотрудничества были подобные проблемы. Поскольку риск создания БОЛЬШОГО КОМКА ГРЯЗИ становился неизбежным, команда решила, что код следует изменить. Кроме того, команда также хотела отказаться от подхода к обеспечению безопасности на основе полномочий и использовать вместо него управление доступом, основанное на ролях. Что они должны были сделать?

Будучи пользователями методологий быстрого проектирования и потенциальными разработчиками гибких инструментов управления проектами, они не боялись потратить усилия на рефакторинг. Итак, можно было бы выполнить несколько итераций рефакторинга. Тем не менее оставался вопрос "Нет помогут ли лучшие шаблоны DDD вытащить их из глубокой трясины неуместного кода?"

Поработав сверхурочно и детально изучив тактические шаблоны структурных элементов по книге [Эванс], некоторые члены команды поняли, что эти шаблоны не помогут. Они следовали рекомендациям по созданию АГРЕГАТОВ, механически составляя ОБЪЕКТЫ и ОБЪЕКТЫ-ЗНАЧЕНИЯ. Кроме того, они использовали ХРАНИЛИЩА и **СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICES) (7)**. Тем не менее они пропускали что-то важное, и, возможно, это указывало на необходимость обратить более пристальное внимание на вторую половину книги Эванса.

Наконец они обнаружили некоторые перспективные методы. Поскольку они детально изучили "Часть III: углубляющий рефакторинг" книги Эванса, стало очевидно, что подход DDD предлагает больше, чем они когда-то думали. Используя методы, описанные в этой части книги Эванса, они могли улучшить свою текущую модель, обращая более пристальное внимание на ЕДИНЫЙ ЯЗЫК. С пользой проводя больше времени с экспертами в предметной области, члены команды смогли создать модель, которая более точно соответствовала их ментальной модели. Но это все еще не выводило проект из болота, заполненного аспектами безопасности, которые исказили их видение чистой модели предметной области сотрудничества.

Далее в книге Эванса следует "Часть IV: стратегическое проектирование" [Эванс]. Один из членов команды нашел решающую рекомендацию, которая в конечном счете привела их к реализации СМЫСЛОВОГО ЯДРА. Одним из первых новых инструментов стали КАРТЫ КОНТЕКСТОВ, которые улучшили понимание текущего состояния проекта. Несмотря на простоту, рисование первой КАРТЫ КОНТЕКСТОВ и обсуждение затруднений стало большим шагом вперед. Это позволило провести продуктивный анализ и в конечном счете разблокировать работу команды.

Теперь у команды оказалось несколько возможностей внести временные усовершенствования, которые позволят стабилизировать становящуюся все более ненадежной модель.

1. Они могли осуществить рефакторинг модели, введя **УРОВНИ ОТВЕТСТВЕННОСТИ (RESPONSIBILITY LAYERS)** [Эванс], разделив функции безопасности и полномочий, одновременно переместив их на более низкий логический

уровень существующей модели. Но это не было похоже на оптимальный подход. Использование УРОВНЕЙ ОТВЕТСТВЕННОСТИ предназначено для крупномасштабных моделей или для моделей, которые со временем станут крупными. Каждый уровень остается в рамках модели, потому что он является частью СМЫСЛОВОГО ЯДРА, даже при том, что уровни тщательно разделены. С другой стороны, команда столкнулась с неприемлемыми понятиями, которые не принадлежали СМЫСЛОВОМУ ЯДРУ.

2. В качестве альтернативы они могли создать **ВЫДЕЛЕННОЕ ЯДРО (SEGREGATED CORE)** (Эванс). Этого можно было достичь с помощью исчерпывающего поиска всех понятий, связанных с безопасностью и полномочиями в КОНТЕКСТЕ СОТРУДНИЧЕСТВА, переноса компонентов идентификационных данных и доступа в полностью изолированные пакеты в той же модели. Это не привело бы к окончательному созданию абсолютно изолированного ОГРАНИЧЕННОГО КОНТЕКСТА, но приблизило бы команду к этой цели. Казалось, команде было точно известно, что ей нужно, поскольку сам шаблон утверждает: "Создавать ВЫДЕЛЕННОЕ ЯДРО следует тогда, когда у вас есть большой ОГРАНИЧЕННЫЙ КОНТЕКСТ, который критически важен для системы, но существенная часть модели затеняется большой вспомогательной функциональной возможностью". Совершенно очевидно, что такой вспомогательной функциональной возможностью были функции безопасности и полномочий. Команда в конечном счете поняла, что в результате этих усилий возникнет отдельный КОНТЕКСТ ИДЕНТИФИКАЦИИ И ДОСТУПА, который будет служить НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ их КОНТЕКСТА СОТРУДНИЧЕСТВА.

Инициатива создать ВЫДЕЛЕННОЕ ЯДРО была непростой. Могло потребоваться несколько недель незапланированной работы. Но если бы они не предприняли меры по ликвидации последствий и не провели рефакторинг как можно скорее, то они заплатили бы за промедление множеством ошибок и уязвимым кодом, плохо поддающимся изменениям. Руководство поддержало это предложение и решило, что успешное выделение новой бизнес-услуги могло бы когда-нибудь привести к созданию нового продукта SaaS.

Важно, что команда теперь поняла значение ОГРАНИЧЕННЫХ КОНТЕКСТОВ и упорной борьбы за поддержание связности СМЫСЛОВОГО ЯДРА. Используя дополнительные шаблоны стратегического проектирования, они смогли выделить модели, допускающие повторное использование в отдельных ОГРАНИЧЕННЫХ КОНТЕКСТАХ и правильно их интегрировать.

Вероятно, будущий ОГРАНИЧЕННЫЙ КОНТЕКСТ идентификации и доступа отличался бы от встроенного проекта безопасности и полномочий. Разработка, предназначенная для повторного использования, заставила бы команду сосредоточиться на более универсальной модели, которая могла быть использована многими приложениями при необходимости. Специальная команда — отличающийся от нашей команды *контекста сотрудничества*, но включающая в себя несколько ее членов — смогла также предложить различные стратегии реализации. Стратегии включали использование сторонних продуктов и интеграцию для конкретного заказчика, которая стала маловероятной из-за встроенной путаницы, связанной с вопросами безопасности.

Так как разработка ВЫДЕЛЕННОГО ЯДРА оказалось временной мерой, мы не будем останавливаться на ее результатах. Кратко говоря, она свелась к перемещению всех классов, связанных с безопасностью и полномочиями, в выделенные МОДУЛИ и к требованиям, чтобы клиенты ПРИКЛАДНЫХ СЛУЖБ проверяли безопасность и полномочия, используя эти объекты до вызова в СМЫСЛОВОЕ ЯДРО. Это освободило ЯДРО для реализации лишь композиций объектов модели сотрудничества и поведения. О трансляции объектов, связанных с безопасностью и полномочиями, позаботилась ПРИКЛАДНАЯ СЛУЖБА.

```
public class ForumApplicationService ... {
    ...
    @Transactional
    public Discussion startDiscussion(
            String aTenantId, String aUsername,
            String aForumId, String aSubject) {
        Tenant tenant = new Tenant(aTenantId);
        ForumId forumId = new ForumId(aForumId);

        Forum forum = this.forum(tenant, forumId);

        if (forum == null) {
            throw new IllegalStateException("Forum does not exist.");
        }

        Author author =
                this.collaboratorService.authorFrom(
                        tenant,
                        anAuthorId);

        Discussion newDiscussion =
                forum.startDiscussion(
                        this.forumNavigationService(),
                        author,
                        aSubject);

        this.discussionRepository.add(newDiscussion);

        return newDiscussion;
    }
    ...
}
```

Результат для класса `Forum` выглядит примерно так.

```
public class Forum extends Entity {
    ...

    public Discussion startDiscussionFor(
        ForumNavigationService aForumNavigationService,
        Author anAuthor,
        String aSubject) {
        if (this.isClosed()) {
```

```
            throw new IllegalStateException("Forum is closed.");
        }

        Discussion discussion = new Discussion(
            this.tenant(),
            this.forumId(),
            aForumNavigationService.nextDiscussionId(),
            anAuthor,
            aSubject);

        DomainEventPublisher
            .instance()
            .publish(new DiscussionStarted(
                    discussion.tenant(),
                    discussion.forumId(),
                    discussion.discussionId(),
                    discussion.subject()));

        return discussion;
    }
    ...
}
```

Это позволило устранить спутанность классов `User` и `Permission` и сфокусировать модель только на сотрудничестве. Это еще не идеальный результат, но он подготовил команду к будущему рефакторингу, чтобы разделить и интегрировать ОГРАНИЧЕННЫЕ КОНТЕКСТЫ. В результате этого рефакторинга команда *контекста сотрудничества* наконец удалила бы все модули и типы безопасности и полномочий из ОГРАНИЧЕННОГО КОНТЕКСТА и с удовольствием использовала бы новый *контекст идентификации и доступа*. Их конечная цель — обеспечение централизованной и допускающей повторное использование системы безопасности — была теперь в пределах досягаемости.

Правда, команда, возможно, могла бы пойти в другом направлении. Она могла бы создать множество (десять и более) миниатюрных изолированных ОГРАНИЧЕННЫХ КОНТЕКСТОВ, по одному для каждого инструмента сотрудничества (например, форум и календарь как отдельные модели). Что могло увлечь их в этом направлении? Так как большинство инструментов сотрудничества не были связаны с другими, каждый из них мог быть развернут как автономный компонент. Помещая каждый инструмент в отдельный ОГРАНИЧЕННЫЙ КОНТЕКСТ, команда могла создать приблизительно десять естественных модулей развертывания. Правда, создание десяти различных моделей предметной области было ненужным для достижения этих целей развертывания и будет, вероятно, противоречить принципам ЕДИНОГО ЯЗЫКА.

Вместо этого команда сохранила единство модели, но приняла решение создать отдельный файл JAR для каждого инструмента сотрудничества. Используя инструмент Jigsaw, члены команды создали основанный на версиях модуль развертывания для каждого инструмента. Помимо файлов JAR, для естественных подразделений сотрудничества им были также нужны совместно используемые объекты модели, такие как `Tenant`, `Moderator`, `Author`, `Participant` и др. Идя по

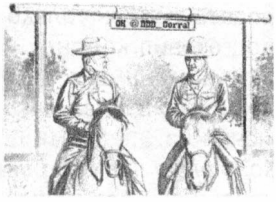
этому пути, команда поддержала разработку унифицированного ЕДИНОГО ЯЗЫКА, одновременно достигая целей развертывания и получая архитектурные и управленческие преимущества.

Понимая это, мы можем понять происхождение *контекста идентификации и доступа*.

Контекст идентификации и доступа

В настоящее время у большинства промышленных приложений должна быть некоторая разновидность компонентов безопасности и полномочий, чтобы гарантировать, что люди, пытающиеся использовать систему, являются подлинными пользователями и имеют право делать то, что они пытаются сделать. Как мы только что выяснили, наивный подход к обеспечению безопасности приложений порождает пользователей и полномочия в каждой дискретной системе и в результате создает "эффект силосной башни"[8] в каждом приложении.

Ковбойская логика



LB: Почему на твоих амбарах и силосных башнях нет замков, но никто не ворует твою кукурузу?

AJ: Мой пес по кличке "Бродяга" управляет доступом. Это мой собственный "эффект силосной башни".

LB: Я не уверен, что ты правильно понял, что написано в книге.

Пользователям одной системы трудно установить связь с пользователями других систем, даже при том, что многие люди используют сразу несколько систем. Для того чтобы "силосные башни" рассыпались по всему деловому ландшафту, архитекторы должны централизовать безопасность и полномочия. Эту задача можно решить, купив или разработав систему управления идентификацией и доступом. Выбранный командой маршрут сильно зависит от степени компетентности ее членов, доступного времени и общей стоимости владения.

[8] "Эффект силосной башни" означает отсутствие общих целей и сотрудничества между подразделениями организации.

Исправление путаницы, связанной с идентификацией и доступом в проекте CollabOvation, можно разделить на несколько этапов. Команда могла бы выполнить рефакторинг, используя ВЫДЕЛЕННОЕ ЯДРО [Эванс]; см. раздел "Контекст сотрудничества". Этот шаг позволил бы достичь цели и в то же время гарантировал бы, что проект CollabOvation будет очищен от аспектов, связанных с безопасностью и полномочиями. Однако команда поняла, что управление идентификацией и доступом должно образовать собственный ОГРАНИЧЕННЫЙ КОНТЕКСТ. Для этого могут потребоваться еще большие усилия.

Итак, возникает новый ОГРАНИЧЕННЫЙ КОНТЕКСТ — *контекст идентификации и доступа*, — который будет использоваться другими ограниченными контекстами с помощью стандартных средств интеграции DDD. С точки зрения контекста потребления *контекст идентификации и доступа* представляет собой НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ. Этот продукт будет назван IdOvation.

Как показано на рис. 2.9, *контекст идентификации и доступа* обеспечивает поддержку многоарендных подписчиков. При разработке продукта SaaS это само собой разумеется. У каждого арендатора и каждого объекта, принадлежавшего данному арендатору, были бы абсолютно уникальные идентификационные данные, логически изолирующие каждого арендатора от всех других. Пользователи систем регистрируются путем самообслуживания только по приглашению. Защищенный доступ обрабатывается службой аутентификации и пароли всегда сильно шифруются. Группы пользователей и вложенные группы позволяют осуществлять сложное управление идентификационными данными как во всей организации, так и в самой маленькой из команд. Доступ к системным ресурсам управляется посредством простых, изящных и в то же время мощных полномочий, основанных на ролях.

Кроме того, когда поведение модели приводит к изменениям состояния, представляющим интерес для наблюдателей, публикуются **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS) (8)**. Эти события обычно моделируются существительными в сочетании с глаголами в прошедшем времени, например `TenantProvisioned`, `UserPasswordChanged`, `PersonNameChanged` и т.п.

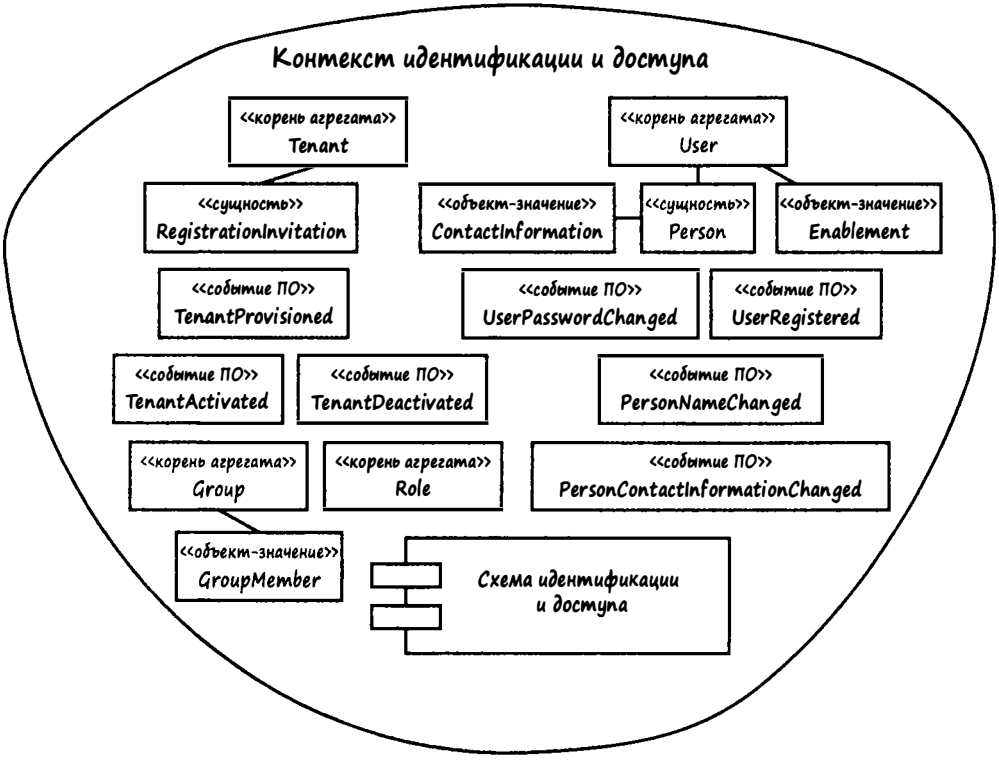


Рис. 2.9. *Контекст идентификации и доступа.* Его ЕДИНЫЙ ЯЗЫК определяет то, что находится в его границах. Для удобства представления некоторые элементы модели не показаны, в частности пользовательский интерфейс и компоненты ПРИКЛАДНОЙ СЛУЖБЫ

В следующей главе показано, как *контекст идентификации и доступа* используется двумя другими КОНТЕКСТАМИ с помощью DDD-шаблонов интеграции.

Контекст управления гибким проектированием

Облегченные методы гибкой разработки способствовали ее популярности, особенно после создания документа Agile Manifesto в 2001 году. В своем заявлении о намерениях компания SaaSOvation назвала в качестве второй стратегической инициативы разработку системы управления гибким проектированием. Вот как развивались события.

Успешно разместив три четверти заказов на проект CollabOvation, выполнив запланированные обновления с поступательным улучшением на основе обратной связи с потребителями и получив прибыль выше ожидаемой, компания приступила к реализации плана по выпуску продукта ProjectOvation. Этот проект имеет свое СМЫСЛОВОЕ ЯДРО и в него были включены главные разработчики из проекта CollabOvation, чтобы они использовали свой опыт разработки многоарендного продукта SaaS и знания DDD.

Этот инструмент фокусируется на управлении быстрым и гибким проектированием, используя методологию Scrum в качестве итеративной и инкрементной основы. Команда ProjectOvation придерживается традиционной модели управления проектами Scrum, дополненной продуктом, владельцем продукта, командой, задачами, списком заданий, запланированными выпусками и спринтами. Оценка задач обеспечивается калькуляторами бизнес-ценности, использующими анализ эффективности затрат.

Бизнес-план преследовал две цели. Продукты CollabOvation и ProjectOvation не должны развиваться изолированно друг от друга. Компания SaaSOvation и ее штат советников придумали новшество, касающееся разработки инструментов сотрудничества на принципах гибкой разработки программного обеспечения. В соответствии с этим новшеством функции продукта CollabOvation будут предлагаться как дополнения к продукту ProjectOvation. Без сомнения, предоставление инструментов для коллективного планирования проектов, обсуждения функциональных возможностей и сценариев, а также проведения дискуссий между командами и внутри команд окажется популярной опцией. Существует также план включить в будущем в продукт ProjectOvation систему планирования корпоративных ресурсов, но сначала следует достичь целей, поставленных перед системой гибкого управления проектами.

Разработчики сначала планировали проектировать функциональные возможности системы ProjectOvation как расширение модели CollabOvation, используя подсистему управления версиями исходного кода. На самом деле это было огромной ошибкой, хотя и типичной для специалистов, игнорирующих ПОДОБЛАСТИ своего пространства задач и ОГРАНИЧЕННЫЕ КОНТЕКСТЫ в пространстве решений.

К счастью, технический персонал извлек уроки из проблем, возникших при разработке *контекста сотрудничества.* Урок, который они усвоили, убедил их, что даже несколько шагов в направлении объединения системы управления гибким проектированием и модели сотрудничества были бы большой ошибкой. Теперь команда приступила к внимательному изучению стратегических шаблонов DDD.

На рис. 2.10 показано, что, приняв *стратегическую точку зрения на проект*, команда ProjectOvation рассматривает своих клиентов как владельцев продукта (product owners) и членов команды (team members). Кроме того, в проекте существуют роли, связанные с использованием системы Scrum. Управление пользователями и ролями осуществляется внутри отдельного *контекста идентификации и доступа.* Используя данный ОГРАНИЧЕННЫЙ КОНТЕКСТ, система самообслуживания позволяет

подписчикам управлять своими идентификационными данными. Компоненты административного управления позволяют менеджерам, например владельцам продукта, указывать членов своей команды, использующих продукт. Правильно управляя ролями, можно создавать владельцев продукта и членов команды в *контексте управления гибким проектированием*. Остальные части проекта будут разрабатываться с помощью реализации концепций ЕДИНОГО ЯЗЫКА управления гибким проектированием в виде тщательно разработанной модели предметной области.

Одно из требований гласит, что проект ProjectOvation должен действовать в виде ряда автономных прикладных служб. Команда желает ограничить зависимость проекта ProjectOvation от других ОГРАНИЧЕННЫХ КОНТЕКСТОВ разумной периодичностью или по крайней мере практической целесообразностью. Вообще говоря, система ProjectOvation может функционировать самостоятельно и, если бы система IdOvation или CollabOvation по каким-то причинам была отключена, система ProjectOvation продолжила бы работать автономно. Конечно, в этом случае некоторые компоненты на некоторое очень короткое время могли бы потерять синхронизацию, но система продолжала бы работу.

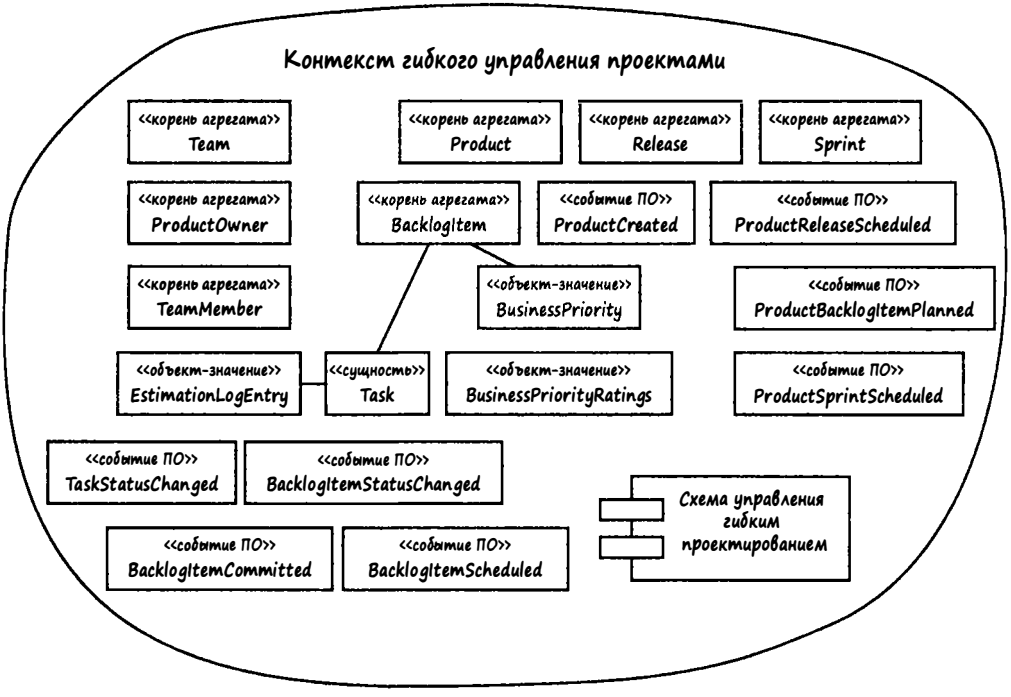


Рис. 2.10. *Контекст гибкого управления проектами.* Его ЕДИНЫЙ ЯЗЫК определяет то, что находится в его границах. Для удобства представления некоторые элементы модели не показаны, в частности пользовательский интерфейс и компоненты ПРИКЛАДНОЙ СЛУЖБЫ

Контекст дает каждой команде очень конкретный смысл

Класс `Product`, основанный на принципах Scrum, содержит произвольное количество экземпляров класса `BacklogItem`, описывающих конструируемое программное обеспечение. Он совсем не похож на продукты, которые кладут в корзину для покупок в интернет-магазинах. Почему? Из-за КОНТЕКСТА. Мы понимаем, что наш класс `Product` означает, потому что он относится к *контексту управления гибким проектированием*. В *контексте интернет-магазина* класс `Product` означает нечто совсем другое. Команде не обязательно называть продукт именем `ScrumProduct`, чтобы правильно понимать его значение.

Смысловое ядро продукта, набор историй, задачи, списки заданий, спринты и выпуски уже имеют более высокие шансы на успех благодаря опыту команды. Однако нам нужны полезные уроки, которые команда может извлечь при тщательном моделировании **АГРЕГАТОВ (AGGREGATES) (10)**.

Резюме

Мы провели серьезное и интенсивное обсуждение важности стратегического проектирования на основе принципов DDD!

- Вы изучили ПРЕДМЕТНЫЕ ОБЛАСТИ, ПОДОБЛАСТИ и ОГРАНИЧЕННЫЕ КОНТЕКСТЫ.
- Вы выяснили, как оценить функционирование предприятия со стратегической точки зрения, используя пространство задач и пространства решений.
- Вы глубоко вникли в детали использования ОГРАНИЧЕННЫХ КОНТЕКСТОВ, чтобы научиться явно лингвистически выделять модели.
- Вы изучили содержимое ОГРАНИЧЕННЫХ КОНТЕКСТОВ, научились определять их правильный размер и способы их создания для развертывания.
- Вы осознали проблемы, с которыми столкнулась команда SaaSOvation в начале проектирования КОНТЕКСТА СОТРУДНИЧЕСТВА, и увидели, как команда выбиралась из этой сложной ситуации.

- Вы увидели формирование текущего СМЫСЛОВОГО ЯДРА, *контекста управления гибким проектированием*, который находится в центре внимания примеров, демонстрирующих проектирование и реализацию.

Как было обещано, в следующей главе начнется глубокое погружение в КАРТЫ КОНТЕКСТОВ. Это важный стратегический инструмент моделирования, предназначенный для использования в проектах. Вы, возможно, уже поняли это, потому что мы уже применяли КАРТЫ КОНТЕКСТОВ в этой главе. Это было неизбежно при оценке различных доменов. Однако в следующей главе мы рассмотрим намного больше деталей.

Глава 3

Карты контекстов

Какой бы курс вы ни избрали, всегда найдется тот, кто скажет вам, что вы не правы. Всегда возникнут сложности, подталкивающие вас к мысли о том, что правы ваши критики. Для того чтобы разработать план действий и следовать ему до конца, нужна храбрость.

Ральф Уолдо Эмерсон

КАРТУ КОНТЕКСТОВ проекта можно представить двумя способами. Проще всего нарисовать простую диаграмму, содержащую отображения между двумя или несколькими существующими **ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (2)**. Следует, однако, понимать, что в этом случае вы рисуете простую диаграмму того, что уже существует. Рисунок иллюстрирует, как ОГРАНИЧЕННЫЕ КОНТЕКСТЫ реального программного обеспечения в пространстве решения связаны друг с другом посредством интеграции. Это значит, что более подробный способ описания КАРТЫ КОНТЕКСТОВ представляет собой реализацию исходного кода интеграции. В этой главе мы рассмотрим оба способа, но большинство деталей реализации мы все же отложим до рассмотрения **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (INTEGRATING BOUNDED CONTEXTS) (13)**.

Имейте в виду, что эта глава посвящена *анализу пространства решений*, в то время как в предыдущей главе мы исследовали *оценку пространства задач*.

Назначение главы

- Объяснить важность создания КАРТЫ КОНТЕКСТОВ для успеха проекта.
- Рассмотреть основные организационные и системные отношения, а также их влияние на проект.
- Научиться на примере команды SaaSOvation разрабатывать КАРТЫ для контроля проекта.

Важность КАРТ КОНТЕКСТОВ

Приступая к проектированию в рамках подхода DDD, сначала нарисуйте визуальную КАРТУ КОНТЕКСТОВ *текущей ситуации вашего проекта.* Нарисуйте КАРТУ КОНТЕКСТОВ, состоящую из текущих ограниченных контекстов вашего проекта, а также интеграционные отношения между ними. На рис. 3.1 показана абстрактная карта контекстов. По мере продвижения вперед мы детализируем ее.

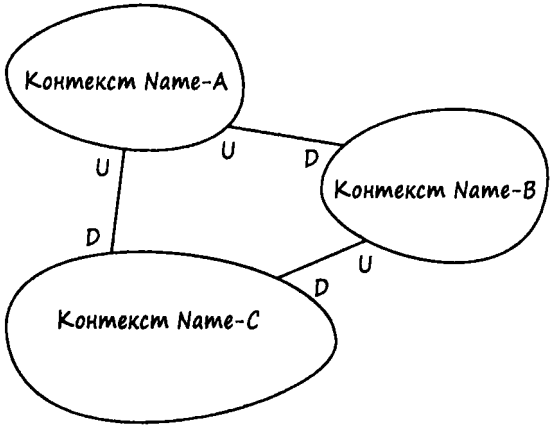


Рис. 3.1. КАРТА КОНТЕКСТОВ **абстрактной** ПРЕДМЕТНОЙ ОБЛАСТИ. **Показаны три** ОГРАНИЧЕННЫХ КОНТЕКСТА **и отношения между ними. Буква "U" означает отношение "вышестоящий", а "D" — отношение "нижестоящий"**

Этот простой рисунок — КАРТА вашей команды. Другие команды проекта могут ссылаться на нее, но при этом, следуя принципам DDD, они обязаны создать собственные КАРТЫ. Основное предназначение вашей карты — дать вашей команде представление о пространстве решений, в котором они пребывают. Другие команды могут не использовать подход DDD и/или не интересоваться вашим мнением.

О, нет! Новая терминология!

Сейчас мы введем термины БОЛЬШОЙ КОМОК ГРЯЗИ (BIG BALL OF MUD), ЗАКАЗЧИК–ПОСТАВЩИК (CUSTOMER–SUPPLIER) и КОНФОРМИСТ (CONFORMIST). Спокойствие! Эти понятия и отношения, связанные с интеграцией, будут рассмотрены в этой главе позднее.

Например, когда вы интегрируете ОГРАНИЧЕННЫЕ КОНТЕКСТЫ в рамках крупного предприятия, вам понадобится интерфейс с **БОЛЬШИМ КОМКОМ ГРЯЗИ**. Команда, поддерживающая целостность "комка грязи", может не интересоваться

направлением вашего проекта, пока вы не присоединитесь к ее интерфейсу прикладного программирования. Ее не интересует ни ваша КАРТА, ни то, что вы будете делать с ее интерфейсом прикладного программирования. И все же ваша КАРТА нужна для отображения отношений с этой командой, потому что *она помогает вашей команде понять ситуацию и обозначает области, в которых необходимо обеспечить взаимодействие между командами.* Это понимание обеспечит успех вашей команды.

Средства коммуникации

Помимо перечня систем, с которыми вы обязаны контактировать, КАРТА КОНТЕКСТОВ служит стимулятором коммуникации между командами.

Представьте себе, что произойдет, если ваша команда будет считать, что команда, поддерживающая целостность "комка грязи", создаст новые интерфейсы прикладного программирования, в то время как она не собирается этого делать или не знает о том, о чем вы думаете. Ваша команда рассчитывает на установление отношения **ЗАКАЗЧИК–ПОСТАВЩИК** с "комком грязи". Однако команда, поддерживающая работу прежней системы, предоставляя только те средства, которые у нее имеются, вынуждает вашу команду установить с ней неожиданное отношение **КОНФОРМИСТ**. В зависимости от того, насколько поздно вы узнаете плохие новости, это невидимое, но реальное отношение может задержать вашу поставку и даже привести к краху проекта. Нарисовав КАРТУ КОНТЕКСТОВ на ранней стадии проекта, вы будете вынуждены осторожно размышлять о ваших отношениях со всеми другими проектами, от которых вы зависите.

> Определите все модели, используемые в проекте, и задайте для каждой свой ОГРАНИЧЕННЫЙ КОНТЕКСТ... Дайте каждому ОГРАНИЧЕННОМУ КОНТЕКСТУ имя и включите эти имена в ЕДИНЫЙ ЯЗЫК проекта. Опишите точки соприкосновения между моделями, явно задавая трансляцию для любого способа коммуникации и выделяя любые совместно используемые ресурсы [Эванс, с. 305].

Приступая к разработке совершенно новой модели, команда CollabOvation должна была использовать КАРТУ КОНТЕКСТОВ. Несмотря на то что модель разрабатывалась практически "с нуля", представление своих предположений о проекте в виде КАРТЫ побудило бы команду подумать об отдельных ОГРАНИЧЕННЫХ КОНТЕКСТАХ. Основные элементы модели следовало бы перечислить на доске, а затем сгруппировать связанные друг с другом лингвистические термины. Это заставило бы команду очертить лингвистические границы и нарисовать простую КАРТУ

КОНТЕКСТОВ. Однако команда не поняла важности стратегического моделирования. Для начала члены команды должны были бы освоить метод стратегического моделирования. Впоследствии они поняли важность этого инструмента, спасающего проект, применив его к общей выгоде. Приступая к разработке СМЫСЛОВОГО ЯДРА проекта, они снова отклонились от курса.

Посмотрим, как можно быстро создать полезную КАРТУ КОНТЕКСТОВ.

Рисование КАРТ КОНТЕКСТОВ

КАРТА КОНТЕКСТОВ описывает *существующую* территорию. Во-первых, мы должны отображать существующее положение дел, а не фантазировать о будущем. Если по мере развития проекта ландшафт изменится, вы сможете обновить КАРТУ. Сначала сосредоточьтесь на текущей ситуации, чтобы понять, где вы находитесь, и определить, куда двигаться дальше.

Создание графических КАРТ КОНТЕКСТОВ не должно быть сложным. Для начала нарисуйте диаграмму маркером на доске. Как указывает Брандолини [Brandolini], стиль "рисуй и стирай" осваивается легко. Если вы решите для рисования диаграмм использовать какой-то инструмент, постарайтесь, чтобы он был неформальным.

Обратите внимание на то, что имена ОГРАНИЧЕННЫХ КОНТЕКСТОВ на рис. 3.1, а также отношения интеграции на диаграмме представляют собой всего лишь пустые поля для заполнения (placeholders). На реальной карте они будут заменены конкретными именами. На диаграмме показаны отношения "вышестоящий" и "нижестоящий", смысл которых будет объяснен в этой главе позднее.

Заметки на доске

Нарисуйте простую диаграмму текущей ситуации, в которой указаны границы и отношения между командами, виды используемой интеграции, а также необходимые трансляции между ними.

> Помните: то, что вы нарисуете, будет реализовано в виде программного обеспечения. Если вам нужна дополнительная информация о том, что вы рисуете, исследуйте систему, с которой интегрируется ваш ОГРАНИЧЕННЫЙ КОНТЕКСТ.

Иногда возникает желание укрупнить диаграмму и добавить детали в конкретную часть КАРТЫ КОНТЕКСТОВ. В этом случае просто возникает другая точка зрения на тот же самый КОНТЕКСТ или те же самые КОНТЕКСТЫ. Помимо

границ, отношений и трансляций, возможно, потребуется включить в диаграмму другие элементы, такие как **МОДУЛИ (9)**, важные **АГРЕГАТЫ (10)**, возможно, места расположения команд, а также другую информацию, относящуюся к данному КОНТЕКСТУ. Этот способ демонстрируется далее в этой главе.

При необходимости все рисунки и любые тексты можно включить в справочник. При этом следует избегать формальностей, чтобы он оставался простым и гибким. Чем более формальными вы будете, тем меньше людей захотят пользоваться вашей КАРТОЙ. Слишком большое количество деталей на диаграммах вряд ли поможет вашей команде. Главное — доверительное общение. Если в ходе разговора вы уточнили стратегическую точку зрения, добавьте его результат в КАРТУ КОНТЕКСТОВ.

Нет, это не АРХИТЕКТУРА ПРЕДПРИЯТИЯ

КАРТА КОНТЕКСТОВ — это *не* АРХИТЕКТУРА ПРЕДПРИЯТИЯ и не диаграмма топологии системы.

Содержащаяся на ней информация относится к взаимодействию моделей и организационных шаблонов DDD. Тем не менее КАРТЫ КОНТЕКСТОВ можно использовать для высокоуровневых архитектурных исследований, если у вас нет других источников информации о структуре предприятия. Кроме того, они помогают выявлять архитектурные недостатки, например узкие места интеграции. Поскольку КАРТЫ КОНТЕКСТОВ отображают организационную динамику, они могут помочь решить даже трудные вопросы управления, блокирующие развитие проекта, а также другие проблемы, возникающие перед командой и руководителями, которые сложно выявить другими методами.

Ковбойская логика

AJ: Жена сказала: "Я была на пастбище с коровами. Ты меня не заметил?" Я ответил: "Не-а". После этого она неделю со мной не разговаривала.

Диаграммы должны постоянно висеть на стенах в ваших кабинетах. Если члены команды часто посещают корпоративный вики-сайт, то диаграммы следует загрузить и туда. Если корпоративный вики-сайт будет часто игнорироваться, не беда. Как говорится, вики-сайт — это место, куда информацию отправляют умирать. Однако независимо от того, где именно отображаются КАРТЫ КОНТЕКСТОВ, на них не будут обращать внимания, если команда не проводит регулярных и содержательных дискуссий.

Проектные и организационные отношения

Напомним, что компания SaaSOvation планирует разработку и совершенствование трех продуктов.

1. Набор программ для обеспечения корпоративного сотрудничества CollabOvation, позволяющий зарегистрированным пользователям публиковать информацию, имеющую бизнес-ценность, используя популярные веб-инструменты, такие как форумы, общие календари, блоги, базы знаний и т.д. Это основной продукт компании SaaSOvation и ее первое **СМЫСЛОВОЕ ЯДРО (2)** (хотя пока команда не знает терминологии DDD). Это КОНТЕКСТ, из которого в конце концов будет извлечена модель IdOvation (точка 2). В данный момент проект CollabOvation использует модель IdOvation как **НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ (2)**. Сам проект CollabOvation будет играть роль **СЛУЖЕБНОЙ ПОДОБЛАСТИ (2)** как вспомогательная надстройка программы ProjectOvation (точка 3).
2. Будучи повторно используемой сущностью и моделью управления доступом, модель IdOvation обеспечивает безопасное ролевое управление доступом для зарегистрированных пользователей. Сначала эти свойства сочетались с проектом CollabOvation (точка 1), но эта реализация была ограниченной и не допускала повторного использования. Тогда компания SaaSOvation выполнила рефакторинг программы CollabOvation, создав новый и ясный ОГРАНИЧЕННЫЙ КОНТЕКСТ. Главной функциональной возможностью продукта является поддержка многих арендаторов, что является жизненно важным для приложения SaaS. Модель IdOvation играет роль СЛУЖЕБНОЙ ПОДОБЛАСТИ для моделей, частью которых она является.
3. Продукт для управления гибким проектированием ProjectOvation в данный момент является новым СМЫСЛОВЫМ ЯДРОМ. Пользователи этого продукта компании SaaS могут создавать ресурсы для управления проектом, а также выполнять анализ и проектирование артефактов и отслеживание прогресса с помощью каркаса на основе методологии Scrum. Как и проект CollabOvation, проект ProjectOvation использует модель IdOvation как НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ. Одна из новаторских функций предусматривает взаимодействие между командами (точка 1) при управлении гибким проектированием, позволяя проводить обсуждение продуктов, выпусков, спринтов и отдельных задач в рамках технологии Scrum.

Итак, определения!

Упомянутые выше проектные и организационные отношения определяются ответами на следующие вопросы.

Какие отношения существуют между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ и отдельными командами проекта? Существует несколько организационных и интеграционных шаблонов DDD, один из которых почти всегда существует между любыми двумя ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Почти все определения, приведенные ниже, взяты из книги [Evans, Ref].

- **ПАРТНЕРСТВО (PARTNERSHIP).** Когда команды в двух КОНТЕКСТАХ достигают успеха или терпят неудачу вместе, должно возникнуть отношение сотрудничества. Эти команды устанавливают процесс координированного планирования разработки и совместное управление интеграцией. Они должны сотрудничать в процессе эволюции своих интерфейсов, чтобы учитывать потребности обеих систем. Взаимозависимые функции должны быть организованы так, чтобы завершение их разработки происходило в рамках одного и того же выпуска.
- **ОБЩЕЕ ЯДРО (SHARED KERNEL).** Общая часть модели и связанного с ней кода образует очень тесную взаимосвязь, которая может как способствовать работе, так и мешать ей. Обозначьте четкую границу определенного подмножества модели предметной области, которую команды согласны считать общей. Ядро должно быть маленьким. Оно имеет особый статус и не должно изменяться без консультаций с другой командой. Установите непрерывный интеграционный процесс, сохраняющий ядро небольшим, и согласуйте **ЕДИНЫЙ ЯЗЫК (1)** команд.
- **РАЗРАБОТКА «ЗАКАЗЧИК-ПОСТАВЩИК» (CUSTOMER-SUPPLIER DEVELOPMENT).** Когда две команды находятся в отношении "нижестоящий–вышестоящий", причем успех вышестоящей команды зависит от нижестоящей команды, потребности нижестоящей команды можно удовлетворить разными способами, имеющими разные последствия. Следует учитывать приоритеты нижестоящих команд при планировании работы вышестоящей команды. Проводите переговоры и согласовывайте сметы, учитывающие потребности нижестоящих команд, чтобы все понимали взятые обязательства и календарный план.
- **КОНФОРМИСТ (CONFORMIST).** Когда две команды находятся в отношении "вышестоящий–нижестоящий", причем вышестоящая команда не имеет причин учитывать потребности нижестоящей команды, то нижестоящая команда беспомощна. Руководствуясь альтруизмом, члены вышестоящей команды могут давать обещания, но они вряд ли будут выполнены. Нижестоящая команда устраняет сложность трансляции между ограниченными контекстами, беспрекословно подчиняясь модели вышестоящей команды.
- **ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER).** При объединении хорошо продуманных ограниченных контекстов с тесно

сотрудничающими командами уровни трансляции могут быть простыми и даже элегантными. Однако если управление или коммуникация не соответствует общему ядру, партнеру или отношению "заказчик–поставщик", то трансляция становится более сложной. Уровень трансляции усиливает защиту. Нижестоящий клиент должен создать изолирующий слой, чтобы обеспечить свою систему функциональной возможностью вышестоящей системы в терминах своей модели предметной области. Этот уровень общается с другой системой с помощью существующего интерфейса, не требуя или почти не требуя модификации другой системы. Внутренне он транслирует информацию в одном или обоих направлениях между моделями.

- **СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ (OPEN HOST SERVICE).** Определите протокол, предоставляющий доступ к вашей системе, как к набору служб. Откройте этот протокол, чтобы его могли использовать все, кто захотят интегрироваться с вами. Уточните и расширьте этот протокол, чтобы учесть новые требования интеграции, за исключением специфических потребностей. Затем используйте одноразовый транслятор для расширения протокола в особых ситуациях, чтобы протокол оставался простым и согласованным.
- **ОБЩЕДОСТУПНЫЙ ЯЗЫК (PUBLISHED LANGUAGE).** Трансляция между моделями двух ОГРАНИЧЕННЫХ КОНТЕКСТОВ требует общего языка. Используйте в качестве основной среды коммуникации хорошо документированный общий язык, который может выразить необходимую информацию о предметной области, выполняя при необходимости перевод информации с другого языка на этот и с этого языка на другой. ОБЩЕДОСТУПНЫЙ ЯЗЫК часто сочетается со СЛУЖБОЙ С ОТКРЫТЫМ ПРОТОКОЛОМ.
- **ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS).** Мы должны быть строгими при определении требований. Если между двумя наборами функциональных возможностей нет важного отношения, их можно полностью отсоединить друг от друга. Интеграция всегда дорого стоит, а выгоды иногда бывают незначительными. Объявите ограниченный контекст, не имеющий связей с другими контекстами. Это позволит разработчикам найти простые и специальные решения в этой небольшой области.
- **БОЛЬШОЙ КОМОК ГРЯЗИ (BIG BALL OF MUD).** Выполняя обзор имеющихся систем, мы обнаруживаем, что существуют части системы, в которых модели перемешаны, а границы стерты. Нарисуйте границу вокруг такой смеси и обозначьте ее как БОЛЬШОЙ КОМОК ГРЯЗИ. Не пытайтесь применять изощренное моделирование внутри этого КОНТЕКСТА. Учтите, что такие системы часто проникают в другие КОНТЕКСТЫ.

Интеграция с *контекстом идентификации и доступа* позволяет *контексту сотрудничества* и *контексту управления гибким проектированием* избежать ОТ-

ДЕЛЬНОГО СУЩЕСТВОВАНИЯ с точки зрения безопасности и полномочий. Действительно, шаблон ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ может применяться в рамках КОНТЕКСТА конкретной системы, но в то же время его можно использовать в каждом случае отдельно. Например, одна команда может отказаться использовать централизованную систему безопасности и решить интегрироваться с другими стандартными системами корпорации.

Команды могут вступать в сотрудничество, играя роли ЗАКАЗЧИК–ПОСТАВЩИК. Руководство компании SaaSOvation не может позволить одной команде вынудить остальных стать КОНФОРМИСТАМИ. Это не значит, что роль КОНФОРМИСТА всегда является отрицательной. Просто шаблон ЗАКАЗЧИК–ПОСТАВЩИК требует, чтобы часть ПОСТАВЩИКА обеспечивала поддержку ЗАКАЗЧИКА, способствуя укреплению отношений между командами, которое руководство компании SaaSOvation считает необходимым для достижения цели. Разумеется, ЗАКАЗЧИКИ не всегда правы, поэтому должен существовать определенный компромисс. Но в целом это полезное организационное отношение, которое команды должны поддерживать.

Для поддержки интеграционных связей между командами можно использовать СЛУЖБУ С ОТКРЫТЫМ ПРОТОКОЛОМ и ОБЩЕДОСТУПНЫЙ ЯЗЫК. Кроме того, команды могут создать ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ, хотя это может показаться странным. Это не противоречие, даже несмотря на то что между их ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ установлены открытые стандарты. Они могут по-прежнему извлекать выгоду из изолированной трансляции, используя ее основные принципы при взаимодействии с нижестоящими КОНТЕКСТАМИ, но при этом по сравнению с БОЛЬШИМ КОМКОМ ГРЯЗИ уровень сложности снижается. Уровни трансляции становятся простыми и элегантными.

В последующих КАРТАХ КОНТЕКСТОВ используются следующие аббревиатуры, обозначающие шаблоны на каждом из концов отношения.

- ACL (ANTICORRUPTION LAYER — ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ)
- OHS (OPEN HOST SERVICE — СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ)
- PL (PUBLISHED LANGUAGE — ОБЩЕДОСТУПНЫЙ ЯЗЫК)

Анализируя следующий пример КАРТ КОНТЕКСТОВ и сопровождающих их текстов, полезно вернуться к главе 2, "Предметные области, подобласти и ограниченные контексты". Кроме того, полезно вспомнить три примера ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Поскольку эти диаграммы относятся к высокому уровню, их можно включать в КАРТЫ каждого КОНТЕКСТА, хотя здесь они не повторяются.

Карты трех контекстов

Вернемся к нашей команде, на ошибках которой мы пытаемся научить вас принципам DDD...

Когда команда CollabOvation поняла, что создала петлю, ее члены погрузились в изучение книги [Эванс] в поисках выхода. Они не только поняли огромную важность стратегических проектных шаблонов, но и открыли для себя практический инструмент под названием КАРТЫ КОНТЕКСТОВ. Кроме того, они нашли в Интернете статью [Brandolini], описывающую этот метод. Поскольку в статье было сказано, что они должны нарисовать карту существующей территории, команда решила, в первую очередь, сделать именно это. Результаты показаны на рис. 3.2.

Первая КАРТА, созданная командой, отражает первичное представление о существовании ОГРАНИЧЕННОГО КОНТЕКСТА под названием *контекст сотрудничества*. Довольно странная граница этого контекста свидетельствует о том, что команда подозревает о существовании второго КОНТЕКСТА, но не понимает, где проходит четкая граница, отделяющая его от СМЫСЛОВОГО ЯДРА.

Рис. 3.2. Петля внутри *контекста сотрудничества* вызвана нежелательными концепциями, выявленными в этой *карте*. Предупреждающий знак отмечает засоренную область

Узкий проход в верхней части карты позволяет внешним концепциям мигрировать вперед и назад практически без цензуры. Именно об этом предупреждает знак опасности. Это не значит, что границы КОНТЕКСТА должны быть полностью непроницаемыми. Просто, как и в случае настоящих границ государства, команда хочет, чтобы *контекст сотрудничества* владел полной информацией о том, что пересекает его границы и с какой целью. В противном случае территория становится проходным двором для неизвестных и нежелательных визитеров. Нежелательные визитеры обычно вызывают путаницу и ошибки в моделях. Разработчики моделей должны быть приветливыми и даже гостеприимными, но при условии сохранения порядка и гармонии. Любые внешние концепции, пересекающие границы, должны продемонстрировать право на это, даже приобретая характеристики, совместимые с внутренними свойствами территории.

Этот анализ привел к более глубокому пониманию не только текущего состояния модели, но и направления, в котором должен развиваться проект. Поняв, что такие концепции, как безопасность, пользователи и полномочия, не принадлежат *контексту сотрудничества*, команда смогла принять правильное решение. Команда должна была отделить эти концепции от СМЫСЛОВОГО ЯДРА и позволить их включение только при благоприятных условиях.

Это жизненно важное свойство проекта DDD. Для того чтобы все модели оставались ясными, должны соблюдаться правила ЯЗЫКА каждого ОГРАНИЧЕННОГО КОНТЕКСТА. Лингвистическая сегрегация и строгое выполнение ее правил помогут каждой команде, вовлеченной в проект, сосредоточить внимание на своем ОГРАНИЧЕННОМ КОНТЕКСТЕ и своих задачах.

Анализ ПОДОБЛАСТИ, или пространства задач, привел команду к диаграмме, показанной на рис. 3.3. Оказалось, что ОГРАНИЧЕННЫЙ КОНТЕКСТ состоит из ПОДОБЛАСТЕЙ. Поскольку желательно, чтобы между ПОДОБЛАСТЯМИ и ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ существовало однозначное соответствие, этот анализ продемонстрировал необходимость разделить один ОГРАНИЧЕННЫЙ КОНТЕКСТ на два.

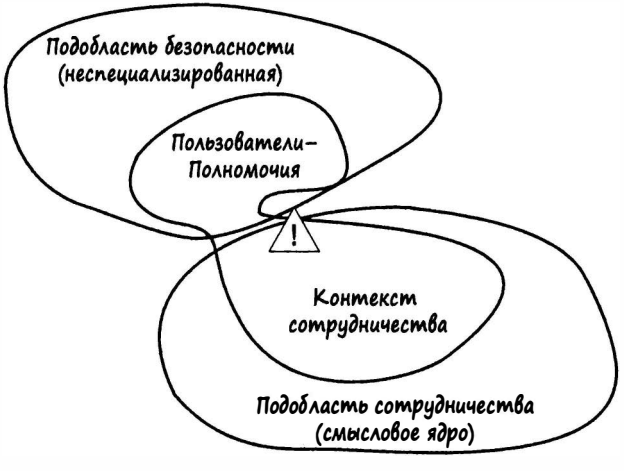


Рис. 3.3. Анализ ПОДОБЛАСТИ позволил обнаружить две новые подобласти: *смысловое ядро сотрудничества* и *неспециализированную подобласть безопасности*

Анализ ПОДОБЛАСТИ и ее границ привел команду к принятию решений. Когда пользователи системы CollabOvation взаимодействуют с доступными функциями, они делают это как УЧАСТНИКИ (PARTICIPANTS), АВТОРЫ (AUTHORS), МОДЕРАТОРЫ (MODERATORS) и т.д. Разнообразие других возможностей, касающихся разделения контекста, мы обсудим позднее, а пока целесообразно разделить созданный контекст. Зная это, можно провести четкие границы, очерчивающие высокоуровневую КАРТУ КОНТЕКСТОВ, как показано на рис. 3.4. Для рефакторинга контекста команда использовала шаблон **ВЫДЕЛЕННОЕ ЯДРО (SEGREGATED CORE)** [Эванс]. Распознанные границы каждого КОНТЕКСТА обозначаются пиктограммами или визуальными знаками. Выделение таких же фигур во всех диаграммах может облегчить процесс познания предметной области.

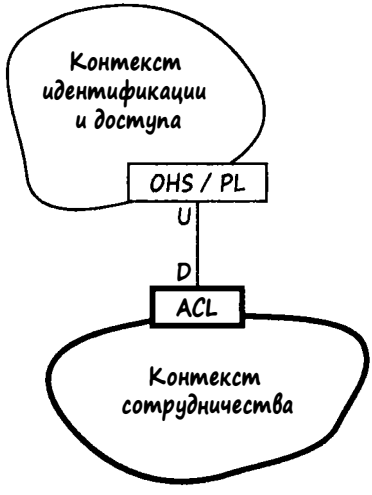


Рис. 3.4. Исходное СМЫСЛОВОЕ ЯДРО обозначено сплошной линией и точками интеграции. В данном случае система IdOvation служит НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ по отношению к нижестоящей системе CollabOvation

КАРТЫ КОНТЕКСТОВ не появляются все сразу, как это может показаться, хотя после завершения анализа их нетрудно воспроизвести. Размышления и обсуждения помогают уточнить КАРТУ путем быстрых итераций. Некоторые уточнения могут выразиться в виде точек интегрирования, описывающих отношения между КОНТЕКСТАМИ.

Первые две карты отображают результаты, достигнутые после применения стратегического проектирования. После того как исходный проект CollabOvation был готов, команда должна была выполнить рефакторинг системы идентификации

и доступа. В итоге она создала КАРТУ КОНТЕКСТОВ, изображенную на рис. 3.4. Команда нарисовала только **СМЫСЛОВОЕ ЯДРО**, которым является *контекст сотрудничества*, а также новую НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ *контекста идентификации и доступа*. Члены команды не стали рисовать другие будущие модели, такие как *контекст управления гибким проектированием*. Это не слишком помогло бы делу. Им было необходимо исправить лишь то, что уже существовало. Трансформации служебных систем, которые еще только будут созданы, делать не требовалось, поэтому эту работу команда отложила на будущее.

Заметки на доске

- Можете ли вы, размышляя о своем ограниченном контексте, идентифицировать концепции, которые ему не принадлежат? Если да, нарисуйте новую КАРТУ КОНТЕКСТОВ, на которой показаны желательные КОНТЕКСТЫ и отношения между ними.
- Какие из девяти организационных и интеграционных отношений DDD вы бы выбрали и почему?

Когда был начат следующий проект, в котором использовалась система ProjectOvation, настал момент использовать существующую КАРТУ вместе с новым СМЫСЛОВЫМ ЯДРОМ — *контекстом управления гибким проектированием*. Результаты изображены на рис. 3.5. Эти результаты не совпали с ожидаемыми, хотя их еще даже не программировали. Детали внутри нового КОНТЕКСТА были не до конца понятны, но это понимание должно было появиться в ходе обсуждений. Применение высокоуровневого стратегического проектирования на столь ранней стадии должно было помочь всем командам выяснить свои зоны ответственности. Поскольку эти три высокоуровневые КАРТЫ представляют собой всего лишь уточнение предыдущих, мы сосредоточим внимание именно на них. В этом месте на первый план выходит система SaaSOvation. Компания назначила руководителем нового проекта опытного сотрудника. Имея на вооружении три КОНТЕКСТА и план дальнейшей работы, руководство решило направить на разработку нового СМЫСЛОВОГО ЯДРА лучших проектировщиков.

Некоторые существенные моменты сегрегации уже хорошо понятны. Аналогично *контексту сотрудничества*, в котором пользователи системы ProjectOvation создают продукцию, планируют выпуски, разрабатывают календарный план для спринтов и работают над выбором задач, в *контексте управления гибким проектированием* пользователи делают все это как ВЛАДЕЛЬЦЫ ПРОДУКТА (PRODUCT OWNERS) и ЧЛЕНЫ КОМАНДЫ (TEAM MEMBERS). *Контекст идентификации и доступа* выделен из СМЫСЛОВОГО ЯДРА. То же самое касается *контекста сотрудничества*. Теперь это СЛУЖЕБНАЯ ПОДОБЛАСТЬ. Любое ее использование в новой модели будет защищено границами и трансляциями в концепции СМЫСЛОВОГО ЯДРА.

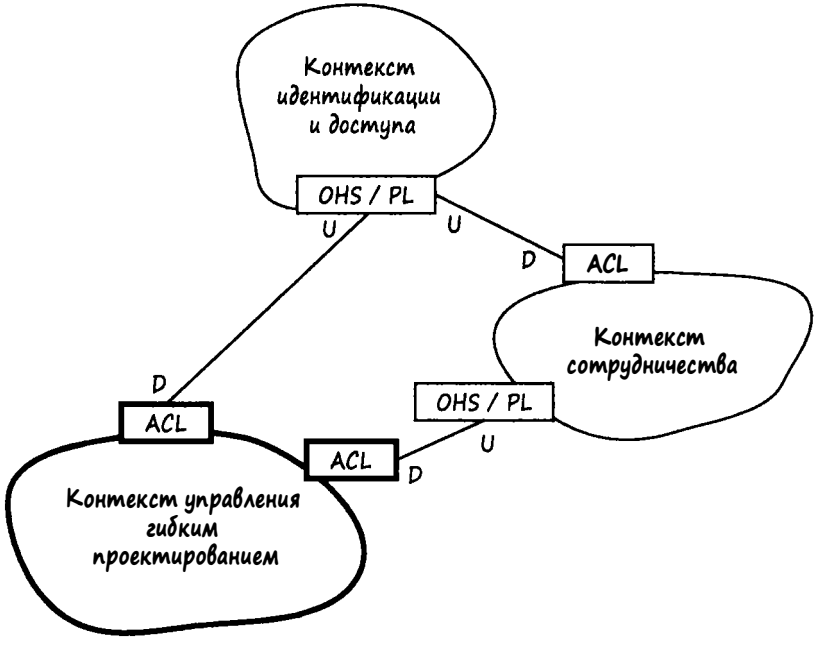


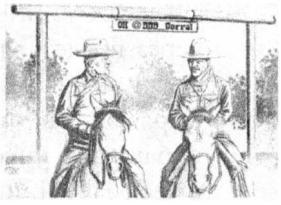
Рис. 3.5. Текущее СМЫСЛОВОЕ ЯДРО обозначено сплошной линией и точками интеграции. СЛУЖЕБНАЯ ПОДОБЛАСТЬ CollabOvation и НЕСПЕЦИАЛИЗИРОВАННАЯ ПОДОБЛАСТЬ IdOvation являются нижестоящими

Рассмотрим более мелкие детали этих диаграмм. Это не архитектурные диаграммы системы. Если бы это было так, то новое СМЫСЛОВОЕ ЯДРО — *контекст управления гибким проектированием* — должно было бы располагаться вверху или в центре диаграммы. В данном случае оно расположено внизу. Эта, возможно, тонкая особенность свидетельствует о том, что модель ядра является нижестоящей по отношению к другим.

Этот нюанс служит также визуальным сигналом. Вышестоящие модели оказывают влияние на нижестоящие, как города, стоящие выше по реке, положительно или отрицательно влияют на города, расположенные ниже по течению. Представьте себе загрязнения, выбрасываемые в реку крупным городом. Эти загрязнения могут мало влиять на сам город, но города, расположенные ниже по течению, могут столкнуться с серьезными последствиями. Вертикальная близость моделей, расположенных на диаграмме, помогает идентифицировать влияние вышестоящих моделей на нижестоящие. Метки U и D возле линий, соединяющих связанные модели, явно указывают на это. Эти метки делают вертикальное позиционирование каждого КОНТЕКСТА менее важным, хотя визуально они выглядят привлекательно.

Ковбойская логика

LB: Если очень хочешь пить, пристраивайся на водопое выше стада по течению.



Контекст идентификации и доступа — это следующий вышестоящий контекст. Он влияет на *контекст сотрудничества* и *контекст управления гибким проектированием*. Наш *контекст сотрудничества* также является вышестоящим по отношению к *контексту управления гибким проектированием*, потому что гибкие модели зависят от модели сотрудничества и служб. Как указано в главе 2, система ProjectOvation работает автономно, поскольку это практично. Операции должны быть максимально независимыми от доступности окружающих систем. Это не значит, что автономные системы должны работать совершенно независимо от вышестоящих моделей. Проектирование должно существенно ограничивать прямые зависимости в реальном времени. Несмотря на свою автономность, *контекст управления гибким проектированием* остается нижестоящим по отношению к остальным.

Установка приложения с автономными службами не означает, что базы данных из вышестоящих КОНТЕКСТОВ просто реплицируются в зависимый КОНТЕКСТ. Репликация вынудила бы локальную систему принять на себя нежелательные обязательства. Это потребовало бы создания ОБЩЕГО ЯДРА (SHARED KERNEL), которое уничтожило бы автономность.

Обратите внимание на узлы, расположенные на вышестоящей стороне каждого соединения на последней КАРТЕ. Оба соединения помечены аббревиатурой OHS/PL, идентифицирующей СЛУЖБУ С ОТКРЫТЫМ ПРОТОКОЛОМ и ОБЩЕДОСТУПНЫЙ ЯЗЫК. Все три узла соединения на нижестоящей стороне помечены аббревиатурой ACL, означающей ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ. Техническая реализация этих узлов описана в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**. Коротко говоря, эти шаблоны интегрирования обладают следующими характеристиками.

- **СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ.** Этот шаблон можно реализовать в виде REST-ресурсов, с которыми взаимодействует клиент ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Обычно мы представляем СЛУЖБУ С ОТКРЫТЫМ ПРОТОКОЛОМ как удаленный вызов процедуры (Remote Procedure Call — RPC) из интерфейса прикладного программирования, но ее можно реализовать и с помощью обмена сообщениями.
- **ОБЩЕДОСТУПНЫЙ ЯЗЫК.** Его можно реализовать несколькими способами, но часто его представляют в виде XML-схемы. Если выразить

ОБЩЕДОСТУПНЫЙ ЯЗЫК с помощью REST-служб, то он будет выражаться через представления концепций предметной области. Эти представления могут включать в себя, например, форматы XML и JSON. Его можно также выразить как буферы протокола Google (Google Protocol Buffers). Если вы публикуете интерфейсы веб-пользователей, то они могут также включать в себя HTML-представления. Одно из преимуществ использования архитектурного стиля REST состоит в том, что клиент может указать предпочтительный ОБЩЕДОСТУПНЫЙ ЯЗЫК, а ресурсы сгенерируют представления в соответствии с требуемым типом контента. Стиль REST также позволяет создавать гипермедийные представления, использующие архитектуру HATEOAS. Гипермедиа делает ОБЩЕДОСТУПНЫЙ ЯЗЫК очень динамичным и интерактивным, позволяя клиентам перемещаться по связанным ресурсам. ЯЗЫК может стать общедоступным благодаря использованию стандартных и/или специальных типов медиа. ОБЩЕДОСТУПНЫЙ ЯЗЫК используется также в **СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ (EVENT-DRIVEN ARCHITECTURE) (4)**, в которой **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS) (8)** передаются как сообщения всем заинтересованным подписчикам.

- **ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ. СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICE) (7)** для каждого типа ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ можно определить в нижестоящем контексте. Кроме того, ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ можно разместить за интерфейсом **ХРАНИЛИЩА (REPOSITORY) (12)**. Если используется архитектура REST, то реализация клиента СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ обращается к СЛУЖБЕ С ОТКРЫТЫМ ПРОТОКОЛОМ. Ответы сервера образуют представления на ОБЩЕДОСТУПНОМ ЯЗЫКЕ. Нижестоящий ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ транслирует представления в объекты предметной области в своем локальном КОНТЕКСТЕ. Здесь, например *контекст сотрудничества* запрашивает у *контекста идентификации и доступа* роль ПОЛЬЗОВАТЕЛЬ-МОДЕРАТОР. Он может получить требуемый ресурс в формате XML или JSON, а затем транслировать его в экземпляр класса `Moderator`, представляющий собой ОБЪЕКТ-ЗНАЧЕНИЕ. Новый экземпляр класса `Moderator` отражает некую концепцию в терминах нижестоящей, а не вышестоящей модели.

Выбранные шаблоны носят общий характер. Ограничение выбора помогает сохранять под контролем объем интеграции, обсуждаемой в этой книге. Как мы увидим, даже среди этих немногих избранных шаблонов существует большое разнообразие способов их применения.

Остается вопрос "Это все, что есть для создания КАРТЫ КОНТЕКСТОВ?" Возможно. Представление высокого уровня обеспечивает хороший объем знаний о

проекте в целом. Однако нас может заинтересовать, куда идут связи и что означают именованные отношения на каждом КОНТЕКСТЕ. Любопытство членов команды стимулирует нас к детализации. Когда мы увеличиваем масштаб, то несколько размытое изображение трех шаблонов интеграции становится более ясным.

Давайте ненадолго вернемся назад. Поскольку *контекст сотрудничества* был первым СМЫСЛОВЫМ ЯДРОМ, давайте всмотримся в него. Сначала мы введем метод увеличения масштаба на примере более простой интеграции, а затем перейдем к более сложным вариантам.

Контекст сотрудничества

Вернемся к нашей команде сотрудничества...

Контекст сотрудничества был первой моделью и системой — первым ОСНОВНЫМ ЯДРОМ, — и его принципы работы в данный момент хорошо понятны. Интеграция, используемая здесь, более простая, но менее устойчивая с точки зрения надежности и автономии. Создание масштабируемой КАРТЫ КОНТЕКСТОВ оказалось сравнительно простой задачей.

В качестве клиента REST-служб, опубликованных *контекстом идентификации и доступа*, *контекст сотрудничества* использует для получения ресурсов традиционные вызовы удаленных процедур. Этот КОНТЕКСТ не записывает данные, полученные от *контекста идентификации и доступа*, на которые он может впоследствии сослаться для локального повторного использования. Вместо этого он обращается к удаленной системе, чтобы запрашивать информацию каждый раз, когда в ней возникает потребность. Этот *контекст*, очевидно, очень зависит от удаленных служб и не автономен. Система SaaSOvation на данный момент готова к работе. Интеграция с НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТЬЮ была абсолютно неожиданна. Для того чтобы выполнить ее напряженный график поставки, команда не могла тратить время на создание более тщательно продуманного автономного проекта. В то же время нельзя было отказаться от принципов простоты проекта. После развертывания системы ProjectOvation и попытки работать автономно подобные методы могут использоваться и для разработки системы CollabOvation.

Пограничные объекты на масштабируемой КАРТЕ, представленной на рис. 3.6, запрашивают ресурс синхронно. Когда представление удаленной модели получено, пограничные объекты извлекают интересующее их содержание и транслируют его, создавая надлежащий экземпляр ОБЪЕКТА-ЗНАЧЕНИЯ. На рис. 3.7 показана КАРТА ТРАНСЛЯЦИИ, предназначенная для того, чтобы превратить представление в ОБЪЕКТ-ЗНАЧЕНИЕ. Здесь объект класса `User`, играющий роль (`Role`) МОДЕРАТОРА в *контексте идентификации и доступа*, транслируется в ОБЪЕКТ-ЗНАЧЕНИЕ `Moderator` в *контексте сотрудничества*.

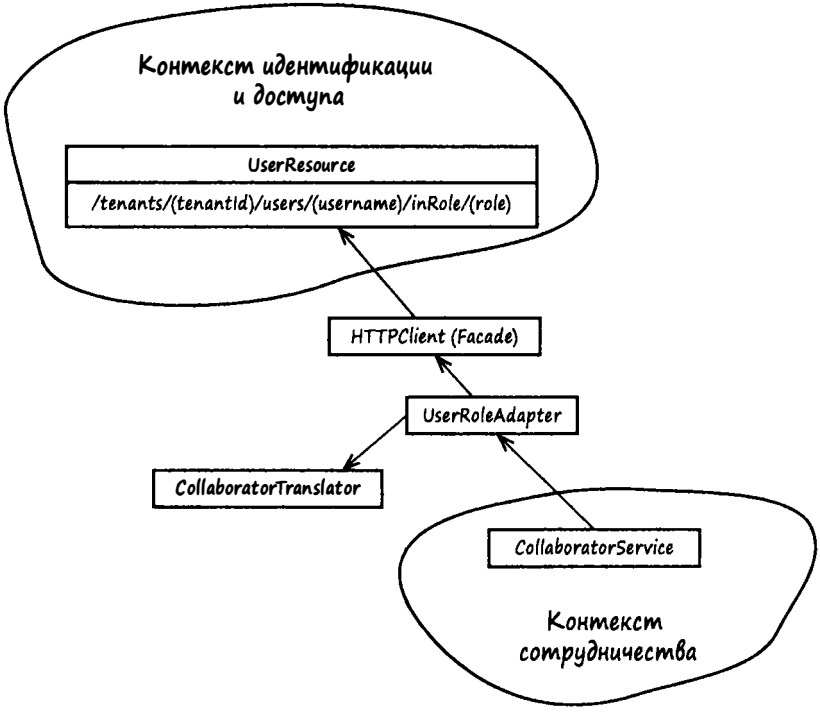


Рис. 3.6. Увеличение масштаба ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ **и** СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ, **участвующих в интеграции между** *контекстом сотрудничества* **и** *контекстом идентификации и доступа*

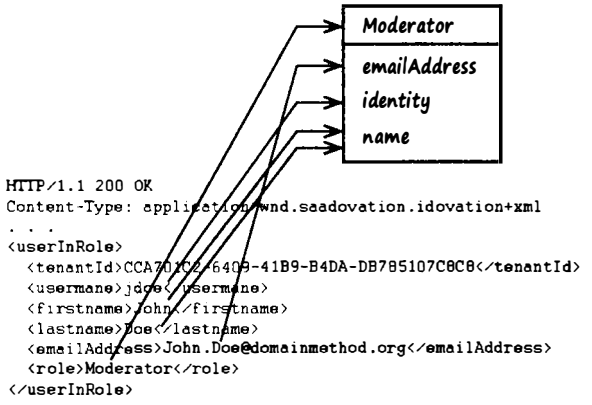


Рис. 3.7. Логическая КАРТА ТРАНСЛЯЦИИ, демонстрирующая, как состояние представления (в данном случае — XML) отображается в ОБЪЕКТ-ЗНАЧЕНИЕ

Заметки на доске

Создайте КАРТУ ТРАНСЛЯЦИИ одного из аспектов интеграции, обнаруженных в ограниченном контексте вашего проекта.

> Что случится, если трансляции окажутся слишком сложными и потребуют копирования и синхронизации слишком большого объема данных, так что транслированный объект будет похож на объект из другой модели? Возможно, это свидетельствует о том, что вы используете слишком много объектов из внешнего ОГРАНИЧЕННОГО КОНТЕКСТА, адаптируете слишком много информации из этой модели и создаете путаницу в своей модели.

К сожалению, если произошел отказ при синхронном запросе из-за того, что удаленная система недоступна, выполнение всех локальных процессов должно прекратиться. Пользователю сообщат о проблеме и попросят попробовать еще раз позже.

Системная интеграция обычно полагается на удаленные вызовы процедур. На высоком уровне удаленные вызовы процедур похожи на обычные вызовы процедур в языках программирования. Библиотеки и инструменты делают вызов привлекательным и простым в использовании. Однако в отличие от вызова процедуры, которая находится в вашем собственном пространстве процесса, удаленный вызов имеет более высокую вероятность снижения производительности или прямого отказа. Загрузка сетевой и удаленной системы может задержать завершение

удаленного вызова процедуры. Если целевая система удаленного вызова процедуры будет недоступна, то запрос пользователя к вашей системе не завершится успешно.

В то время как использование REST-ресурсов на самом деле не сводится к удаленным вызовам процедур, у него все же есть аналогичные характеристики. Несмотря на то, что отказ всей системы — относительно редкое событие, это потенциально раздражающее ограничение. Команда надеется изменить к лучшему эту ситуацию как можно скорее.

Контекст управления гибким проектированием

Поскольку *контекст управления гибким проектированием* — новое СМЫСЛОВОЕ ЯДРО, он заслуживает особого внимания. Увеличим его масштаб и рассмотрим его связи с другими моделями.

Для того чтобы повысить степень автономности по сравнению с удаленными вызовами процедур, команда, разрабатывающая *контекст управления гибким проектированием*, должна осторожно ограничить его использование. По этой причине была принята внеполосная, или асинхронная, обработка событий.

Большая степень автономии может быть достигнута, если в нашей локальной системе уже существует зависимое состояние. Можно представить себе кеш всех зависимых объектов, но при использовании DDD это обычно не так. Вместо этого мы создаем локальные объекты предметной области, транслированные из внешней модели, поддерживая только минимальное количество состояний, необходимых в локальной модели. Для того чтобы вывести состояние на первый план, мы, возможно, должны выполнить удаленные вызовы процедур или подобные запросы REST-ресурсов. Однако любой необходимой синхронизации с изменениями удаленной модели часто можно достичь посредством уведомлений, публикуемых удаленными системами в виде сообщений. Уведомления могли бы быть отправлены по сервисной шине или через очередь сообщений или опубликованы через систему REST.

> **Стремитесь к минимализму**
>
> Синхронизированное состояние является ограниченным, минимальным атрибутом удаленных моделей, необходимым для локальных моделей. Вопрос не только в том, чтобы ограничить наши потребности в синхронизации данных, но и в правильном моделировании концепций.

Целесообразно ограничить использование удаленного состояния, даже рассматривая возможность проектирования элементов локальной модели. Например, объекты классов `ProductOwner` и `TeamMember` в реальности не должны отражать объекты классов `UserOwner` и `UserMember`, потому что они получают

так много характеристик удаленного объекта User, что невольно возникает гибридизация.

Интеграция с контекстом идентификации и доступа

Анализируя увеличенную КАРТУ, изображенную на рис. 3.8, мы видим, что ресурсы URI обеспечивают уведомление о значительных СОБЫТИЯХ ПРЕДМЕТНОЙ ОБЛАСТИ, происходящих в *контексте идентификации и доступа*. Они доступны благодаря экземпляру класса NotificationResource, который публикует ресурс RESTful. Ресурсы уведомлений — это группы публикуемых СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ. Каждое опубликованное СОБЫТИЕ всегда доступно для получения в порядке поступления, но каждый клиент несет ответственность за предотвращение получения дубликатов.

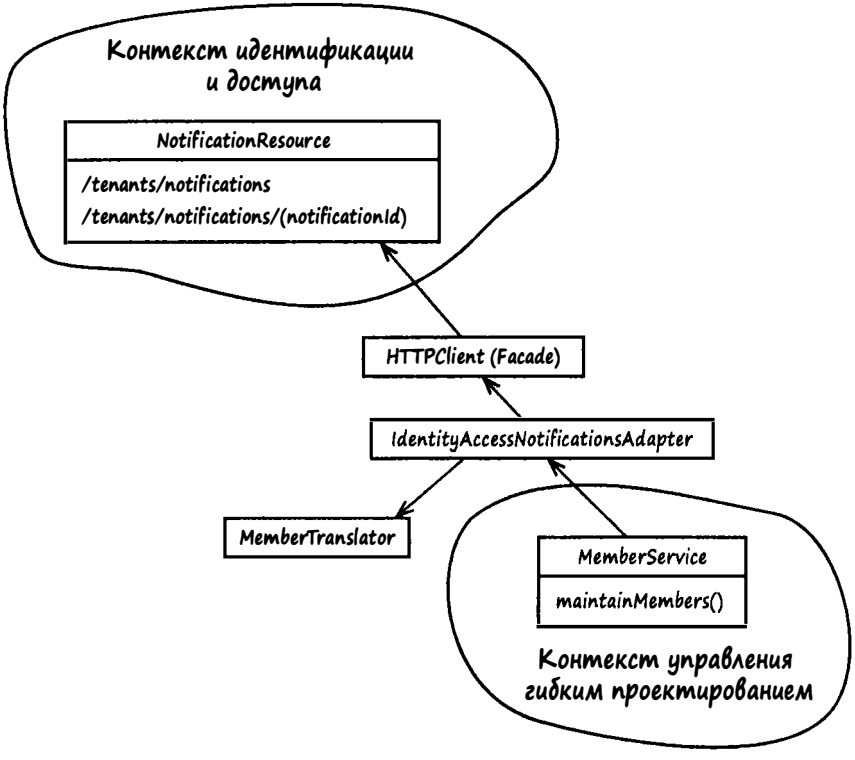


Рис. 3.8. Увеличение масштаба ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ и СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ, участвующих в интеграции между *контекстом сотрудничества* и *контекстом идентификации и доступа*

Тип пользовательского носителя означает, что нам могут понадобиться два ресурса.

```
application/vnd.saasovation.idovation+json
//iam/notifications
//iam/notifications/{notificationId}
```

Адрес URI первого ресурса позволяет клиентам получать (точнее с помощью запроса HTTP GET) журнал текущих уведомлений (фиксированный набор индивидуальных уведомлений). В типе документированного пользовательского носителя

```
application/vnd.saasovation.idovation+json
```

Адрес URI ресурса считается стабильным, потому что никогда не изменяется. Независимо от того, из чего состоит текущий журнал уведомления, этот URI обеспечивает его. Текущий журнал — это ряд новых событий, которые произошли в модели идентификации и доступа. Второй адрес URI ресурса позволяет клиентам получить и перемещаться по цепочке всех предыдущих основанных на событии уведомлений, которые были заархивированы. Зачем нужен текущий журнал и разные заархивированные журналы уведомлений? Подробности, касающиеся уведомлений, передаваемых по каналам, изложены в главах 8 и 13.

На самом деле команда ProjectOvation не обязана использовать архитектуру REST во всех случаях. Например, в настоящий момент она ведет переговоры с командой CollabOvation об использовании альтернативной инфраструктуры передачи сообщений. Кроме того, рассматривается возможность использования платформы RabbitMQ. Тем не менее пока их механизмы интеграции с *контекстом идентификации и доступа* основаны все же на архитектуре REST.

Теперь отложим в сторону технологические подробности и рассмотрим роль, которую играет каждый объект в крупномасштабной КАРТЕ. Ниже приводится объяснение этапов интеграции, продемонстрированных на рис. 3.9.

- Класс `MemberService` — это СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ, ответственная за передачу экземпляров классов `ProductOwner` и `TeamMember` в локальную модель. Она представляет собой интерфейс базового ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ. В частности, метод `maintainMembers()` периодически используется для проверки новых уведомлений, поступающих от КОНТЕКСТА ИДЕНТИФИКАЦИИ И ДОСТУПА. Этот метод не вызывается обычными клиентами модели. При очередном срабатывании таймера компонент, получивший уведомление, использует объект класса `MemberService`, вызывая метод `maintainMembers()`. На рис. 3.9 получателем уведомлений от таймера служит экземпляр класса `MemberSynchronizer`, делегирующий его обработку службе `MemberService`.
- Служба `MemberService` делегирует уведомление экземпляру класса `IdentityAccessNotificationAdapter`, играющему роль АДАПТЕРА между

СЛУЖБОЙ ПРЕДМЕТНОЙ ОБЛАСТИ и удаленной СЛУЖБОЙ С ОТКРЫТЫМ ПРОТОКОЛОМ. АДАПТЕР действует как клиент удаленной системы. Взаимодействие с удаленным объектом класса `NotificationResource` не показано.

- Как только АДАПТЕР получает ответ от удаленной СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ, он делегирует его экземпляру класса `MemberTranslator` для перевода носителя ОБЩЕДОСТУПНОГО ЯЗЫКА в концепции локальной системы. Если локальный экземпляр класса `Member` уже существует, трансляция обновляет существующий объект предметной области. Этот факт показан как самоделегирование сообщения от объекта класса `MemberService` его внутреннему методу `updateMember()`. Класс `Member` имеет подклассы `ProductOwner` и `TeamMember`, отражающие локальные концепции контекста.

Мы не будем рассматривать технологии или средства интеграции. Вместо этого, четко выделив ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, мы сможем сохранить чистоту каждого КОНТЕКСТА, применяя данные из других КОНТЕКСТОВ для выражения своих концепций.

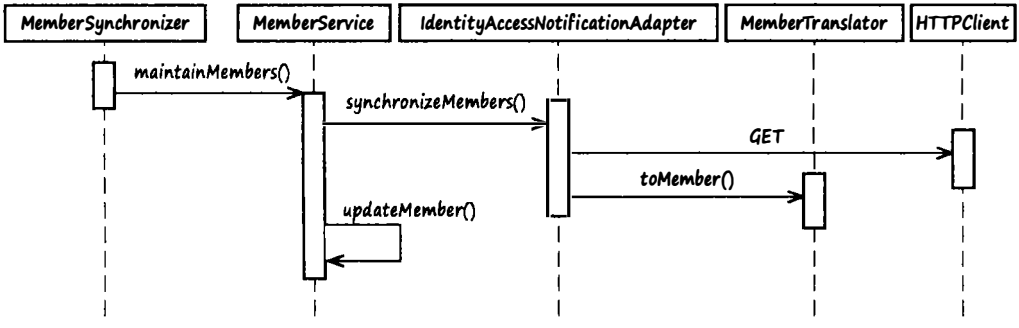


Рис. 3.9. Представление внутреннего функционирования КОНТЕКСТА УПРАВЛЕНИЯ ГИБКИМ ПРОЕКТИРОВАНИЕМ и ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ

Диаграммы и комментарии демонстрируют, как создавать документы, сопровождающие КАРТЫ КОНТЕКСТА. Они не должны быть обширными и в то же время обязаны предоставлять новым членам команды достаточно полную информацию. В общем, документ стоит писать, только если он полезен для команды.

Интеграция с контекстом сотрудничества

Теперь рассмотрим, как *контекст управления гибким проектированием* взаимодействует с *контекстом сотрудничества*. Здесь мы снова будем бороться за автономию, обсуждая интересные вопросы, связанные с обеспечением независимости системы.

Система ProjectOvation получает от системы CollabOvation дополнительные функции. В частности, к ним относятся форумы, посвященные объекту, и общие календари. Пользователи не должны взаимодействовать с системой CollabOvation непосредственно. Система ProjectOvation должна самостоятельно определять, доступны ли те или иные функциональные возможности для конкретного арендатора, и в случае положительного ответа создавать собственный вспомогательный ресурс в системе CollabOvation.

Рассмотрим раздел сценария использования *Создание продукта.*

Предусловие: существует возможность сотрудничества (средство куплено).

1. Пользователь предоставляет описательную информацию о **продукте**.
2. Пользователь выражает желание обсудить ее с командой.
3. Пользователь посылает запрос на создание указанного **продукта**.
4. Система создает **продукт**, а также **форум** и **обсуждение**.

Форум и **обсуждение** должны быть созданы в *контексте сотрудничества* от имени **продукта**. В противоположность этому вряд ли в *контексте идентификации и доступа* уже предусмотрен агент, определены пользователи, группы и роли и доступны сообщения об этих событиях. В *контексте сотрудничества* эти объекты уже существуют, но в *контексте управления гибким проектированием* требуются объекты, которые еще не существуют и не должны существовать, пока на них не поступит запрос. Это может мешать автономии, потому что при создании удаленного объекта мы зависим от доступности *контекста сотрудничества.* Стремясь к автономии, мы должны решить интересную проблему.

> **Почему *обсуждение* используется в обоих *контекстах***
>
> Возникла интересная ситуация, потому что одно и то же имя, *обсуждение*, появляется в обоих ОГРАНИЧЕННЫХ КОНТЕКСТАХ, имеющих разные типы, разные объекты, а значит, разное состояние и разное поведение.
>
> **Обсуждение** в *контексте сотрудничества* представляет собой АГРЕГАТ и управляет множеством **постов** — неявных наследников, которые сами являются АГРЕГАТАМИ. **Обсуждение** в *контексте управления гибким проектированием* — это ОБЪЕКТ-ЗНАЧЕНИЕ и лишь содержит ссылку на реальное **обсуждение** с **постами** во внешнем КОНТЕКСТЕ. Отметим, однако, что в главе 13 команда, выполняющая интеграцию, сталкивается с необходимостью строгой типизации разных видов **обсуждений** в *контексте управления гибким проектированием.*

Для поддержания согласованности необходимо использовать шаблоны **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (8)** и **СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА (4)**. Ничто не свидетельствует о том, что только удаленные системы могут получать уведомления, созданные нашей локальной системой. Когда в нашей

модели публикуется СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ ProductInitiated, оно обрабатывается нашей системой. Локальный обработчик отправляет запрос на создание удаленных **форума** и **обсуждения**. Это можно сделать либо с помощью вызова удаленных процедур, либо путем передачи сообщений, в зависимости от того, какой механизм поддерживает система CollabOvation. Если в данный момент механизм вызова удаленных процедур и системы удаленного сотрудничества недоступны, локальный обработчик будет вынужден просто периодически пытаться выполнить свою работу, пока она не завершится успешно. Если же передача сообщений поддерживается механизмом вызова удаленных процедур, то локальный обработчик пошлет сообщение системе сотрудничества. В свою очередь, система сотрудничества ответит своим сообщением, когда создание ресурса будет завершено. Когда обработчик СОБЫТИЯ в системе ProjectOvation получает это сообщение, он устанавливает ссылку идентификации в объекте класса `Product` на вновь созданное **обсуждение**.

Что происходит, если владелец продукта или члены команды пытаются использовать **обсуждение** до того, как оно появится? Рассматривается ли недоступное **обсуждение** как ошибка в модели? Сообщает ли в этом случае система о невыполнимом условии? Обратите внимание на то, что подписчики не обязаны платить за использование системы обеспечения сотрудничества. Это одна из нетехнических причин, по которым в проекте учитывается недоступность ресурса. Обойти проблемы итоговой согласованности, конечно, не получится. Это просто еще одно допустимое состояние, которое следует моделировать.

Элегантный способ решения этой задачи заключается в обработке всех возможных сценариев недоступности системы, чтобы они стали явными. Рассмотрим данный шаблон **СТАНДАРТНЫЙ ТИП (STANDARD TYPE)**, реализованный как шаблон **СОСТОЯНИЕ (STATE)** [Гамма и др.] и описанный в главе 6.

```
public enum DiscussionAvailability {
    ADD_ON_NOT_ENABLED, NOT_REQUESTED, REQUESTED, READY;
}

public final class Discussion implements Serializable {
    private DiscussionAvailability availability;
    private DiscussionDescriptor descriptor;
 ...

}

public class Product extends Entity {
 ...

private Discussion discussion;
 ...
}
```

Эта схема позволяет защитить ОБЪЕКТ-ЗНАЧЕНИЕ Discussion от неправильного использования, потому что его защищает СОСТОЯНИЕ, определенное экземпляром класса DiscussionAvailability. Когда кто-нибудь пытается принять участие в обсуждении класса Product, он может безопасно передать свое состояние обсуждения. Если это состояние — не READY, то участник получит одно из трех сообщений.

Для участия в командном сотрудничестве необходимо приобрести надстройку.

Владелец продукта не запрашивал создание обсуждения.

Настройка обсуждения еще не завершена; повторите попытку позднее.

Если состояние экземпляра Discussionavailability равно READY, вы получите возможность полноценного командного участия.

Как следует из первого сообщения о состоянии обсуждения, у команды существует возможность выборочного участия в сотрудничестве, даже если это участие не было оплачено. Это может оказаться правильным маркетинговым ходом, стимулирующим дальнейшие покупки. Кто сможет быстрее уговорить руководство приобрести надстройку, чем тот, кто каждый день получает уведомление о том, что он мог бы использовать ее, но пока не может? Очевидно, что доступность СОСТОЯНИЯ позволяет оценить не только технические преимущества.

В настоящий момент команда не уверена в том, какой на самом деле должна быть интеграция с системой сотрудничества. Для того чтобы понять схему обсуждения ЗАКАЗЧИК-ПОСТАВЩИК, она создала рис. 3.10. *Контекст управления гибким проектированием* может использовать второй ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ для интеграции с *контекстом сотрудничества*. Это похоже на то, как он использует *контекст идентификации и доступа*. На диаграмме показаны основные пограничные объекты, похожие на свои аналоги, используемые при интеграции с системой идентификации и доступа. На самом деле в системе нет ни одного объекта класса CollaborationAdapter. Это просто рамка для объекта, который будет нужен, но пока неизвестен.

Внутри локального КОНТЕКСТА показаны объекты классов Discussion-Service и SchedulingService. Они представляют собой СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, которые можно использовать для управления обсуждениями и календарными записями в системе сотрудничества. Реальные механизмы будут определены в ходе переговоров между командами по шаблону ЗАКАЗЧИК-ПОСТАВЩИК, описанному в главе 13.

Теперь команда может понять часть своей модели. Например, что произойдет, если будет создано обсуждение, а результат будет передан в локальный контекст. Асинхронный компонент — системы вызовов удаленных процедур или обработчик событий — передает методу attachDiscussion() класса Product новый

экземпляр ЗНАЧЕНИЯ Discussion. Все локальные агрегаты с отложенными удаленными ресурсами будут обрабатываться именно таким образом.

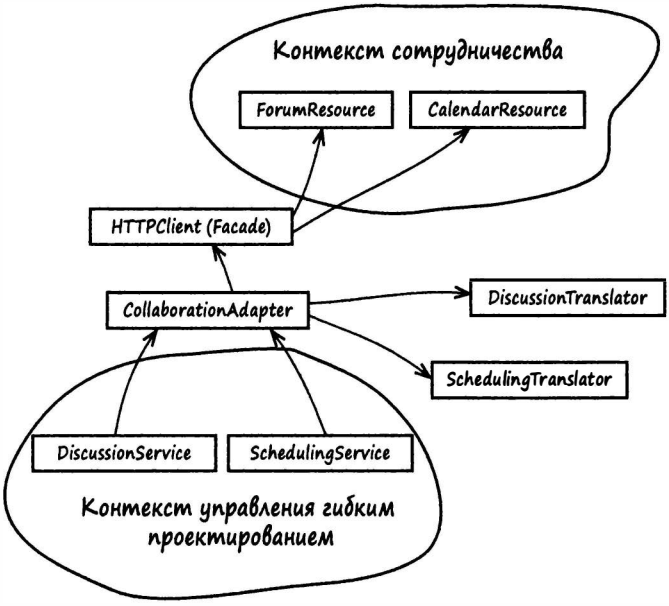


Рис. 3.10. Увеличение масштаба ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ и СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ, участвующих в интеграции между *контекстом сотрудничества* и *контекстом идентификации и доступа* в локальном КОНТЕКСТЕ

Эта проверка выявила интересные детали КАРТ КОНТЕКСТОВ. Однако мы должны ограничиться этим, потому что приближаемся к точке падения эффективности. Вероятно, можно было бы включить в систему **МОДУЛИ (9)**, но им посвящена отдельная глава. Включите в модель все релевантные высокоуровневые элементы, необходимые для обеспечения важных аспектов коммуникации между командами. С другой стороны, остановитесь, когда детали станут слишком подробными.

Нарисуйте КАРТУ КОНТЕКСТОВ, которую можно было бы повесить на стене. Ее можно загрузить в командную базу знаний, поскольку она касается не только команды. Постоянно обсуждайте проект, стараясь уточнить свою КАРТУ.

Резюме

Мы провели очень продуктивную работу над КАРТОЙ КОНТЕКСТОВ.

- Мы обсудили, что собой представляет КАРТА КОНТЕКСТОВ, чем она может помочь команде и как ее легче создать.
- Мы подробно рассмотрели три ОГРАНИЧЕННЫХ КОНТЕКСТА системы SaaSOvation и их вспомогательных КАРТ КОНТЕКСТОВ.
- Используя карты, мы увеличили масштаб интеграции между всеми КОНТЕКСТАМИ.
- Мы исследовали пограничные объекты, поддерживающие ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ и его взаимодействия.
- Мы увидели, как создать КАРТУ ТРАНСЛЯЦИИ, демонстрирующую локальное отображение между REST-ресурсом и соответствующим объектом в модели предметной области.

Не каждый проект требует такой детализации, как этот. Другие проекты могут потребовать еще большей проработки. Необходимо найти баланс между понятностью и практичностью и не слишком зарываться в детали на каждом уровне. Напоминаю, что мы не собираемся создавать очень подробную КАРТУ проекта. Она нужна лишь для того, чтобы во время обсуждений члены команды могли ссылаться на нее. Если отбросить условности и поставить во главу угла простоту и гибкость, то можно создать полезную КАРТУ КОНТЕКСТОВ, которая поможет нам идти вперед, а не увязнуть в болоте трудностей.

Глава 4

Архитектура

Архитектура должна говорить о своем времени и месте, но при этом стремиться к вечности.

Фрэнк Гери

Одним из больших преимуществ подхода DDD является то, что он не требует использования никакой конкретной архитектуры. Поскольку наше тщательно разработанное **СМЫСЛОВОЕ ЯДРО (2)** лежит в сердцевине **ОГРАНИЧЕННОГО КОНТЕКСТА (2)**, на все приложение или систему могут оказывать влияние один или несколько архитектурных стилей.[1] Одни модели предметной области могут испытывать сильное влияние окружающих ее архитектурных стилей, а другие могут соответствовать лишь конкретным требованиям. Наша цель состоит лишь в том, чтобы показать, как *делать правильный выбор и правильно сочетать архитектурные стили и шаблоны.*

К использованию тех или иных архитектурных стилей и шаблонов должны приводить реальные требования, предъявляемые к свойствам конкретного программного обеспечения. Сделав выбор, мы должны доказать, что он соответствует предъявляемым требованиям или даже превосходит их. Избегать злоупотребления архитектурными стилями и шаблонами не менее важно, чем использовать их правильно. Принятие архитектурных решений на основе реальных и естественных требований — это полезный подход, основанный на оценке рисков [Fairbanks]. Архитектуру следует использовать только для уменьшения риска сбоев программного обеспечения, а не для того, чтобы повысить этот риск из-за использования неправильно выбранных архитектурных стилей и шаблонов. Таким образом, мы должны обосновать каждое архитектурное решение или исключить его из нашей системы.

---

[1] Эта глава посвящена архитектурным стилям, архитектуре приложений и архитектурным шаблонам. Стиль описывает реализацию конкретной архитектуры, в то время как архитектурный шаблон объясняет, как решить ту или иную задачу в рамках архитектурного стиля, но при этом он шире, чем шаблон проектирования. Я предполагаю, что читатели не слишком глубоко понимают эту разницу, поэтому им достаточно знать, что подход DDD может лежать в основе многочисленных архитектурных решений, окружающих проект.

Способность обосновывать выбор любых архитектурных стилей и шаблонов ограничена доступными функциональными требованиями, например пользовательскими сценариями или историями, и даже сценариями, специфичными только для данной модели предметной области. Иначе говоря, невозможно определить необходимые качества программного обеспечения без учета функциональных требований. Отсутствие такой информации не позволяет принимать обоснованные архитектурные решения, а это, в свою очередь, означает, что в настоящее время при разработке программного обеспечения все еще используется архитектурный подход, основанный на пользовательских сценариях.

Назначение главы

- Проанализировать интервью с исполнительным директором компании SaaSOvation.
- Показать, как улучшить шаблон МНОГОУРОВНЕВАЯ АРХИТЕКТУРА с помощью архитектурных стилей DIP и ГЕКСАГОНАЛЬНЫЙ.
- Показать, что архитектурный стиль ГЕКСАГОНАЛЬНЫЙ может поддерживать СЕРВИСНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ и архитектурный стиль REST.
- Описать перспективу применения архитектурных стилей ФАБРИКА ДАННЫХ (DATA FABRIC), РАСПРЕДЕЛЕННЫЙ КЕШ ДЛЯ КОЛЛЕКТИВНОЙ РАБОТЫ (GRID-BASED DISTRIBUTED CACHE) и СОБЫТИЙНО-ОРИЕНТИРОВАННЫЙ (EVENT-DRIVEN).
- Описать полезную роль нового архитектурного шаблона CQRS в рамках подхода DDD.
- Продемонстрировать применение архитектурных стилей командой SaaSOvation.

Архитектура — не самый важный фактор

Рассматриваемые ниже архитектурные стили и шаблоны — это вовсе не набор самых важных инструментов, которые следует применять всюду, где только можно. Наоборот, их следует применять только там, где они снижают конкретный риск, и там, где без них вероятность неудачи проекта или сбоя системы возрастает.

Книга [Эванс] посвящена МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ. Исходя из этого, компания SaaSOvation сначала решила, что подход DDD окажется эффективным, только если будет использован этот известный шаблон. Командам потребовалось некоторое время, чтобы понять, что возможности подхода DDD намного превосходят возможности шаблона МНОГОУРОВНЕВАЯ АРХИТЕКТУРА, несмотря на то что он был самым популярным во время написания книги [Эванс].

Впрочем, МНОГОУРОВНЕВАЯ АРХИТЕКТУРА может оказаться полезной для выработки правильного решения. Однако мы не будем останавливаться на этом, поскольку рассмотрим более современные архитектурные стили и шаблоны, которые при необходимости можно дополнить. Это докажет универсальность и широкую применимость DDD.

Разумеется, компании SaaSOvation были нужны не все без исключения архитектурные стили, но его команды должны были принять самое правильное из доступных решений.

Интервью с успешным директором по информатизации

Для того чтобы объяснить, зачем нужен каждый из архитектурных стилей, рассматриваемых в главе, мы должны перенестись на десять лет вперед и взять интервью у директора по информатизации компании SaaSOvation. Первые шаги компании были неудачными, но архитектурное решение помогло ей постепенно достичь успеха. Итак, включим телевизор и посмотрим передачу *TechMoney*, которую ведет журналистка Мария Финанс-Ильмундо...

Мария: Сегодня вечером я беру эксклюзивное интервью у Митчелла Уильямса, директора по информатизации чрезвычайно успешной компании SaaSOvation. Мы продолжаем серию передач "Познай свои архитектурные стили". Сегодня мы узнаем, как выбрать правильный архитектурный стиль, который может принести долгосрочный успех. Добро пожаловать на шоу, Митчелл! Благодарю Вас, что пришли к нам.

Митчелл: Я рад снова с Вами встретиться, Мария. Это всегда приятно для меня.

Мария: Не могли бы Вы описать архитектурные решения, которые Вы принимали на первых этапах, и объяснить, почему Вы их использовали?

Митчелл: Конечно. Хотите верьте, хотите нет, сначала мы планировали разработать программное обеспечение для настольных систем. Наша команда проектировала настольную систему, которая хранила бы информацию в центральной базе данных. Для этого мы решили применить МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ.

Мария: Это имело смысл?

Митчелл: Да, мы думали, что это имело смысл, особенно потому что мы работали над изолированным прикладным уровнем, связанным с центральной базой данных. В рамках простого стиля "клиент/сервер" этот подход был бы вполне оправдан.

Мария: Однако вскоре все изменилось, не так ли?

Митчелл: Определенно. Мы объединили усилия с бизнес-партнером и решили продвигаться вперед с помощью модели подписки SaaS. Нам требовалось значительное финансирование для поддержки наших усилий, и мы нашли его. Мы решили на некоторое время отодвинуть наше приложение для управления гибким проектированием на задний план, пока не будет разработан комплект инструментов для обеспечения сотрудничества. Это давало двойное преимущество. Во-первых, мы вышли бы на развивающийся рынок инструментов для обеспечения сотрудничества, а во-вторых, у нас было бы естественное дополнение к приложению для управления проектами. Вы знаете, сотрудничество при проектировании программного обеспечения приносит результаты.

Мария: Интересно. Пока все это выглядит довольно обыденно. И куда эти решения вас привели?

Митчелл: Поскольку сложность программного обеспечения возросла, нам нужно было управлять качеством с помощью инструментов для модульного и функционального тестирования. Для этого мы применили МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ как бы наоборот, использовав принцип DIP (Dependency Inversion Principle — принцип инверсии зависимостей). Это было важным решением, потому что команда могла легко тестировать программное обеспечение, не обращая внимания на ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС и ИНФРАСТРУКТУРНЫЕ УРОВНИ и сосредоточившись на ПРИЛОЖЕНИИ и ПРЕДМЕТНОЙ ОБЛАСТИ. Фактически мы могли разрабатывать пользовательский интерфейс отдельно и отложить на какое-то время решения, касающиеся технологии хранения данных. На самом деле это не было большим отклонением от УРОВНЕЙ. Команде было очень удобно работать.

Мария: Ух ты, отключить пользовательский интерфейс *и* хранение данных! Выглядит рискованно. Насколько это было опасно?

Митчелл: Ну, на самом деле не очень. Оказалось, что тактические шаблоны предметно-ориентированного проектирования ничем нам не повредили. Поскольку мы использовали шаблон АГРЕГАТ и ХРАНИЛИЩЕ, то могли разрабатывать механизмы хранения данных в оперативной памяти на основе интерфейсов ХРАНИЛИЩА и заменить механизм постоянного хранения данных после изучения существующих возможностей.

Мария: Круто.

Митчелл: Точно.

Мария: И?

Митчелл: Бац! Все закрутилось. Мы поставили системы CollabOvation и ProjectOvation и получали прибыль на протяжении нескольких кварталов.

Мария: Дзинь-дзинь.[2]

Митчелл: Ясное дело. Затем возник бум мобильных устройств, и мы решили поддерживать мобильные устройства, помимо браузеров на настольных системах. Для этого мы решили использовать архитектуру REST. Подписчики стали спрашивать о таких вещах, как федеральная идентификационная информация и защита данных, а также о сложных инструментах управления проектированием и ресурсами времени. Затем новые инвесторы захотели увидеть отчеты о предпочтительных направлениях развития.

Мария: Удивительно. Но ведь бурно развивался не только рынок мобильных устройств. Как вы со всем этим справились?

Митчелл: Команда решила перейти на ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ, чтобы реализовать все эти дополнения. Они выяснили, что шаблон ПОРТЫ И АДАПТЕРЫ дал им возможность почти моментально привлечь новых клиентов. Это же относится и к новым типам ПОРТОВ, таким как новые механизмы постоянного хранения, например NoSQL, и к возможностям передачи сообщений. И все это называется одним словом — *облако*.

Мария: И вы уверены во всех этих модификациях?

Митчелл: Абсолютно.

Мария: Колоссально! Если вы справились со всем этим, то, вероятно, вы сделали отличный выбор, который способствовал вашему долгосрочному успеху.

Митчелл: Точно. Теперь каждый месяц к нам подключаются сотни новых клиентов. Мы добавили службу для переноса существующих данных с унаследованных инструментов для корпоративного сотрудничества в наше облако. Команда решила, что концентрация на сервис-ориентированной архитектуре позволила агрегировать эти данные с помощью инструмента Collection Aggregator компании Mule. Он может располагаться на границе службы, оставаясь в рамках ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ.

Мария: Ага, значит, вы внедрили сервис-ориентированную архитектуру не потому, что это звучит круто. Вы использовали ее, потому что это было целесообразно. Прекрасно. Во всей отрасли мы не видели таких хороших решений.

Митчелл: Да, Мария, и это наша долговременная стратегия. Это был план нашего успеха. Например, со временем мы добавили систему TrackOvation, наше программное обеспечение для отслеживания дефектов, которое было интегрировано с системой ProjectOvation. По мере расширения функциональных возможностей системы ProjectOvation ее пользовательский интерфейс становился все более и более сложным. С появлением новой прикладной команды и соответствующего ей события инструментальная панель ВЛАДЕЛЬЦА всех

[2] Имеется в виду звук кассового аппарата. — *Примеч. ред.*

Scrum-продуктов обновлялась и обнаруживались новые дефекты. Поскольку ВЛАДЕЛЬЦЫ ПРОДУКТОВ среди арендаторов-подписчиков имели разные представления, инструментальные панели становились все сложнее. И, естественно, мы должны были поддерживать мобильные устройства. Команда стала рассматривать возможность использования архитектурного шаблона CQRS.

Мария: CQRS? Да бросьте Митч, это же безрассудно. Не та ли эта неопределенность, о которой нам ничего не известно? Как насчет того, чтобы оторваться от проектировочной доски?

Митчелл: Нет, совсем нет. Поскольку команда имела весомую причину для применения CQRS, чтобы снизить трение между командами и запросами, она шла только вперед, никогда не оглядываясь назад.

Мария: Точно. Не происходило ли это в то время, когда ваши подписчики стали спрашивать у вас о функциональных возможностях, которых требовала распределенная обработка данных?

Митчелл: Да; если бы мы не сделали это тогда, то утонули бы в море сложности. Некоторые функциональные возможности требовали выполнения ряда распределенных процессов перед тем, как выдать ответ. Команда ProjectOvation не могла заставлять пользователей ждать выполнения потенциально долгосрочных задач и повышать риск простоев. Она применила полноценную СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ, развернув классический шаблон КАНАЛЫ И ФИЛЬТРЫ.

Мария: Но ведь это не конец вашего путешествия по минному полю? Насколько трудным оно было?

Митчелл: Нет, нет и нет. Все не так, как кажется. Однако, если у вас умная команда, то проход через минное поле может быть таким же легким, как прогулка по парку. На самом деле СОБЫТИЙНО-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА упрощает многие вопросы, связанные с расширением систем.

Мария: Да, это правда. Идем дальше. Ведь это была *очевидная* возможность. Мы переходим к самой интересной части этой истории. Вы знаете... [В глазах мелькают доллары.]

Митчелл: Наша архитектура допускает быстрое масштабирование и управление изменениями так хорошо, что компания RoaringCloud приобрела компанию SaaSOvation для... ну... в общем, это вся публичная информация.

Мария: Я сказала бы *очень* публичная. Цена по 50 центов за обычную акцию дала в сумме около трех миллиардов долларов. Это достойно обнародования.

Митчелл: У вас хорошая память на финансовые факты! Это было сильным стимулом для приобретения прав на интеграцию. Мы приобрели огромное

количество новых подписчиков, так что база пользователей стала подвергать инфраструктуру системы ProjectOvation серьезному давлению. Настало время перейти к распределенной и параллельной обработке данных с помощью шаблона КАНАЛЫ И ФИЛЬТРЫ. Это потребовало выполнения длительных процессов, которые иногда называют САГАМИ.

Мария: Прекрасно. Можете ли вы категорически утверждать, что это было увлекательно?

Митчелл: Действительно увлекательно, и даже больше.

Мария: Мне кажется, что эта история никогда не закончится. Вероятно, наиболее неожиданные и даже шокирующие главы этой долгой истории успеха еще впереди.

Митчелл: Вам виднее. Теперь, когда компания RoaringCloud заняла монопольное положение на рынке благодаря широкому спектру приложений, предлагаемых по подписке, и миллионам пользователей, правительство обратило на нас внимание и стало регулировать индустрию. Был принят новый закон, требующий от компании RoaringCloud отслеживать каждое изменение в проекте. На самом деле лучше всего было рассматривать эту спорную ситуацию как естественную часть модели предметной области и использовать шаблон ИСТОЧНИК СОБЫТИЙ.

Мария: Вы устояли! Это безумие. Я имею в виду, что это действительно чистое безумие.

Митчелл: Это действительно безумно интересная проблема.

Мария: Меня больше всего удивляет, что на протяжении всех этих лет в основе ваших приложений лежали DDD-модели. Совершенно очевидно, что подход DDD вам не повредил. Похоже, что у вас с ним не было трудностей.

Митчелл: На самом деле все было почти наоборот. Мы твердо уверены, что благодаря раннему выбору подхода DDD и затратам времени на его освоение мы справились с бизнес-ситуациями, которых не могли (и не хотели) избежать.

Мария: Хорошо, как я люблю говорить, "дзинь-дзинь!" Еще раз благодарю вас, Митчелл. Вы показали нам, как правильный выбор архитектуры может принести долгосрочный успех, прямо здесь на передаче "Познай свои архитектурные стили".

Митчелл: Пожалуйста, Мария. Спасибо за приглашение.

Разговор получился довольно путаным, но полезным. Он продемонстрировал нам, как архитектурные решения, рассматриваемые в следующих разделах, можно использовать в рамках подхода DDD и как выбрать правильный момент для их применения.

Уровни

Многие считают шаблон МНОГОУРОВНЕВАЯ АРХИТЕКТУРА [Buschmann et al.] "дедушкой" всех шаблонов. Он описывает многоярусные системы и поэтому широко используется с сети веб, а также в промышленных и настольных приложениях. В этом шаблоне проводится четкое разделение между концепциями приложения и системы в виде точно определенных уровней.

> Разбейте сложную программу на уровни. Внутри каждого уровня разработайте связную структуру, полагающуюся только на нижние уровни и зависящую только от них. Чтобы обеспечить связь с верхними уровнями, используйте стандартные архитектурные шаблоны. Сосредоточьте весь код, относящийся к предметной модели, в одном уровне, и изолируйте его от кода интерфейса пользователя, прикладных операций и инфраструктуры [Эванс, Справочник, с. 81].

На рис. 4.1 показаны уровни, характерные для DDD-приложения, использующего традиционную МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ. Здесь изолированное СМЫСЛОВОЕ ЯДРО занимает отдельный уровень. Над ним расположены **УРОВЕНЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (USER INTERFACE LAYER)** и **ПРИКЛАДНОЙ УРОВЕНЬ (APPLICATION LAYER)**. Еще ниже располагается **ИНФРАСТРУКТУРНЫЙ УРОВЕНЬ (INFRASTRUCTURE LAYER)**.

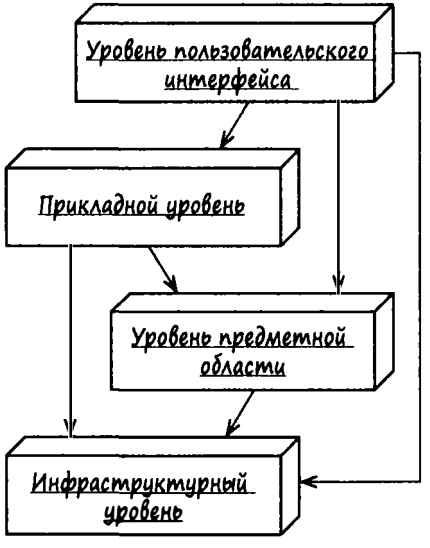


Рис. 4.1. Традиционная многоуровневая архитектура, в которой используется подход DDD

Важное правило этой архитектуры утверждает, что элементы каждого уровня могут связываться только с другими элементами этого же уровня и элементами нижележащих уровней. Между стилями этого шаблона существует небольшая разница. **АРХИТЕКТУРА СТРОГИХ УРОВНЕЙ (STRICT LAYERS ARCHITECTURE)** допускает связь с элементами только ближайшего нижележащего слоя. В то же время **АРХИТЕКТУРА НЕСТРОГИХ УРОВНЕЙ (RELAXED LAYERS ARCHITECTURE)** позволяет элементам любого вышележащего уровня связываться с элементами любых нижележащих уровней. Поскольку ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС и ПРИКЛАДНЫЕ СЛУЖБЫ часто требуют развертывания инфраструктуры, многие, если не подавляющее большинство, системы основаны на АРХИТЕКТУРЕ НЕСТРОГИХ УРОВНЕЙ.

На самом деле нижележащие уровни могут быть слабо связанными с вышележащими уровнями, но для этого используются исключительно такие механизмы, как **НАБЛЮДАТЕЛЬ (OBSERVER)** или **ПОСРЕДНИК (MEDIATOR)** [Гамма и др.]; от нижележащего уровня к вышележащему не существует прямых ссылок. Например, с помощью ПОСРЕДНИКА вышележащий уровень может реализовать интерфейс, определенный нижележащим уровнем, а затем передать реализующий объект нижележащему уровню как аргумент. Нижележащий уровень использует реализующий объект, не зная, где он находится с архитектурной точки зрения.

ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС должен содержать только код, предназначенный для реализации пользовательского представления и запросов. Он не должен содержать операций, связанных с логикой предметной области или бизнес-логикой. Поскольку ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС требует авторизации, может показаться, что он должен содержать операции бизнес-логики. Однако авторизация, предусмотренная ПОЛЬЗОВАТЕЛЬСКИМ ИНТЕРФЕЙСОМ, не относится исключительно к модели предметной области. Как указано в описании шаблона **СУЩНОСТИ (5)**, мы хотим ограничить рамками модели крупномодульную авторизацию, выражающую глубокие знания о предметной области.

Если в компонентах ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА используются объекты из модели предметной области, то все обычно ограничивается обновлением данных на экране. Если принять такой подход, то для изоляции представления от объектов предметной области можно использовать **МОДЕЛЬ ПРЕЗЕНТАЦИИ (PRESENTATION MODEL) (14)**.

Поскольку пользователем может быть как человек, так и другие системы, иногда этот уровень предоставляет средства для удаленного вызова служб API в форме **СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ (13)**.

Компоненты ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА являются непосредственными клиентами ПРИКЛАДНОГО УРОВНЯ.

ПРИКЛАДНЫЕ СЛУЖБЫ (14) находятся на ПРИКЛАДНОМ УРОВНЕ. Этим они отличаются от **СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ (7)** и поэтому должны избегать

логики предметной области. Они могут управлять хранением данных и вопросами безопасности. Кроме того, они могут нести ответственность за рассылку уведомлений о событиях другим системам и/или за составление электронных сообщений, рассылаемых пользователям. ПРИКЛАДНЫЕ СЛУЖБЫ на этом уровне являются прямыми клиентами модели предметной области, хотя сами они не обладают бизнес-логикой. Они остаются очень легковесными, координируя операции, выполняемые над объектами предметной области, такими как **АГРЕГАТЫ (10)**. Они являются основными средствами выражения пользовательских сценариев или пользовательских историй в рамках модели. Следовательно, основная функция ПРИКЛАДНОЙ СЛУЖБЫ сводится к получению параметров от ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА, использованию **ХРАНИЛИЩА (12)** для получения экземпляра АГРЕГАТА, а затем к выполнению операции командной строки над ним.

```
@Transactional
public void commitBacklogItemToSprint(
   String aTenantId, String aBacklogItemId, String aSprintId) {
   TenantId tenantId = new TenantId(aTenantId);

   BacklogItem backlogItem =
       backlogItemRepository.backlogItemOfId(
              tenantId, new BacklogItemId(aBacklogItemId));

   Sprint sprint = sprintRepository.sprintOfId(
               tenantId, new SprintId(aSprintId));

   backlogItem.commitTo(sprint);
}
```

Если ПРИКЛАДНЫЕ СЛУЖБЫ становятся еще сложнее, это, возможно, свидетельствует о том, что в ПРИКЛАДНЫЕ СЛУЖБЫ проникли операции логики предметной области, и эта модель становится анемичной. Следовательно, лучше всего оставлять этих клиентов модели как можно более тонкими. Если требуется создать новый АГРЕГАТ, то ПРИКЛАДНАЯ СЛУЖБА должна использовать **ФАБРИКУ (FACTORY) (11)** или создать объект с помощью конструктора АГРЕГАТА, а затем использовать ХРАНИЛИЩЕ для его хранения. ПРИКЛАДНАЯ СЛУЖБА может также использовать СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ для выполнения задач, специфичных для предметной области, определенной как операция без фиксации состояния.

Если модель предметной области должна публиковать **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS) (8)**, то ПРИКЛАДНОЙ УРОВЕНЬ может регистрировать подписчиков на любое количество СОБЫТИЙ. Это позволяет хранить, пересылать и обрабатывать СОБЫТИЯ в качестве одной из основных функций приложения. Это расширяет модель предметной области и помогает **ИЗДАТЕЛЮ СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENT PUBLISHER) (8)** обеспечивать простоту и независимость модели от инфраструктуры передачи сообщений.

Поскольку модель предметной области, реализующая все операции, подробно описывается в других главах, мы не будем здесь повторяться. Тем не менее существует ряд проблем, связанных с предметной областью и традиционной МНОГОУРОВНЕВОЙ АРХИТЕКТУРОЙ. Для использования этого шаблона может потребоваться УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ, ограничивающий использование ИНФРАСТРУКТУРЫ. Это не значит, что все должны делать объекты предметной области. Нет, этого как раз следует избегать. Однако в соответствии с определением УРОВНЕЙ некоторые интерфейсы, возможно, придется реализовать на УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ, который зависит от технологий, обеспечиваемых ИНФРАСТРУКТУРОЙ.

Например, интерфейсы ХРАНИЛИЩА требуют реализаций, использующих такие компоненты, как механизмы постоянного хранения данных, включенные в ИНФРАСТРУКТУРУ. Что если мы просто реализуем интерфейсы ХРАНИЛИЩА в ИНФРАСТРУКТУРЕ? Поскольку ИНФРАСТРУКТУРНЫЙ УРОВЕНЬ расположен ниже УРОВНЯ ПРЕДМЕТНОЙ ОБЛАСТИ, ссылки от ИНФРАСТРУКТУРЫ вверх в ПРЕДМЕТНУЮ ОБЛАСТЬ нарушают правила МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ. Однако это не значит, что основные объекты предметной области должны быть связаны с ИНФРАСТРУКТУРОЙ. Для того чтобы избежать нарушения правил, мы должны использовать реализацию **МОДУЛЕЙ (9)**, скрывающую вспомогательные классы.

```
com.saasovation.agilepm.domain.model.product.impl
```

Как указано в описании **МОДУЛЕЙ (9)**, класс `MongoProductRepository` можно разместить в этом же пакете. Впрочем, это не единственный способ решения данной проблемы. В качестве альтернативы мы могли бы реализовать такие интерфейсы на ПРИКЛАДНОМ УРОВНЕ, что соответствует правилам МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ. Схема этого подхода представлена на рис. 4.2. И все же это решение довольно безвкусно.

Как указано ниже в разделе "Принцип инверсии зависимостей", существует лучшее решение.

В традиционной МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ ИНФРАСТРУКТУРА находится на нижнем уровне. Сущности вроде механизмов постоянного хранения данных и передачи сообщений расположены там же. Среди сообщений могут быть сообщения, отосланные промежуточными системами управления сообщениями на предприятии (enterprise messaging middleware systems), или более привычные электронные письма (SMTP) и текстовые сообщения (SMS). Подумайте обо всех этих технических компонентах и каркасах, обеспечивающих приложению услуги нижнего уровня. Обычно они рассматриваются как часть ИНФРАСТРУКТУРЫ. Более высокие УРОВНИ связываются с низкоуровневыми компонентами, чтобы обеспечить повторное использование технических средств. При этом мы снова отвергаем идею связать объекты модели предметной области с ИНФРАСТРУКТУРОЙ.

Рис. 4.2. ПРИКЛАДНОЙ УРОВЕНЬ **может содержать техническую реализацию интерфейсов, определенных на** УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ

Команды SaaSOvation заметили, что расположение ИНФРАСТРУКТУРНОГО УРОВНЯ **на дне создает определенные неудобства. Например, затрудняется реализация технических аспектов с ориентацией на** УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ**, поскольку при этом нарушаются правила** МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ**. И действительно, такой код трудно тестировать. Как же устранить эти недостатки?**

Нельзя ли сделать на скорую руку нечто более удобное, уточнив порядок УРОВНЕЙ?

Принцип инверсии зависимостей

Существует способ улучшить традиционную МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ, уточнив зависимости. ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТЕЙ (Dependency Inversion Principle — DIP) был сформулирован Робертом Мартином (Robert C. Martin) и описан в работе [Martin, DIP]. Его формальное определение звучит следующим образом.

> Модули высокого уровня не должны зависеть от модулей низкого уровня. Модули обоих видов должны зависеть от абстракций.

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Смысл этого определения заключается в том, что компонент, обеспечивающий услуги нижнего уровня (в нашем случае — ИНФРАСТУКТУРА), должен зависеть от интерфейсов, определенных компонентами высокого уровня (в нашем случае — от ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА, ПРИЛОЖЕНИЯ и ПРЕДМЕТНОЙ ОБЛАСТИ). Хотя существует несколько способов выражения архитектуры, использующей принцип DIP, мы остановимся на структуре, изображенной на рис. 4.3.

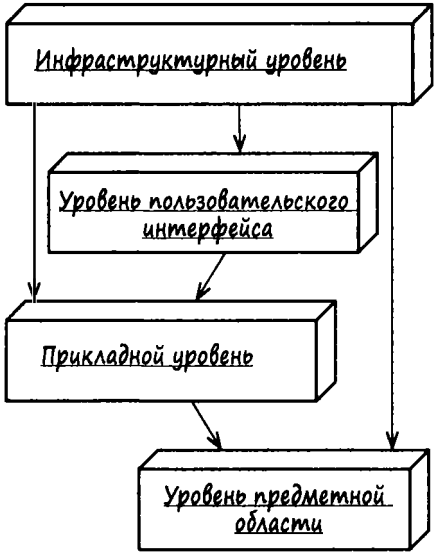


Рис. 4.3. Возможные УРОВНИ, возникающие при использовании принципа DIP. Мы перенесли ИНФРАСТРУКТУРНЫЙ УРОВЕНЬ на самый верх, позволив ему реализовывать интерфейсы для всех нижележащих УРОВНЕЙ

Действительно ли принцип DIP относится ко всем этим уровням

Некоторые читатели могут прийти к выводу, что принцип DIP относится только к двум уровням — самому верхнему и самому нижнему. Самый верхний уровень может реализовывать абстракции интерфейса, определенные на самом нижнем уровне. Уточнив рис. 4.3 с учетом этого факта, мы перенесли бы ИНФРАСТРУКТУРНЫЙ УРОВЕНЬ вверх, а УРОВЕНЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА, ПРИКЛАДНОЙ УРОВЕНЬ и УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ — вниз. Вы можете принимать или не принимать такую архитектуру, основанную на принципе DIP. Это не имеет значения; все эти вопросы снимает **ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА (HEXAGONAL ARCHITECTURE)** [Cockburn] или **АРХИТЕКТУРА ПОРТОВ И АДАПТЕРОВ (PORTS AND ADAPTERS ARCHITECTURE)**.

Как следует из рис. 4.3, мы должны иметь ХРАНИЛИЩЕ, реализованное в ИНФРАСТРУКТУРЕ для интерфейса, определенного в ПРЕДМЕТНОЙ ОБЛАСТИ.

```
package com.saasovation.agilepm.infrastructure.persistence;

import com.saasovation.agilepm.domain.model.product.*;

public class HibernateBacklogItemRepository
    implements BacklogItemRepository {
    ...
    @Override
    @SuppressWarnings("unchecked")
    public Collection<BacklogItem> allBacklogItemsComittedTo(
        Tenant aTenant, SprintId aSprintId) {
    Query query =
        this.session().createQuery(
             "from -BacklogItem as _obj_ "
              + "where _obj_.tenant = ? and _obj_.sprintId = ?");
```

```
        query.setParameter(0, aTenant);
        query.setParameter(1, aSprintId);

        return (Collection<BacklogItem>) query.list();
    }
    ...
}
```

На УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ использование принципа DIP позволяет устанавливать зависимость ПРЕДМЕТНОЙ ОБЛАСТИ и ИНФРАСТРУКТУРЫ от абстракций (интерфейсов, определенных в модели предметной области). Поскольку ПРИКЛАДНОЙ УРОВЕНЬ является непосредственным клиентом ПРЕДМЕТНОЙ ОБЛАСТИ, он зависит от интерфейсов ПРЕДМЕТНОЙ ОБЛАСТИ и имеет косвенный доступ к классам реализации СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, предоставленным ИНФРАСТРУКТУРОЙ. Он может использовать любую форму владения этими реализациями, включая **ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DEPENDENCY INJECTION)**, **ФАБРИКА СЛУЖБ (SERVICE FACTORY)** и **ДОПОЛНИТЕЛЬНЫЙ МОДУЛЬ (PLUG IN)** [Fowler, P of EAA]. В примерах, приведенных в книге, используется шаблон ВНЕДРЕНИЕ ЗАВИСИМОСТИ, реализуемый с помощью каркаса Spring Framework, а иногда — шаблон ФАБРИКА СЛУЖБ, реализуемый классом `DomainRegistry`. Фактически класс `DomainRegistry` использует каркас Spring для поиска ссылок на компоненты, реализующие интерфейсы, определенные моделью предметной области, включая ХРАНИЛИЩА и СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ.

Довольно интересно, что, размышляя о влиянии принципа DIP на эту архитектуру, можно прийти к выводу, что уровней больше нет вообще. Все верхние и нижние уровни зависят только от абстракций и напоминают перевернутый стек.

А что если действительно попробовать оставить архитектуру в таком виде и сделать ее немного симметричнее? Посмотрим, что из этого может получиться.

Гексагональная архитектура, или Архитектура портов и адаптеров

ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРОЙ (HEXAGONAL ARCHITECTURE)[3] Элистер Кокберн (Alistair Cockburn) назвал стиль, предназначенный для создания симметрии [Cockburn]. Она достигает этой цели, разрешая множеству разнообразных клиентов взаимодействовать с системой на равной основе. Нужен новый клиент? Не проблема. Просто добавьте АДАПТЕР для преобразования любой входной информации клиента в данные, понятные для внутреннего интерфейса прикладного программирования. В то же время механизмы вывода, развернутые в системе, такие как средства графической визуализации, постоянного хранения данных и передачи сообщений, могут быть разнообразными и взаимозаменяемыми. Это возможно благодаря АДАПТЕРУ, созданному для преобразования результатов работы приложения в форму, приемлемую для конкретного механизма вывода.

Когда мы обсудим все вопросы, вы согласитесь, что эта архитектура имеет все шансы стать вечной.

В настоящее время многие команды, утверждающие, что они используют МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ, на самом деле используют ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ. Частично это объясняется количеством проектов, в которых существует та или иная форма ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ. Это не значит, что ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ автоматически создает ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ. Просто она стимулирует создание архитектуры, которая естественным образом приводит к использованию ПОРТОВ И АДАПТЕРОВ. В любом случае этот вопрос можно прояснить путем более глубоких размышлений.

Обычно место, в котором клиенты взаимодействуют с системой, называют "внешним интерфейсом" ("front end"). Аналогично место, откуда приложение извлекает постоянно хранящиеся данные, где оно сохраняет постоянно хранящиеся данные или пересылает выходную информацию, называется "внутренним интерфейсом" ("back end"). Однако гексагональная архитектура вводит другую классификацию частей системы, как показано на рис. 4.4. Существуют две основные

[3] Мы называем эту архитектуру ГЕКСАГОНАЛЬНОЙ, хотя ее все чаще называют ПОРТЫ И АДАПТЕРЫ. Несмотря на переименование, сообщество разработчиков все еще называет ее ГЕКСАГОНАЛЬНОЙ. Появилась также Луковая архитектура (Onion Architecture). Однако похоже, что ЛУКОВАЯ — это (неудачный) синоним ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ. Мы можем вполне обоснованно считать, что все эти стили совпадают, и придерживаться определения, данного Кокберном [Cockburn].

области — *внешняя* и *внутренняя*. Внешняя область позволяет разнообразным клиентам вводить данные и предоставляет механизмы для извлечения постоянно хранящихся данных, сохранять результаты работы приложения (например, в базе данных) или посылать их куда-нибудь (например, посылать сообщения).

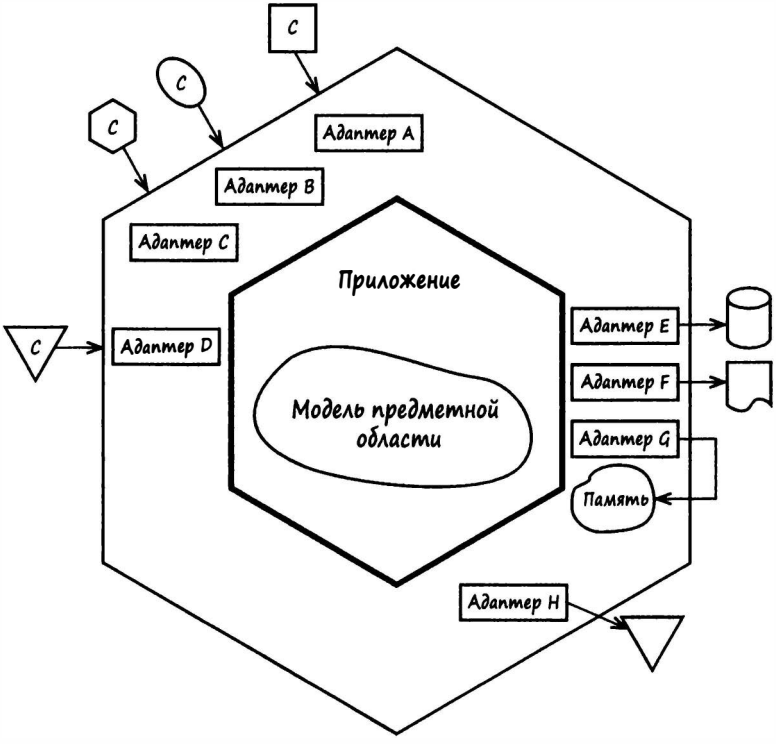


Рис. 4.4. ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА называется также ПОРТЫ И АДАПТЕРЫ. Для каждого из *внешних* типов данных существует отдельный АДАПТЕР. *Внешний* тип данных достигает *внутреннего* с помощью интерфейса прикладного программирования

Ковбойская логика

AJ: Моим лошадям, определенно, нравится новый гексагональный загон. У них теперь больше углов, куда они могут стать, когда я их седлаю.

На рис. 4.4 каждому типу клиентов соответствует свой АДАПТЕР [Гамма и др.], трансформирующий протоколы ввода в информацию, совместимую с

внутренним интерфейсом приложения. Каждая сторона шестиугольника соответствует разным видам ПОРТОВ — ввода или вывода. Три запроса клиентов поступают через ПОРТ ввода (АДАПТЕРЫ A, B и C), а один использует другой вид ПОРТА (АДАПТЕР D). Возможно, эти три клиента используют протокол HTTP (браузер, REST, SOAP и т.п.), а один использует протокол AMQP (например, на платформе RabbitMQ). Строгого определения, что такое ПОРТ, не существует и это обеспечивает гибкость концепции. Независимо от разновидности ПОРТА, на него поступают запросы клиента, а АДАПТЕР преобразовывает их входную информацию. Затем он вызывает операцию приложения или посылает приложению событие. Тем самым управление передается во внутреннюю часть системы.

Чаще всего порты создаются другими

Обычно нам не приходится создавать ПОРТЫ. ПОРТ можно представить как протокол HTTP, а АДАПТЕР — как Java Servlet или класс, аннотированный JAX-RS и получающий вызовы метода от контейнера (JEE) или каркаса (RESTEasy или Jersey). Кроме того, можно создать слушателя сообщений для каркаса NServiceBus или платформы RabbitMQ. В этом случае с некоторой натяжкой ПОРТ можно считать механизмом передачи сообщений, а АДАПТЕР — слушателем сообщений, потому что именно слушатель сообщений должен извлекать данные из сообщения и транслировать их в параметры, приемлемые для передачи во внутренний интерфейс API приложения (клиенту модели предметной области).

Разрабатывайте приложение, руководствуясь функциональными требованиями

Используя ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ, мы разрабатываем приложение с учетом своих сценариев использования, а не сценариев многочисленных поддерживаемых клиентов. Любое количество клиентов любого типа может обращаться к разным ПОРТАМ, но при этом каждый АДАПТЕР делегирует их приложению, используя один и тот же внутренний интерфейс API.

Приложение получает запросы с помощью своего открытого интерфейса API. Границы приложения, или внутренний шестиугольник, также представляют собой границу сценария использования (или пользовательские истории). Иначе говоря, мы должны создавать сценарии использования, основываясь на функциональных требованиях приложения, а не на требованиях многочисленных и разнообразных клиентов или механизмов вывода. Когда приложение получает запрос через свой интерфейс API, оно использует модель предметной области для выполнения всех запросов, связанных с реализацией бизнес-логики. Таким образом, интерфейс API приложения публикуется как множество СЛУЖБ ПРИЛОЖЕНИЯ. Здесь СЛУЖБЫ ПРИЛОЖЕНИЯ также являются непосредственными клиентами модели предметной области, как и в МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ.

Приведенный ниже код описывает ресурс RESTful, опубликованный с помощью каркаса JAX-RS. Запрос поступает через входной ПОРТ протокола HTTP, а его обработчик действует как АДАПТЕР, делегируя обработку СЛУЖБЕ ПРИЛОЖЕНИЯ.

```
@Path("/tenants/{tenantId}/products")
public class ProductResource extends Resource {

    private ProductService productService;
    ...
    @GET
    @Path("{productId}")
    @Produces({ "application/vnd.saasovation.projectovation+xml" })
    public Product getProduct(
            @PathParam("tenantId") String aTenantId,
            @PathParam("productId") String aProductId,
            @Context Request aRequest) {

      Product product = productService.product(aTenantId, aProductId);

      if (product == null) {
          throw new WebApplicationException(
                  Response.Status.NOT_FOUND);
      }

      return product; // сериализован в XML с помощью MessageBodyWriter
    }
    ...

}
```

Разные аннотации JAX-RS составляют значительную часть АДАПТЕРА, разделяя путь к ресурсу на лексемы и превращая его параметры в экземпляры класса `String`. Для делегирования внутрь приложения этот запрос внедряет и использует экземпляр класса `ProductService`. Объект класса `Product` сериализуется в формат XML и размещается в объекте класса `Response`, который затем передается через выходной ПОРТ протокола HTTP.

Каркас JAX-RS нас не интересует

Это просто способ использования приложения и внутренней модели предметной области. По существу, каркас JAX-RS не важен. Мы могли бы использовать пакет Restful или создать сервер Node.js, запускающий модуль типа restify. Более того, АДАПТЕРЫ, разработанные для обработки входной информации, поступающей от других ПОРТОВ, могут делегировать ее с помощью одного и того же интерфейса API, как мы вскоре увидим.

Что можно сказать о правой части приложения? Будем рассматривать реализации ХРАНИЛИЩА в виде АДАПТЕРОВ постоянного хранения, обеспечивающих доступ к ранее сохраненным экземплярам АГРЕГАТА и сохраняющих его новые

экземпляры. Как показано на диаграмме (АДАПТЕРЫ E, F и G), мы можем иметь реализации ХРАНИЛИЩА для реляционных баз данных, документов, распределенных кешей, а также хранилища в оперативной памяти. Если приложение посылает СООБЩЕНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ во внешнюю среду, оно может использовать другой АДАПТЕР (H) для передачи сообщения. АДАПТЕР для передачи выходных сообщений противоположен по смыслу АДАПТЕРУ ввода, который поддерживает протокол AMQP и поэтому использует ПОРТ, который отличается от ПОРТА, используемого для постоянного хранения данных.

Большое преимущество ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ заключается в том, что АДАПТЕРЫ легко поддаются тестированию. Все приложение и модель предметной области можно разрабатывать и тестировать до того, как появятся клиенты и механизмы хранения данных. Можно создавать тесты для выполнения методов класса `ProductService` до принятия каких-либо решений о поддержке протоколов HTTP/REST, SOAP или ПОРТОВ, рассылающих сообщения. Любое количество тестовых клиентов можно создать еще до того, как будет завершен макет пользовательского интерфейса. Задолго до того как будет выбран механизм постоянного хранения данных, в памяти можно развернуть ХРАНИЛИЩА для имитации постоянного хранения во время тестирования. Детали такой реализации изложены в описании **ХРАНИЛИЩ (12)**. Таким образом, можно достичь значительного прогресса в разработке ядра независимо от вспомогательных технических компонентов.

Если вы используете настоящую МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ, рассмотрите преимущества переворота структуры и альтернативной разработки на основе ПОРТОВ и АДАПТЕРОВ. При правильном проектировании внутренний шестиугольник — приложение и модель предметной области — никак не повлияет на внешние части. Это создаст четкие границы, внутри которых будет реализован каждый сценарий использования. Помимо произвольного количества клиентов, АДАПТЕР может поддерживать многочисленные автоматизированные тесты и реальные клиенты, а также механизмы хранения, рассылки сообщений и другие средства вывода.

Когда команды SaaSOvation рассмотрели преимущества использования ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ, они решили отказаться от МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ. Это оказалось не трудно. Для этого потребовалось лишь принять немного другие принципы с учетом знакомого каркаса Spring Framework.

Поскольку ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА такая разносторонняя, она может стать основой для поддержки других архитектурных стилей, требующихся в системе. Например, мы могли бы применить **СЕРВИС-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ (SERVICE-ORIENTED ARCHITECTURE)**, **REST** или **СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ (EVENT-DRIVEN ARCHITECTURE)**; применить принцип CQRS; использовать **ФАБРИКУ ДАННЫХ (DATA FABRIC)** или **РАСПРЕДЕЛЕННЫЙ КЕШ ДЛЯ КОЛЛЕКТИВНОЙ РАБОТЫ (GRID-BASED DISTRIBUTED CACHE)**; или придерживаться распределенной и параллельной разработки по модели Map-Reduce. Большинство из этих возможностей мы обсудим в этой главе позже. ГЕКСАГОНАЛЬНЫЙ стиль формирует строгую основу для поддержки любых дополнительных архитектурных стилей. Существуют и другие способы, но *в оставшейся части главы в основе наших рассуждений будут лежать ПОРТЫ И АДАПТЕРЫ.*

Сервис-ориентированная архитектура

СЕРВИС-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА (SERVICE-ORIENTED ARCHITECTURE — SOA) для разных людей имеет разный смысл. Это несколько затрудняет ее обсуждение. Лучше всего сначала попытаться найти общий знаменатель или по крайней мере определить основу для дискуссии. Рассмотрим несколько принципов SOA, сформулированных Томасом Эрлом (Thomas Erl) [Erl]. Кроме того что службы всегда взаимодействуют друг с другом, они еще и подчиняются восьми принципам проектирования, представленным в табл. 4.1.

Таблица 4.1. Принципы проектирования служб

Принцип	Описание
1. Контракт службы	Службы выражают свои цели и возможности посредством контракта, сформулированного в одном или нескольких описаниях
2. Слабая связанность службы	Службы минимизируют зависимость и лишь всего лишь знают о существовании друг друга
3. Абстракция службы	Службы открывают только свои контракты и скрывают внутреннюю логику от клиентов
4. Повторное использование служб	Службы можно использовать повторно и в другом месте, чтобы создать более крупномодульные службы
5. Автономность службы	Службы контролируют свою среду и ресурсы, чтобы оставаться независимыми. Это позволяет им оставаться согласованными и надежными

Окончание табл. 4.1

Принцип	Описание
6. Отсутствие собственного состояния службы	Службы возлагают ответственность за управление состоянием на своих клиентов, чтобы не создавать противоречия с принципом автономии служб
7. Служба должна быть обнаруживаемой	Службы описываются с помощью метаданных, позволяющих обнаруживать и понимать их контракты. Это позволяет обеспечить их повторное использование
8. Служба должна допускать возможность композиции	Службы могут включаться в более крупномодульные службы независимо от размера и сложности композиции

Эти принципы можно сочетать с ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРОЙ, перенеся границу службы далеко налево, а модель предметной области — в середину. Основная архитектура представлена на рис. 4.5. Здесь клиенты обращаются к службам с помощью служб REST, SOAP и обмена сообщениями. Отметим, что гексагональная архитектура поддерживает множество терминалов технических служб. Это отражается на использовании принципов DDD в архитектуре SOA.

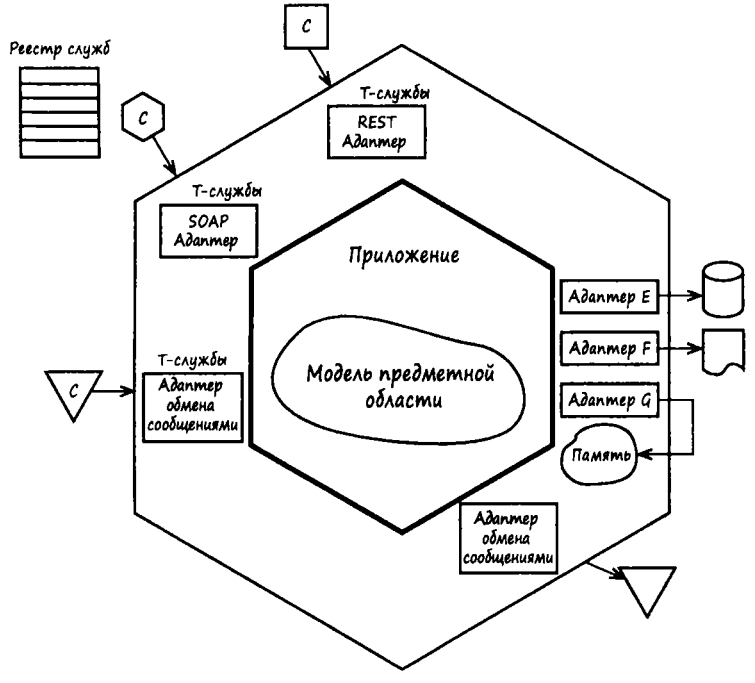


Рис. 4.5. ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА, поддерживающая принципы SOA и службы REST, SOAP и обмена сообщениями

Поскольку существует широкий спектр мнений об архитектуре SOA и ее ценности, нет ничего удивительного, если вы не согласитесь с представленной здесь схемой. Мартин Фаулер (Martin Fowler) назвал эту ситуацию "сервисно-ориентированной неоднозначностью" [Fowler, SOA]. Я не хотел бы здесь предпринимать геройскую попытку устранить неоднозначность архитектуры SOA. Однако я опишу перспективу включения принципов DDD в множество *приоритетов*, заявленных в Манифесте SOA (SOA Manifesto).[4]

Во-первых, рассмотрение прагматичных принципов, сформулированных одним из авторов манифеста [Tilkov, Manifesto], формирует важный контекст. Комментируя манифест, он по крайней мере приблизил нас на один или два шага к пониманию того, какими могут быть службы SOA.

> [Манифест] позволил мне представить службу как набор интерфейсов SOAP/WSDL или коллекцию ресурсов RESTful... Это не попытка определения — это попытка выяснить, какие выгоды мы можем получить, если примем его.

Комментарий Стефана заслуживает внимания. Поиск согласия всегда полезен, и вы, возможно, согласитесь с тем, что бизнес-службы могут быть оснащены любым количеством технических служб.

Технические службы могут быть ресурсами RESTful, интерфейсами SOAP или механизмами передачи сообщений. Бизнес-служба помогает реализовать *бизнес-стратегию*, т.е. способ ведения бизнеса на основе технологии. Однако определение отдельной бизнес-службы не означает определение отдельной **ПОДОБЛАСТИ (2)** или ОГРАНИЧЕННОГО КОНТЕКСТА. Несомненно, оценивая пространства задач и решений, мы обнаружим, что бизнес-служба состоит из множества служб. Таким образом, на рис. 4.5 показана архитектура только отдельного ОГРАНИЧЕННОГО КОНТЕКСТА, который может содержать набор технических служб, реализованных на основе многочисленных ресурсов RESTful, интерфейсов SOAP или механизмов передачи сообщений, и который представляет собой лишь часть общей бизнес-службы. В пространстве решений SOA можно увидеть много ОГРАНИЧЕННЫХ КОНТЕКСТОВ, в любом из которых может использоваться ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА или другой стиль. Ни архитектура SOA, ни подход DDD не обязаны регламентировать, как именно должен быть спроектирован и развернут каждый набор технических служб, что открывает широкий выбор вариантов.

Тем не менее при использовании принципов DDD наша цель заключается в создании ОГРАНИЧЕННОГО КОНТЕКСТА с полной, лингвистически точно определенной моделью предметной области. Как указано в описании **ОГРАНИЧЕННЫХ**

[4] Манифест SOA сам по себе стал предметом острой критики, тем не менее мы смогли извлечь из него кое-какую пользу.

КОНТЕКСТОВ (2), мы не хотим, чтобы архитектура оказывала влияние на размер модели предметной области. Это может произойти, если один или несколько терминалов технических служб, таких как отдельный ресурс REST, отдельный интерфейс SOAP или механизм передачи системных сообщений, повлияют на размер ОГРАНИЧЕННОГО КОНТЕКСТА. Это может привести к созданию множества очень маленьких ОГРАНИЧЕННЫХ КОНТЕКСТОВ и моделей предметных областей, каждая из которых может состоять только из одной СУЩНОСТИ, действующей как КОРЕНЬ отдельного маленького АГРЕГАТА. В результате на отдельном предприятии могут появиться сотни миниатюрных ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Хотя этот подход может иметь технические преимущества, он не всегда согласуется со стратегическими целями подхода DDD. Он препятствует разработке ясной и хорошо определенной модели предметной области, основанной на полном и непротиворечивом **ЕДИНОМ ЯЗЫКЕ (1)**, на самом деле фрагментируя этот ЯЗЫК. В соответствии с Манифестом SOA неестественная фрагментация ОГРАНИЧЕННОГО КОНТЕКСТА не всегда соответствует принципам SOA, приведенным ниже.

1. Превалирование **бизнес-ценности** над технической стратегией.
2. Превалирование **стратегических целей** над выгодами, специфичными для проекта.

Эти принципы можно согласовать со стратегическими целями DDD. Как объясняется в описании **ОГРАНИЧЕННЫХ КОНТЕКСТОВ (2)**, важность архитектуры технических компонентов при дроблении моделей снижается.

Команды SaaSOvation получили трудный и важный урок: учет лингвистических аспектов лучше соответствует принципам DDD. Каждый из трех ОГРАНИЧЕННЫХ КОНТЕКСТОВ, разработанных ими, отражает цели архитектуры SOA — как с точки зрения бизнес-служб, так и с точки зрения технических служб.

Каждая из трех учебных моделей, рассмотренных при описании **ОГРАНИЧЕННЫХ КОНТЕКСТОВ (2)**, **КАРТ КОНТЕКСТА (3)** и **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕСТОВ (13)**, представляет отдельную лингвистически хорошо определенную модель предметной области. Каждая модель предметной области сопровождается набором открытых служб, реализующих архитектуру SOA, соответствующую целям бизнеса.

Передача репрезентативного состояния — REST

Автор: Стефан Тильков (Stefan Tilkov)

За последние несколько лет архитектура REST стала наиболее широко используемой и критикуемой. Как обычно, разные люди по-разному толкуют эту аббревиатуру. Для одних REST означает пересылку файлов XML по протоколу HTTP без использования архитектуры SOAP; другие считают REST эквивалентом использования протокола HTTP и формата JSON; третьи думают, что REST необходим для передачи методам аргументов в виде параметров URI-запросов. Все эти интерпретации являются ошибочными, но, к счастью, — в отличие от многих других концепций, таких как "компоненты" или "SOA" — существует авторитетный источник сведений об архитектуре REST: диссертация Роя Филдинга (Roy T. Fielding), который изобрел этот термин и дал его четкое определение.

REST как архитектурный стиль

Первое, что нужно понять, пытаясь освоить REST, — это концепция архитектурных стилей. Архитектурный стиль — это выбор проектного шаблона для конкретного проекта. Это абстракция аспектов, общих для нескольких конкретных реализаций, позволяющая обсуждать их относительные преимущества, не вникая в технические детали. Существует множество архитектурных стилей распределенных систем, включая клиент/сервер и распределенные объекты. Некоторые из них объясняются в первых главах диссертации Филдинга вместе с присущими им ограничениями. Концепция архитектурных стилей и ограничений, накладываемых на них, может показаться вам слишком теоретической, и вы будете правы. Эти стили образуют теоретическую основу для нового архитектурного стиля, предложенного Филдингом. Это стиль REST, характерный для веб-архитектуры.

Разумеется, сеть веб — со всеми наиболее важными стандартами URI, HTTP и HTML — предвосхитила диссертацию Филдинга. Но именно он был одним из основных разработчиков стандарта HTTP 1.1 и оказал огромное влияние на многие проектные решения, благодаря которым сеть веб стала такой, какой мы ее знаем.[5] Таким образом, REST — это теоретическая ретроспективная экстраполяция архитектуры сети веб.

Так почему же мы считаем стиль REST конкретным способом создания систем или, точнее говоря, создания веб-служб? Оказывается, как и любую другую

[5] Филдинг также является автором одной из первых широко используемых библиотек HTTP, одним из первых разработчиков HTTP-сервера Apache и основателем фонда Apache Software Foundation.

технологию, веб-протоколы можно использовать по-разному. Одни из этих способов соответствуют первоначальным намерениям разработчиков, а другие нет. Эта ситуация напоминает ситуацию, сложившуюся в сообществе разработчиков систем управления реляционными базами данных (СУРБД). Мы можем использовать СУРБД в соответствии с их архитектурными концепциями, т.е. определять таблицы со столбцами, отношения по внешнему ключу, представления, ограничения и т.д., или создать отдельную таблицу с двумя столбцами, один из которых назвать "ключ", а второй — "значение", и просто хранить объекты в столбце "значение" в виде последовательности битов. Разумеется, это можно назвать использованием СУРБД, но многие из преимуществ этой концепции в таком случае станут недоступными (осмысленные запросы, объединения, сортировка, группировка и т.д.).

Аналогично веб-протоколы можно использовать в соответствии с исходными идеями — архитектурным стилем REST, благодаря которому они стали тем, чем стали — или не следовать им. Как и в примере с СУРБД, можно проигнорировать базовый архитектурный стиль на свой страх и риск. В этом случае может оказаться приемлемой другая архитектура распределенных систем, в которой нет преимуществ использования протокола HTTP в стиле RESTful, как, например, хранилище NoSQL/ключ-значение — лучший выбор для хранения значений, ассоциированных с одним уникальным ключом.

Ключевые аспекты HTTP-сервера RESTful

Каковы ключевые аспекты распределенной архитектуры, использующей HTTP-сервер RESTful? Сначала рассмотрим серверную сторону. Отметим, что совершенно неважно, говорим ли мы о сервере, который используется человеком с помощью веб-браузера (веб-приложение), или каким-то другим агентом, например программой-клиентом, написанной на языке программирования (веб-служба).

Прежде всего, как следует из названия, ключевой концепцией являются ресурсы. Каким образом? В качестве разработчика системы вы решаете, какие осмысленные сущности следует сделать доступными для внешнего окружения, и присваиваете каждой из них уникальный идентификатор. Обычно каждый ресурс имеет один идентификатор URI и, что еще важнее, каждый идентификатор URI должен ссылаться на один ресурс — сущность, видимая извне, должна иметь индивидуальный адрес. Например, вы можете решить, что каждый клиент, каждый товар, каждый список товаров, каждый результат поиска и, возможно, каждое изменение в каталоге товаров должно быть полноценным ресурсом. Ресурсы имеют представления в одном или нескольких форматах. Именно с помощью представлений — документов в формате XML, JSON, HTML или простом бинарном формате — клиенты взаимодействуют с ресурсом.

Следующий ключевой аспект — идея коммуникации без сохранения состояния, использующей самодостаточные сообщения. Примером такой коммуникации является HTTP-запрос, содержащий всю информацию, которую должен обработать сервер. Конечно, сервер может (и обычно должен) использовать собственное сохраняющееся состояние, но важно то, что клиент и сервер не полагаются на отдельные запросы для настройки неявного контекста (сеанса). Это открывает доступ к каждому ресурсу независимо от других запросов, что обеспечивает огромные возможности масштабирования.

Если рассматривать ресурсы как объекты — а это вполне разумная точка зрения, — имеет смысл поинтересоваться, какой интерфейс они должны иметь. Ответ на этот вопрос — еще один важный аспект, отличающий стиль REST от всех других архитектурных стилей для распределенных систем. Набор методов, которые вы можете вызвать, фиксирован. Каждый объект поддерживает один и тот же интерфейс. На HTTP-сервере RESTful методы — это ключевые слова протокола HTTP (самые важные среди них — `GET`, `PUT`, `POST`, `DELETE`), которые можно применять к ресурсам.

Несмотря на первое впечатление, эти методы не переводятся в операции CRUD. Часто приходится создавать ресурсы, не представляющие никакой хранимой сущности, а вместо этого инкапсулирующие поведение, которое вызывается при использовании соответствующего слова. Каждый метод HTTP имеет очень четкое определение в спецификации протокола HTTP. Например, метод `GET` должен использоваться только для "безопасных" операций: 1) он может выполнять действия, отражающие эффекты, которые не запрашивал клиент; 2) он всегда читает данные; 3) он потенциально может кешироваться (если на это указывает сервер с помощью соответствующего заголовка ответа).

Ни кто иной, как сам Дон Бокс (Don Box), одна из главных фигур, стоящих за внедрением стиля SOAP в веб-службы, назвал метод `GET` протокола HTTP "наиболее оптимизированной частью распределенных систем в мире". Его слова подчеркивают производительность сети веб и ее масштабируемость, которую можно обеспечить в данном конкретном случае за счет оптимизации протокола HTTP.

Некоторые методы протокола HTTP являются *идемпотентными*, т.е. их можно безопасно вызывать, не опасаясь ошибок или неясных результатов. Это относится к методам `GET`, `PUT` и `DELETE`.

В заключение отметим, что сервер RESTful позволяет клиенту раскрыть с помощью гипермедиа путь возможных переходов из одного состояния в другое. В диссертации Филдинга этот принцип называется *"гипермедия как двигатель состояния приложения"* (Hypermedia as the Engine of Application State — HATEOAS). Проще говоря, индивидуальные ресурсы не зависят от самих себя. Они соединяются, связываются друг с другом. Это не удивительно. Помимо прочего, благодаря

этой особенности сеть веб получила свое название.[6] Для сервера это значит, что в свои ответы он будет встраивать связи, позволяя клиенту взаимодействовать с взаимосвязанными ресурсами.

Ключевые аспекты HTTP-клиента RESTful

HTTP-клиент RESTful переходит от одного ресурса к другому, следуя по ссылкам, содержащимся в представлениях ресурса, или перенаправляется к ресурсам в результате пересылки данных для обработки на сервер. Сервер и клиент совместно оказывают динамическое влияние на распределенное поведение клиента. Поскольку идентификатор URI содержит всю информацию, необходимую для определения адреса — включая имя хоста и порта, — клиент, придерживающийся принципа гипермедиа, может обращаться к произвольному ресурсу, относящемуся к другому приложению, другому хосту и даже другой компании.

В идеальной схеме REST клиент начинает работу с отдельного хорошо известного идентификатора URI, а затем подчиняется средствам управления гипермедиа. Именно эта модель используется браузером при обработке и отображении HTML-документов, включая ссылки и формы, предназначенные для пользователя. Затем браузер использует входную информацию, полученную от пользователя, для взаимодействия с многочисленными веб-приложениями, без априорного знания об их интерфейсе и реализациях.

К счастью, браузер не является самодостаточным агентом. Он требует от пользователя принятия реальных решений. Однако программный клиент может следовать тем же принципам, даже если часть логики была "зашита" в него. Он просто переходит по ссылкам и не предполагает, что под определенными URI расположены конкретные структуры и что ресурсы могут даже размещаться на одном сервере. Программный клиент будет использовать ту информацию, которую ему предоставил тот или иной ресурс.

Принципы REST и DDD

Иногда возникает соблазн непосредственно выразить модель предметной области через HTTP-сервер RESTful, но мы не рекомендуем так поступать. Этот подход часто приводит к созданию слишком уязвимых системных интерфейсов, на которых отражается любое изменение модели предметной области. Существуют два альтернативных подхода к сочетанию принципов DDD и HTTP-сервера RESTful.

[6] Web — паутина. — *Примеч. ред.*

Первый подход подразумевает создание отдельного ОГРАНИЧЕННОГО КОНТЕКСТА для уровня системного интерфейса и использование соответствующих стратегий для доступа к реальному СМЫСЛОВОМУ ЯДРУ из модели системного интерфейса. Этот подход считается классическим, поскольку в нем системный интерфейс рассматривается как связное целое, которое открывает доступ к себе с помощью абстракций ресурсов, а не служб или удаленных интерфейсов.

Рассмотрим конкретный пример этого подхода. Создадим систему, управляющую рабочей группой, а также ее задачами, календарными планами и встречами, подгруппами и всеми вспомогательными процессами. Нам потребуется чистая модель предметной области, незамутненная инфраструктурными деталями, которая включала бы в себя ЕДИНЫЙ ЯЗЫК и реализовала бы необходимую бизнес-логику. Для того чтобы опубликовать интерфейс для этой тщательно разработанной модели предметной области, мы обеспечим удаленный интерфейс в виде набора ресурсов RESTful. Эти ресурсы отражают пользовательские интерфейсы, необходимые для клиентов, которые, скорее всего, отличаются от чистой модели предметной области. При этом каждый ресурс создается, например, на основе одного или нескольких АГРЕГАТОВ, принадлежащих СМЫСЛОВОМУ ЯДРУ.

Разумеется, в рамках второго подхода мы могли бы просто использовать объекты предметной области как параметры методов ресурса JAX-RS. Скажем, конструкцию `/:user/:task` можно было бы отобразить в метод `getTask()`, возвращающий объект типа `Task`. Это кажется простым решением, но на самом деле создает одну большую проблему. Любое изменение структуры объекта `Task` немедленно скажется на удаленном интерфейсе, возможно, нарушив работу многих клиентов, даже если это изменение совсем не относилось к внешнему миру. Это нехорошо!

Итак, следует предпочесть первый подход, в котором СМЫСЛОВОЕ ЯДРО отделено от модели системного интерфейса. Это позволит нам вносить изменения в СМЫСЛОВОЕ ЯДРО, а затем в каждой отдельной ситуации решать, следует ли отображать это изменение в модели системного интерфейса, и, если да, делать это оптимальным образом. В этом подходе классы, разработанные для модели системного интерфейса, обычно являются производными от соответствующих классов СМЫСЛОВОГО ЯДРА, но, конечно, учитывают пользовательские сценарии. Примечание: даже в этом случае придется определить пользовательский тип среды.

Другой подход можно принять в ситуациях, когда основной акцент делается на стандартные типы среды. Если конкретные типы сред разрабатываются для поддержки не только отдельного системного интерфейса, но и целой категории аналогичных взаимодействий "клиент/сервер", то можно создать модель предметной области для представления каждого стандартного типа среды. Такую модель предметной среды можно повторно использовать для клиентов и серверов, даже если некоторые сторонники архитектурных стилей REST и SOA называют это

антишаблоном. Примечание: в терминах DDD этот шаблон можно назвать **ОБЩИМ ЯДРОМ (SHARED KERNEL) (3)** или **ОБЩЕДОСТУПНЫМ ЯЗЫКОМ (PUBLISHED LANGUAGE) (3)**.

Этот подход можно назвать комплексным (crosscutting). В упомянутых выше предметных областях рабочей группы и управления задачами существует много общих форматов. Рассмотрим в качестве примера формат *ical*. Это общий формат, который можно использовать в разных приложениях. В данном случае мы можем начать с выбора типа среды (ical), а затем создать модель предметной области для этого формата. Затем эту модель можно было бы использовать в любой системе, в которой необходим данный формат, — в нашем серверном приложении, например, и в любых других (например, в клиенте Android). Естественно, в рамках этого подхода сервер должен работать со многими типами сред, а один и тот же тип сред может использоваться на многих серверах.

Выбор одного из этих подходов в большой степени зависит от целей проектировщика системы с точки зрения повторного использования. Чем более специализированным является решение, тем более полезным оказывается первый подход. Чем более общим является решение, в пределе оно может даже быть официальным стандартом, тем больше смысла в использовании второго подхода, ориентированного на типы сред.

Почему REST?

Как показывает мой опыт, системы, разработанные в соответствии с принципами REST, как правило, обеспечивают слабую связанность. Обычно в существующие представления ресурсов очень легко добавлять новые ресурсы и ссылки на них. Кроме того, легко при необходимости добавлять новые форматы, обеспечивая более высокую надежность системных соединений. Системы, основанные на стиле REST, более понятные, поскольку их можно разделить на небольшие части — ресурсы, — каждая из которых представляет собой отдельную точку входа, допускающую тестирование, отладку и использование. Структура протокола HTTP и зрелость инструментов для таких функций, как перезапись URI и кеширование, делают архитектурный стиль RESTful HTTP отличным выбором, если нужна слабо связанная архитектура с большими возможностями масштабирования.

Разделение ответственности на команды и запросы, или Принцип CQRS

Иногда затруднительно запрашивать из ХРАНИЛИЩА все данные, которые необходимо предоставить пользователям. Это особенно характерно для ситуаций,

в которых проектирование взаимодействия пользователя и системы (user experience design) приводит к созданию представлений данных, включающих большое количество типов и экземпляров АГРЕГАТОВ. Чем более сложной является ваша предметная область, тем вероятнее такая ситуация.

Использование только ХРАНИЛИЩ для решения этой проблемы является совсем нежелательным. Мы могли бы потребовать от клиентов использовать множество ХРАНИЛИЩ для получения всех необходимых экземпляров АГРЕГАТОВ, а затем собирать необходимые компоненты в **ОБЪЕКТ ПЕРЕДАЧИ ДАННЫХ (DATA TRANSFER OBJECT)** (DTO) [Fowler, P of EAA]. Кроме того, мы могли бы разработать специальные инструменты для поиска информации в разных ХРАНИЛИЩАХ, чтобы собирать разбросанные данные с помощью одного запроса. Если эти решения кажутся неприемлемыми, возможно, целесообразно поступиться принципами проектирования взаимодействия пользователя и системы и жестко привязать представления данных к границам АГРЕГАТА в рамках модели. Большинство согласится, что в долгосрочной перспективе механистичный и спартанский пользовательский интерфейс к успеху не ведет.

Существует ли совершенно другой способ отображения данных предметной области в представления? Ответ заключается в архитектурном шаблоне, имеющем странное название **CQRS** [Dahan, CQRS; Nijof, CQRS]. Этот шаблон является результатом применения жесткого принципа проектирования объектов (или компонентов), который называется "разделением команд и запросов".

Этот принцип, сформулированный Бертраном Мейером (Bertrand Meyer), утверждает следующее.

> Каждый метод должен быть либо командой, выполняющей действие, либо запросом, возвращающим данные в вызывающий модуль, но не тем и другим одновременно. Иначе говоря, формулировка вопроса не должна изменять ответ. Точнее, методы должны возвращать значение, только если они не вносят изменений в переданные им по ссылкам объекты (включая текущий объект метода), а значит, не имеют побочных эффектов. [Wikipedia, CQS]

На уровне объектов это значит следующее.

1. Если метод модифицирует состояние объекта, то он является *командой* и не должен возвращать никаких значений. В языках Java и C# такой метод должен быть объявлен как `void`.
2. Если метод возвращает какое-то значение, то он является *запросом* и не должен ни прямо, ни косвенно модифицировать состояние объекта. В языках Java и C# такой метод должен быть объявлен вместе с типом значения, которое он возвращает.

Это очень ясный принцип, имеющий твердую теоретическую и практическую основу. Как и почему его следует применять в рамках подхода DDD?

Нарисуйте модель предметной области, например ту, которую мы рассматривали при описании **ОГРАНИЧЕННЫХ КОНТЕКСТОВ (2)**. Обычно на таком рисунке можно увидеть АГРЕГАТЫ, содержащие команды и запросы. Кроме того, на нем есть ХРАНИЛИЩА, имеющие множество методов поиска, выполняющих фильтрацию по определенным свойствам. Принцип CQRS запрещает такую "обычную ситуацию" и предлагает совершенно другой способ запроса просматриваемых данных.

Теперь представьте себе, что мы отделили все "чистые" запросы, которые есть в традиционных моделях, от всех "чистых" команд в той же модели. АГРЕГАТЫ не должны иметь запросов (get-методов) и содержать только команды. ХРАНИЛИЩА должны ограничиться только методом `add()` или `save()` (поддерживающими создание объектов и сохранение изменений) и только одним запросом, например `fromId()`. Этот единственный метод запроса получает уникальный идентификатор АГРЕГАТА и возвращает его. ХРАНИЛИЩЕ никак нельзя использовать для поиска АГРЕГАТА, например на основе фильтрации по дополнительным свойствам. Все эти компоненты, извлеченные из традиционной модели, называются *командной моделью*. Но нам все еще нужно представить данные пользователю. Для этого мы создадим вторую модель, настроенную на оптимизированные запросы. Она называется *моделью запросов*.

Не слишком ли это сложно?

Вам может показаться, что предложенный стиль требует большого объема работы и мы просто заменили одно множество проблем другим, написав много дополнительного кода.

Тем не менее не спешите отказываться от этого стиля. В некоторых ситуациях дополнительная сложность вполне оправданна. Напомним, что принцип CQRS предназначен для решения конкретной сложной проблемы представления данных, а не для того, чтобы дать вам возможность продемонстрировать свое мастерство.

Синонимы

Отметим, что некоторые области и компоненты шаблона CQRS могут быть известны под другими именами. Например, модель запросов известна также как модель чтения, а командная модель называется также моделью записи.

В результате традиционную модель предметной области можно разделить на две. Командная модель хранится в одном ХРАНИЛИЩЕ, а модель запросов — в другом. В итоге мы получим набор компонентов, представленных на рис. 4.6. Некоторые дополнительные детали в дальнейшем прояснят этот шаблон.

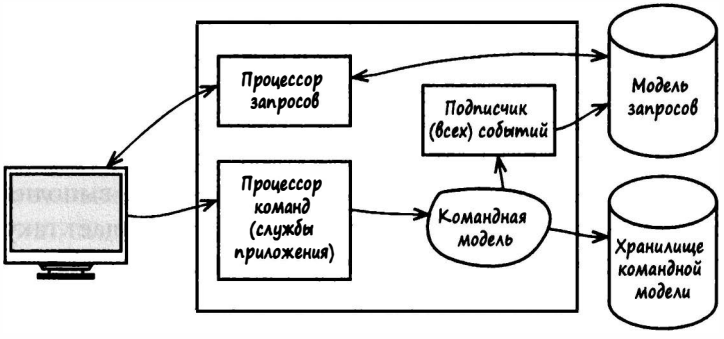


Рис. 4.6. В шаблоне CQRS команды, поступающие от клиента, приходят в командную модель. Запросы поступают в отдельный источник данных, оптимизированный для представления, и принимают форму пользовательского интерфейса или отчетов

Исследование областей шаблона CQRS

Рассмотрим основные области этого шаблона. Можно начать с поддержки клиента и запросов, а затем перейти к командной модели и посмотреть, как происходит модификация модели запросов.

Процессор клиентов и запросов

Клиент (крайний слева на рис. 4.6) может быть веб-браузером или пользовательским интерфейсом настольного приложения. Он использует процессоры запросов, работающие на сервере. На диаграмме нет архитектурно значимого разделения между уровнями на сервере или серверах. Независимо от того, существуют ли эти уровни, процессор запросов представляет собой отдельный компонент, который способен только выполнять типичные запросы к базе данных, например к хранилищу SQL.

На диаграмме нет сложных уровней. Самое большое — этот компонент выполняет запрос к базе данных и при необходимости может сериализовать результат запроса в определенном формате для пересылки (возможно, в виде DTO, а может быть, и нет). Если программа-клиент написана на языках Java или C#, она может обращаться к базе данных непосредственно. Однако для этого может потребоваться множество лицензий для клиентов базы данных, по одной на каждое соединение. Лучше всего использовать процессор запросов, использующий пул соединений.

Если клиент может получать результаты запросов от базы данных (например, по технологии JDBC), сериализация не обязательна, но может быть желательной. Существует две точки зрения на эту проблему. Согласно первой точке зрения для достижения простоты необходимо, чтобы клиент получал результат запроса или его простейшую сериализацию (в формате XML или JSON). Специалисты, придерживающиеся другой точки зрения, утверждают, что следует создавать и

передавать клиенту ОБЪЕКТЫ ПЕРЕДАЧИ ДАННЫХ. Хотя это дело вкуса, добавление ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ и **СБОРЩИКОВ ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ (DTO ASSEMBLERS)** [Fowler, P of EAA] создает *ненужную сложность* (accidental complexity). Команда должна сама выбрать наилучший вариант для своего проекта.

Модель запросов (или модель чтения)

Модель запросов — это денормализованная модель данных. Она не реализует поведение предметной области, а лишь отображает данные (и, возможно, создает отчеты). Если эта модель данных представляет собой базу данных SQL, то в каждой таблице должны храниться данные для отдельного вида клиентского представления (дисплея). В таблице может содержаться несколько столбцов и даже их супермножество, необходимое для любого представления пользовательского интерфейса. Табличные представления можно создавать из таблиц, каждое из которых используется как логическое подмножество целого.

> **Создавайте столько представлений, сколько требуется**
>
> Стоит отметить, что представления, основанные на шаблоне CQRS, могут быть дешевыми и одноразовыми (с точки зрения разработки и сопровождения), особенно если вы используете простую форму ПОРОЖДЕНИЯ СОБЫТИЙ (см. раздел "Порождение событий" в этой главе и приложение А), сохраняете все СОБЫТИЯ в постоянном хранилище и можете в любой момент опубликовать их снова для создания нового хранимого представления данных. В этом случае можно любое отдельное представление написать заново или переключить всю модель запросов на совершенно другую технологию постоянного хранения данных. Это упрощает создание и сопровождение представлений, постоянно учитывающих потребности пользовательского интерфейса. В результате можно достичь более интуитивного взаимодействия пользователя и системы, которое избегает табличной парадигмы, но обладает более богатыми возможностями.

Например, можно разработать таблицу с достаточным количеством данных для отображения в интерфейсах обычных пользователей, менеджеров и администраторов. Если для каждого из этих видов пользователей создать соответствующее табличное представление, то данные для каждой роли безопасности можно разделить соответствующим образом. Это обеспечивает безопасность просматриваемых данных для каждого типа пользователей. Компонент представления для обычного пользователя может выбирать все столбцы из табличного представления, предназначенного для обычного пользователя. Компонент представления для менеджера может выбирать все столбцы из табличного представления, предназначенного для менеджера. Таким образом обычные пользователи не смогут увидеть то, что предназначено для менеджеров.

Предпочтительно, чтобы инструкция выбора требовала только первичный ключ для используемого представления. В данном случае процессор запросов выбирает все столбцы из табличного представления товаров, предназначенного для обычного пользователя.

```
SELECT * FROM vw_usr_product WHERE id = ?
```

Заметим, что именование табличных представлений, принятое в этом примере, не носит обязательного характера. Эти имена просто показывают, как происходит выбор. Первичный ключ соответствует уникальному идентификатору определенного типа АГРЕГАТОВ или комбинированному множеству типов АГРЕГАТОВ, объединенных в одной таблице. В этом примере столбцу первичного ключа `id` в командной модели соответствует уникальный идентификатор `Product`. Проект модели данных должен по возможности содержать одну таблицу для каждого типа интерфейсного представления, а количество табличных представлений должно соответствовать количеству ролей безопасности в приложении. Однако следует быть практичными.

Будьте практичными

Если табло высокочастотного трейдинга используют 25 трейдеров и каждый из них торгует ценными бумагами, которые большинство других трейдеров не может видеть из-за правил, установленных комиссией SEC, то нужны ли вам 25 представлений? Более удобным было бы использовать фильтр трейдеров. В противном случае возникнет слишком много представлений, что с практической точки зрения нецелесообразно.

На практике этого бывает трудно достичь и в случае необходимости в запросах нужно объединять множество таблиц или табличных представлений. Объединение представлений или таблиц может оказаться необходимым или просто практичным способом фильтрации. В частности, это происходит, когда в предметной области существует много пользовательских ролей.

Не приводят ли табличные представления к появлению дополнительных расходов?

Классическое табличное представление в базе данных не связано с дополнительными расходами при модификации базовой таблицы. Представление просто соответствует запросу, и в этом случае даже не требуется объединения. Дополнительные расходы порождают только *материализованные представления*, потому что для того, чтобы сделать выбор, данные представления необходимо скопировать в одно место. Таблицы и представления следует разрабатывать так, чтобы модель запросов оптимально выполняла модификацию.

Обработка команд происходит под управлением клиента

Клиенты пользовательского интерфейса посылают команды на сервер (или косвенно запускают метод СЛУЖБЫ ПРИЛОЖЕНИЯ) в качестве средств реализации поведения АГРЕГАТОВ, содержащихся в командной модели. Отосланные команды содержат имя выполняемого метода и его параметры. Пакет команд представляет собой сериализованный вызов метода. Поскольку командная модель имеет тщательно разработанный контракт и поведение, соответствие между командами и контрактами представляет собой прямое отображение.

Для выполнения этого задания пользовательский интерфейс должен собрать данные, необходимые для правильной параметризации команды. Это значит, что взаимодействие пользователя и системы следует тщательно продумать. В результате пользователь должен достичь своей цели и послать явную команду. Лучше всего для этого подходит индуктивный, проблемно-ориентированный пользовательский интерфейс [Inductive UI]. Он фильтрует все неприемлемые опции, фокусируясь на точном выполнении команды. С другой стороны, можно разработать и дедуктивный пользовательский интерфейс, генерирующий явные команды.

Процессоры команд

Команда поступает на ОБРАБОТЧИК КОМАНД/процессор, который может иметь несколько разных стилей. Рассмотрим эти стили с их преимуществами и недостатками.

Мы можем использовать *классифицированный стиль* (categorized style) с несколькими ОБРАБОТЧИКАМИ КОМАНД в рамках одной СЛУЖБЫ ПРИЛОЖЕНИЯ. Этот стиль создает интерфейс СЛУЖБЫ ПРИЛОЖЕНИЯ и реализацию для категории команд. Каждая СЛУЖБА ПРИЛОЖЕНИЯ может иметь несколько методов, по одному для каждого типа команды с параметрами, соответствующими ее категории. Основным преимуществом этого подхода является его простота. Этот вид обработчика хорошо понятен, легко создается и легко сопровождается.

Мы можем создать обработчик *специализированного стиля* (dedicated style). Каждый из них должен представлять собой отдельный класс с одним методом. Контракт этого метода описывает конкретную команду с параметрами. Этот подход имеет явные преимущества: каждый обработчик/процессор имеет отдельную ответственность; каждый обработчик может быть заново введен в действие независимо от других; типы обработчиков можно масштабировать в соответствии с большими объемами некоторых команд.

Это приводит к ОБРАБОТЧИКУ КОМАНД в *стиле сообщений* (messaging style). Каждая команда отправляется как асинхронное сообщение и поступает на обработчик, разработанный в специализированном стиле. Это не только позволяет каждому компоненту обработчика получать сообщения специального типа, но и добавлять процессоры заданного типа для обработки команд. Этот подход не следует применять по умолчанию, поскольку у него более сложная структура.

Вместо этого следует начать с двух других стилей и создать синхронные командные процессоры. На асинхронные процессоры следует переходить, только если этого требуют проблемы масштабирования. С другой стороны, может показаться, что асинхронный подход, обеспечивающий временное распараллеливание, приводит к созданию более гибких систем. Эта точка зрения часто приводит к выбору стиля сообщений при разработке ОБРАБОТЧИКОВ КОМАНД.

Какой бы вид обработчиков вы ни выбрали, изолируйте их друг от друга. Не позволяйте одному обработчику зависеть от остальных. В этом случае вы сможете повторно использовать любой тип обработчика независимо от остальных.

ОБРАБОТЧИКИ КОМАНД обычно выполняют лишь несколько задач. Они создают новый экземпляр АГРЕГАТА и добавляют его в свое ХРАНИЛИЩЕ. Чаще всего они извлекают экземпляр АГРЕГАТА из своего хранилища и выполняют соответствующую команду.

```
@Transactional
public void commitBacklogItemToSprint(
    String aTenantId, String aBacklogItemId, String aSprintId) {
    TenantId tenantId = new TenantId(aTenantId);

    BacklogItem backlogItem =
       backlogItemRepository.backlogItemOfId(
           tenantId, new BacklogItemId(aBacklogItemId));

    Sprint sprint = sprintRepository.sprintOfId(
       tenantId, new SprintId(aSprintId));

    backlogItem.commitTo(sprint);
}
```

После завершения работы ОБРАБОТЧИКА КОМАНД происходит модификация отдельного экземпляра АГРЕГАТА, и модель команды публикует СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. Это гарантирует модификацию модели запроса. Обратите особое внимание на то, что, как было сказано при обсуждении **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)** и **АГРЕГАТОВ (10)**, опубликованное СОБЫТИЕ также может быть использовано для синхронизации других экземпляров АГРЕГАТА с помощью отдельной команды, но модификация дополнительных экземпляров АГРЕГАТА происходит в соответствии с командой, зафиксированной транзакцией.

Командная модель (или модель записи) реализует поведение

После выполнения каждой команды в командной модели командная модель публикует СОБЫТИЕ, как сказано в описании **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**. В нашем примере класс `BacklogItem` выполнил бы команду следующим образом.

```
public class BacklogItem extends ConcurrencySafeEntity {
    ...
    public void commitTo(Sprint aSprint) {



DomainEventPublisher
    .instance()
    .publish(new BacklogItemCommitted(
            this.tenant(),
            this.backlogItemId(),
            this.sprintId()));
     }
    ...
}
```

Что стоит за компонентом издателя

Конкретный класс DomainEventPublisher — это облегченный компонент, основанный на шаблоне **НАБЛЮДАТЕЛЬ (OBSERVER)** [Гамма и др.]. Подробности широкого распространения СОБЫТИЯ можно найти в описании **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**.

Это краеугольный камень для внесения самых свежих обновлений в командную модель. СОБЫТИЯ необходимы для хранения состояния модифицированного АГРЕГАТА (в нашем примере — BacklogItem), если используется шаблон ПОРОЖДЕНИЕ СОБЫТИЙ. Однако шаблон ПОРОЖДЕНИЕ СОБЫТИЙ не обязательно использовать совместно с шаблоном CQRS. Если регистрация СОБЫТИЯ не является требованием логики приложения, то команду можно сохранить с помощью механизма объектно-реляционного отображения (object-relational mapper — ORM) в реляционной базе данных или как-то иначе. В любом случае СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ должно быть опубликовано, чтобы гарантировать обновление модели запросов.

Когда команды не приводят к порождению событий

Существуют ситуации, в которых диспетчеризация команд не приводит к публикации СОБЫТИЙ. Например, если команда была отправлена с помощью системы сообщений, не гарантирующей ее доставку адресату, а само приложение гарантирует идемпотентность операций (т.е. повторное действие не изменяет состояние объекта), то повторное сообщение будет удалено без предупреждения.

Кроме того, рассмотрим ситуацию, в которой приложение проверяет корректность входящих команд. Все авторизованные клиенты знают о правилах проверки и всегда их выполняют. Однако неавторизованные клиенты, например, злоумышленники, посылающие некорректные команды, не пройдут проверку и могут быть удалены без предупреждения, не создав опасности для авторизованных пользователей.

Подписчик событий модифицирует модель запросов

Для получения всех СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ, публикуемых командной моделью, регистрируется специальный подписчик. Этот подписчик использует каждое СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ для обновления модели запросов, чтобы отобразить самые последние изменения командной модели. Отсюда следует, что каждое СОБЫТИЕ должно быть достаточно полным, чтобы можно было предоставлять все данные, необходимые для создания корректного состояния модели запросов.

Как следует выполнять эти обновления: синхронно или асинхронно? Это зависит от нормальной нагрузки на систему и, возможно, от того, где хранится база данных модели запросов. На решение влияют ограничения, накладываемые на согласованность данных, и требования, предъявляемые к производительности.

Для синхронного обновления модель запросов и командная модель должны были бы храниться в одной и то же базе данных (или иметь одну и ту же схему). Тогда мы обновляли бы обе модели в рамках одной транзакции. В этом случае обе модели остаются полностью согласованными одна с другой. Впрочем, для обновления нескольких таблиц потребовалось бы больше рабочего времени, что может противоречить соглашению о качестве предоставляемых услуг (service-level agreement — SLA). Если нагрузка на систему обычно высокая и обновление модели запросов происходит долго, следует использовать асинхронное обновление. Это может создать проблемы при обеспечении итоговой согласованности, потому что пользовательский интерфейс не будет немедленно отражать самые последние изменения в командной модели. Продолжительность задержки предсказать невозможно, но этот вопрос может быть предметом компромисса, связанного с выполнением других соглашений о качестве предоставляемых услуг.

Что происходит, если новое представление пользовательского интерфейса уже создано, но данные для него еще только должны быть созданы? Спроектируйте таблицу и табличные представления, как описано выше. Заполните новую таблицу текущим состоянием с помощью одной из технологий. Если командная модель сохраняется с помощью ПОРОЖДЕНИЯ СОБЫТИЙ или существует полное ХРАНИЛИЩЕ ИСТОРИЧЕСКИХ СОБЫТИЙ, замените исторические СОБЫТИЯ, чтобы внести изменения. Это возможно, только если в хранилище уже существуют события соответствующего вида. В противном случае таблицу следует заполнять по мере ввода в систему будущих команд. Впрочем, может существовать и другой вариант.

Если командная модель хранится сохраняется с помощью технологии ORM, используйте вспомогательное хранилище командной модели для заполнения новой таблицы модели запросов. Для этого можно использовать технологию генерации хранилища общих данных (или базы отчетов), например ETL (extract, transform, load — извлечь, преобразовать, загрузить). Т.е. извлекаем данные из хранилища

командной модели, преобразовываем ее с учетом потребностей пользовательского интерфейса и загружаем их в хранилище модели запросов.

Работа с моделью запросов с условием итоговой согласованности

Если проектируемая модель запросов должна иметь итоговую согласованность — т.е. обновление модели запросов осуществляется асинхронно вслед за выполнением записей в хранилище командной модели, — в пользовательском интерфейсе, работающем с ней, могут возникнуть проблемы. Например, будет ли полностью обновляться следующее представление пользовательского интерфейса и будет ли оно отражать согласованные данные из модели запросов после того, как пользователь введет команду? Это может зависеть от загрузки системы и других факторов. Однако лучше предполагать худшее и считать, что пользовательский интерфейс никогда не является согласованным.

Одно из решений — спроектировать пользовательский интерфейс для временной демонстрации данных, которые были успешно загружены на сервер как параметры только что выполненной команды. Это напоминает трюк, но он позволяет пользователю сразу увидеть то, что будет в конечном счете отражено в модели запроса. Возможно, это единственный способ гарантировать, что пользовательский интерфейс не будет отображать абсолютно устаревшие данные сразу после успешного выполнения команды.

Что делать, если для конкретного пользовательского интерфейса предложенный метод не подходит? А если даже и подходит, бывают моменты, когда какой-нибудь пользователь выполняет команду, а все другие пользователи, просматривающие связанные данные, видят абсолютно устаревшие данные. Как справиться с этой проблемой?

Метод, предложенный в работе [Dahan, CQRS], предусматривает, что в пользовательском интерфейсе всегда явно указываются дата и время получения данных от модели запроса, которую в настоящее время рассматривает пользователь. Для этого каждый отчет в модели запроса должен содержать дату и время последнего обновления. Это простое решение, которое обычно поддерживается триггером базы данных. Имея дату и время последнего обновления, пользовательский интерфейс может сообщать пользователю, насколько старыми являются данные. Если пользователь приходит к выводу, что данные слишком устарели, чтобы их использовать, он или она может в этот момент запросить более свежие данные. По общему признанию этот подход является эффективным, но некоторые критикуют его, считая уловкой или трюком. Эти противоположные точки зрения указывают на то, что перед реализацией того или иного решения необходимо проанализировать реакцию пользователей.

Все же возможно, что отсроченная синхронизация данных представления — вообще не критическая проблема. Ее можно решить другими средствами, такими как шаблон **КОМЕТА (COMET)** (или Ajax Push), или с помощью другой формы

скрытого обновления, такой как одна из вариаций шаблонов подписки на события **НАБЛЮДАТЕЛЬ (OBSERVER)** [Гамма и др.] и **РАСПРЕДЕЛЕННАЯ КЕШ-ПАМЯТЬ/GRID-СИСТЕМА (DISTRIBUTED CACHE/GRID)** (например, хранилища Coherence или Gemfire). Для устранения проблем с задержками вполне может быть достаточным только проинформировать пользователей о том, что их запрос был принят, а для получения результата потребуется некоторое время на обработку данных. Тщательно определите, создает ли проблемы возможная задержка из-за стремления обеспечить итоговую согласованность. Если это так, следует найти лучший способ решить эту проблему в заданном окружении.

Как и любой шаблон, CQRS порождает много конкурирующих факторов. Следует все тщательно продумать и сделать мудрый выбор. Конечно, если пользовательский интерфейс не слишком сложный или регулярно охватывает несколько разных АГРЕГАТОВ в рамках одного и того же представления, использование шаблона CQRS создало бы ненужную, а не необходимую сложность. Шаблон CQRS — правильный выбор, когда он устраняет риск, игнорирование которого с высокой вероятностью приводит к сбою.

Событийно-ориентированная архитектура

> Событийно-ориентированная архитектура (event-driven architecture — EDA) — это шаблон архитектуры программного обеспечения, позволяющий создавать, обнаруживать, потреблять события и реагировать на них [Wikipedia, EDA].

ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА, продемонстрированная на рис. 4.4, выражает концепцию системы, интегрированной в архитектуру EDA с помощью входящих и исходящих сообщений. Архитектура EDA не обязана использовать ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ, но с ее помощью можно довольно просто изложить эту концепцию. При разработке проекта "с нуля" целесообразно рассмотреть возможность использовать ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ в качестве основного стиля.

Анализ рис. 4.4 показывает, что треугольный клиент и соответствующий треугольный механизм вывода представляют собой механизм передачи сообщений, используемый ОГРАНИЧЕННЫМ КОНТЕКСТОМ. Входящие события поступают в ПОРТ, который отличается от ПОРТА, используемого другими тремя клиентами. Исходящие события аналогично идут через другой ПОРТ. Как предположено ранее, отдельные ПОРТЫ могли бы осуществлять передачу сообщений по протоколу AMQP, как в системе RabbitMQ, а не по более общему протоколу HTTP, который используют другие клиенты. Каким бы ни был реальный механизм передачи

сообщений, мы будем предполагать, что события входят и выходят из системы посредством символических треугольников.

Может существовать много различных видов событий, которые входят и выходят из шестиугольника. Мы интересуемся конкретно СОБЫТИЯМИ ПРЕДМЕТНОЙ ОБЛАСТИ. Приложение может также подписаться на события системы, предприятия или другие типы событий. Возможно, оно имеет дело с обеспечением работоспособности системы и мониторингом, регистрацией, динамическим обеспечением и т.п. Впрочем, именно СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ требуют особого внимания при моделировании.

Как показано на рис. 4.7, мы можем реплицировать систему в ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ столько раз, сколько потребуется, чтобы представить дополнение систем на предприятии, поддерживающем СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ. Еще раз подчеркнем: не каждая система основывается на ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ. На диаграмме просто показано, как можно реализовать СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ на основе нескольких систем, основанных на ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ. В других ситуациях следует свободно заменять шестиугольники УРОВНЯМИ или применять другой стиль.

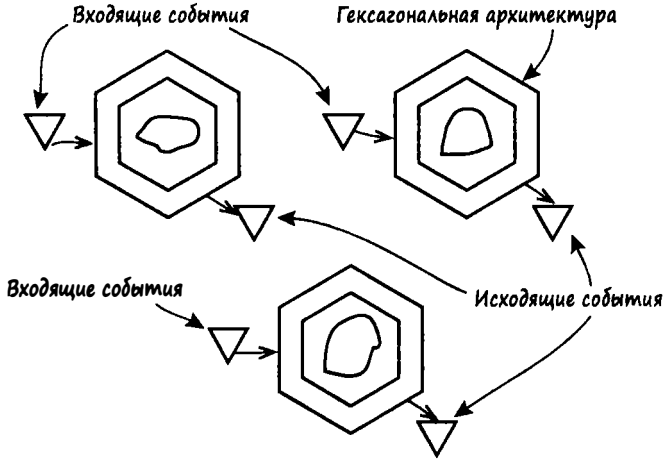


Рис. 4.7. Три системы, использующие СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ на основе ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ. Стиль EDA разрывает все, кроме системных, зависимости от механизма передачи сообщений и типа событий

СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, изданные одной такой системой через ПОРТ вывода, поставляются подписчикам, представленным в других системах через их ПОРТ ввода. В каждом ОГРАНИЧЕННОМ КОНТЕКСТЕ, получающем события, разные события предметной области имеют конкретный смысл или

вообще не имеют смысла.[7] Если тип СОБЫТИЯ представляет интерес в определенном КОНТЕКСТЕ, его свойства адаптируются к интерфейсу API приложения и используются, чтобы выполнить операцию. Командная операция, выполненная на интерфейсе API приложения, затем отражается в модель предметной области согласно ее протоколу.

Конкретное СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, полученное ОГРАНИЧЕННЫМ КОНТЕКСТОМ, может представлять собой только часть многозадачного процесса. Пока не поступят все предусмотренные СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, многозадачный процесс не считается законченным. Но как же начинается этот процесс? Как он распределяется по всему предприятию? И как завершается? Ответы на эти вопросы мы обсудим в разделе, посвященном длительным процессам. Но сначала необходимо заложить определенную теоретическую основу. Системы, основанные на обмене сообщениями, часто отражают стиль КАНАЛЫ И ФИЛЬТРЫ (PIPES AND FILTERS).

КАНАЛЫ И ФИЛЬТРЫ

В одной из своих простейших форм шаблон КАНАЛЫ И ФИЛЬТРЫ реализуется на основе командной строки системной оболочки или консоли.

```
$ cat phone_numbers.txt | grep 303 | wc -l
3
$
```

Выше приведена командная строка в системе Linux, которая позволяет выяснить, сколько контактов из штата Колорадо (телефонный код 303) хранится у менеджера в персональной адресной книге (файл `phone_numbers.txt`). По общему признанию, это не очень надежный способ реализации сценария использования, но он хорошо демонстрирует, как работают КАНАЛЫ И ФИЛЬТРЫ.

1. Утилита `cat` выводит содержимое файла `phone_numbers.txt` в *стандартный поток вывода*. Обычно этот поток соединен с консолью. Однако, если используется символ `|`, вывод направляется на вход следующей утилиты.
2. Затем утилита `grep` считывает свою входную информацию из стандартного потока ввода, который содержит результат работы утилиты `cat`. Аргумент утилиты `grep` предписывает ей найти строки, содержащие текст `303`. Затем каждая строка, которую она найдет, направляется в стандартный поток вывода. Как и в утилите `cat`, поток вывода утилиты `grep` направляется на вход следующей утилиты.

[7] Используя фильтры сообщений или ключи маршрутизатора, подписчики могут избежать получения СООБЩЕНИЙ, которые для них не имеют смысла.

3. В заключение утилита `wc` считывает свой стандартный поток ввода, который был заполнен содержимым стандартного потока вывода утилиты `grep`. Аргумент командной строки утилиты `wc` равен `-l`. Это значит, что утилита должна подсчитывать количество прочитанных строк. Утилита выводит результат, в данном случае равный 3, потому что утилита `grep` вывела три строки. Отметим, что теперь стандартный вывод отображается на консоли, потому что каналов к следующей команде больше нет.

Этот процесс можно также реализовать и с помощью командной строки системы Windows, но с меньшим количеством каналов.

```
C:\fancy_pim> type phone_numbers.txt | find /c "303"
3
```

Посмотрим, что происходит с каждой из этих утилит. Каждая утилита получает набор данных, обрабатывает его и выводит другой набор данных. Набор данных, предназначенный для вывода, отличается от введенного набора данных, потому что каждая утилита работает как ФИЛЬТР. В конце процесса фильтрации выходные данные совершенно отличаются от входных. На вход поступает текстовый файл, содержащий строки с информацией о номерах телефонов, а на выходе получаем цифру 3.

Как применить описанные выше принципы к СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ? Здесь мы можем обнаружить полезные совпадения. Последующее обсуждение основано на шаблоне **КАНАЛЫ И ФИЛЬТРЫ**, описанном в работе [Hohpe, Woolf]. Однако следует понимать, что подход КАНАЛЫ И ФИЛЬТРЫ, основанный на обмене сообщениями, не точно соответствует его версии, основанной на командной строке. Впрочем, эти версии вовсе не обязаны совпадать. Например, ФИЛЬТР в архитектуре EDA на самом деле не обязан ничего фильтровать. ФИЛЬТР в архитектуре EDA можно использовать для выполнения какой-нибудь обработки, оставляющей данные сообщения неизменными. И все же шаблон КАНАЛЫ И ФИЛЬТРЫ в архитектуре EDA достаточно похож на шаблон КАНАЛЫ И ФИЛЬТРЫ, основанный на командной строке, чтобы построить наше изложение на анализе предыдущего примера. Если вы опытный специалист, то сможете "профильтровать" следующий текст.

В табл. 4.2 приведены некоторые из основных характеристик процесса КАНАЛЫ И ФИЛЬТРЫ, основанного на обмене сообщениями.

Таблица 4.2. Основные характеристики процесса КАНАЛЫ И ФИЛЬТРЫ, основанного на обмене сообщениями

Характеристика	Описание
КАНАЛЫ передают сообщения	ФИЛЬТРЫ получают сообщения по входящему КАНАЛУ и посылают сообщения по исходящему КАНАЛУ. Таким образом, КАНАЛ представляет собой канал передачи сообщений
ПОРТЫ соединяют ФИЛЬТРЫ с КАНАЛАМИ	ФИЛЬТРЫ соединяются с входящими и исходящими КАНАЛАМИ с помощью ПОРТА. Благодаря ПОРТАМ **ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА (ПОРТЫ И АДАПТЕРЫ)** может стать базовой
ФИЛЬТРЫ — это процессоры	ФИЛЬТРЫ могут обрабатывать сообщения, на самом деле не фильтруя их
Отдельные процессоры	Процессор каждого ФИЛЬТРА представляет собой отдельный компонент. При правильном проектировании можно достичь требуемой степени детализации
Слабая связанность	Процессор каждого ФИЛЬТРА участвует в процессе независимо от всех других ФИЛЬТРОВ. Структуру процессора ФИЛЬТРА можно определить с помощью конфигурации
Взаимозаменяемость	Порядок, в котором процессоры получают сообщения, можно изменять в зависимости от требований сценария использования, конфигурируя их структуру
ФИЛЬТРЫ могут быть многоканальными	В то время как ФИЛЬТРЫ командной строки читают и записывают данные только в один КАНАЛ, ФИЛЬТРЫ, пересылающие сообщения, могут читать и/или записывать данные в несколько КАНАЛОВ, в результате чего возникает параллельная обработка
Параллельно используется один и тот же тип ФИЛЬТРОВ	Наиболее загруженные и, возможно, самые медленные ФИЛЬТРЫ можно развернуть параллельно, чтобы повысить производительность

Что если теперь мы будем считать каждую из утилит `cat`, `grep` и `wc` (или `type` и `find` в системе Windows) компонентами в СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ? Что если для аналогичной обработки телефонных номеров мы реализуем компоненты в виде отправителей и получателей сообщений, обрабатывающих телефонные номера? (Напоминаю, что я не пытаюсь иллюстрировать буквальную замену командной строки, а просто привожу простой пример передачи сообщений с теми же основными целями.)

Рассмотрим по этапам, как работает шаблон КАНАЛЫ И ФИЛЬТРЫ на основе передачи сообщений (рис. 4.8).

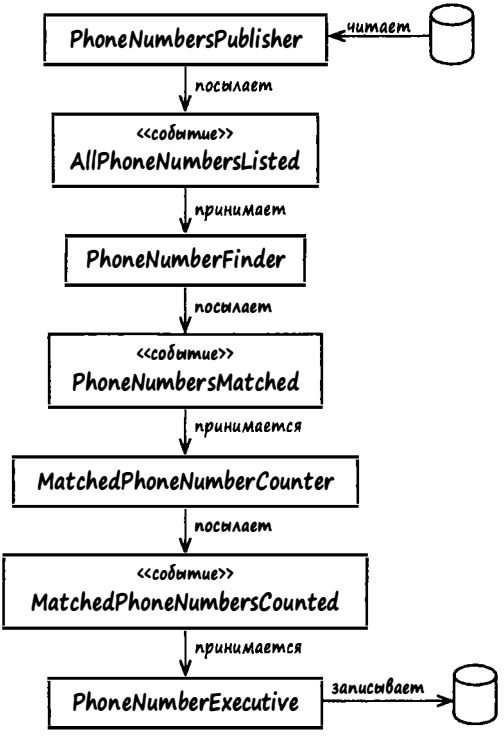


Рис. 4.8. Конвейер, образованный с помощью пересылки СОБЫТИЙ, которые обрабатываются ФИЛЬТРАМИ

1. Начнем с компонента `PhoneNumbersPublisher`, читающего все строки из файла `phone_numbers.txt`, а затем создающего и посылающего сообщение о СОБЫТИИ, содержащее все эти текстовые строки. СОБЫТИЕ называется `AllPhoneNumbersListed`. Конвейерная обработка начинается с его отсылки.
2. Компонент `PhoneNumberFinder`, обрабатывающий сообщение, является подписчиком события `AllPhoneNumbersListed` и получает его. Этот обработчик сообщений является первым ФИЛЬТРОМ в КОНВЕЙЕРЕ. Этот ФИЛЬТР настроен на поиск текста `303`. Этот компонент обрабатывает СОБЫТИЕ путем просмотра каждой строки в поисках текстовой последовательности `303`. Затем он создает новое СОБЫТИЕ с именем `PhoneNumbersMatched` и включает в него все строки, найденные в СОБЫТИИ. Новое сообщение о СОБЫТИИ пересылается дальше по конвейеру.

3. Компонент MatchedPhoneNumberCounter, обрабатывающий событие, является подписчиком события PhoneNumbersMatched и получает его. Этот обработчик событий является вторым ФИЛЬТРОМ в конвейере. Его единственной задачей является подсчет телефонных номеров в СОБЫТИИ и пересылка результатов в виде нового СОБЫТИЯ. В данном случае он насчитывает три искомые строки, содержащие телефонные номера. ФИЛЬТР завершает работу, создавая СОБЫТИЕ MatchedPhoneNumbersCounted и присваивая свойству count значение 3. Это СОБЫТИЕ пересылается дальше по конвейеру.
4. В заключение компонент PhoneNumberExecutive получает СОБЫТИЕ MatchedPhoneNumbersCounted, которое он должен обработать. Его единственной задачей является запись результата в файл, включая свойство СОБЫТИЯ count, а также дату и время его получения. В данном случае он заносит в файл запись

```
3 phone numbers matched on July 15, 2012 at 11:15 PM
```

На этом наша конвейерная обработка завершается.[8]

Этот вид конвейера довольно гибкий. Если мы хотим добавить в конвейер новые ФИЛЬТРЫ, достаточно создать новые СОБЫТИЯ, на которые подписан и которые публикует каждый из существующих ФИЛЬТРОВ. По существу, мы должны аккуратно изменить порядок конвейерной обработки с помощью изменения параметров конфигурации. Разумеется, изменить этот процесс не так легко, как в рамках подхода, основанного на использовании командной строки. Однако, как правило, конвейеры для обработки СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ изменяются редко. Несмотря на то что описанный процесс сам по себе не слишком интересен, он демонстрирует работу КАНАЛОВ И ФИЛЬТРОВ в СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ, основанной на обмене сообщениями.

Итак, следует ли ожидать, что с помощью шаблона КАНАЛЫ И ФИЛЬТРЫ мы сможем решить проблемы, аналогичные описанной? Ну, теоретически нет. (На самом деле, если вам этот пример уже надоел, то это, вероятно, объясняется тем, что вы уже знаете лучший вариант. Это прекрасно, но есть много людей, которым он может помочь.) Это всего лишь искусственный пример, демонстрирующий концепции. На реальном предприятии мы использовали бы этот шаблон, чтобы разбить большую проблему на подзадачи, которые легче понять и выполнить. Кроме

[8] Для простоты мы ничего не говорили о ПОРТАХ, АДАПТЕРАХ и прикладном интерфейсе приложения в ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ.

того, он показывает, как организовать несколько систем, чтобы они хорошо решали собственные задачи.

В реальном сценарии DDD СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ отражают имена, относящиеся к бизнесу. Этап 1 мог опубликовать СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, основанное на результате работы АГРЕГАТА в одном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Этапы 2–4 могли возникнуть в одном или нескольких ОГРАНИЧЕННЫХ КОНТЕКСТАХ, которые получают первичное СОБЫТИЕ, а затем публикуют одно из последующих СОБЫТИЙ. Эти три этапа могут создавать или изменять АГРЕГАТЫ в своих соответствующих КОНТЕКСТАХ. КОНТЕКСТ зависит от предметной области, но АГРЕГАТЫ являются общими результатами обработки СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ в АРХИТЕКТУРЕ КАНАЛОВ И ФИЛЬТРОВ.

Как указывается в описании **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**, они не являются просто техническими уведомлениями. Они явно моделируют действия, возникающие в ходе бизнес-процесса, которые являются полезными для знающих о них подписчиках в предметной области. При этом они содержат уникальный идентификатор и столько информативных свойств, сколько необходимо для ясного выражения модели. Хотя они являются синхронными, стиль пошаговой обработки можно обобщить и сделать параллельным.

ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ, или САГИ

Искусственный пример, посвященный КАНАЛАМ И ФИЛЬТРАМ, можно расширить для демонстрации другого событийно-ориентированного шаблона распределенной и параллельной обработки под названием **ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ (LONG-RUNNING PROCESSES)**. Длительные процессы иногда называют САГАМИ (SAGA), но это имя может означать другой, уже существовавший ранее шаблон, описанный в работе [Garcia-Molina & Salem]. Пытаясь избежать путаницы и неоднозначности, я выбрал имя ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ, а иногда для краткости буду писать просто ПРОЦЕССЫ.

Ковбойская логика

LB: Сериалы *Даллас* и *Династия*, вот что я называю *сагами*!

AJ: Все твои немецкие читатели видели *Династию* под названием *Клан из Денвера*.

Расширяя предыдущий пример, мы можем создать параллельный конвейер, добавив только один новый ФИЛЬТР под именем `TotalPhoneNumbersCounter` как дополнительный подписчик на СОБЫТИЕ `AllPhoneNumbersListed`.

Он получает СОБЫТИЕ AllPhoneNumbersListed параллельно с ФИЛЬТРОМ PhoneNumberFinder. Новый ФИЛЬТР имеет очень простое предназначение — подсчитать количество номеров телефонов в существующем файле контактов. Однако на этот раз ФИЛЬТР PhoneNumberExecutive начинает ДЛИТЕЛЬНЫЙ ПРОЦЕСС и следит за ним до самого конца. Исполнитель может использовать или не использовать ФИЛЬТР PhoneNumbersPublisher, важно лишь, что он имеет новое задание. Исполнитель, реализованный как СЛУЖБА ПРИЛОЖЕНИЯ или ОБРАБОТЧИК КОМАНД, отслеживает ход выполнения ДЛИТЕЛЬНОГО ПРОЦЕССА и знает, когда он завершается и что делать, когда это произойдет. Пример ДЛИТЕЛЬНОГО ПРОЦЕССА приведен на рис. 4.9.

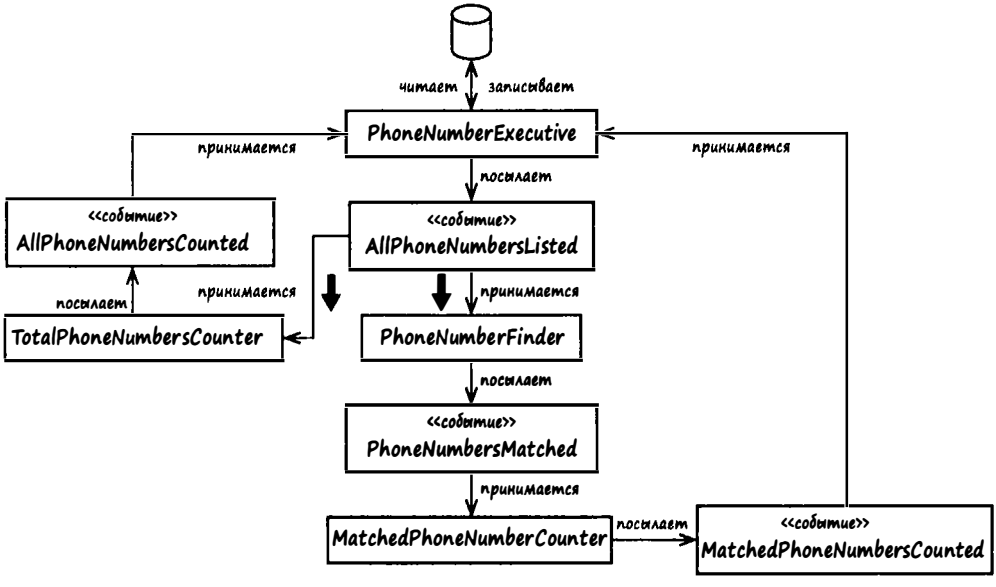


Рис. 4.9. Исполнитель отдельного ДЛИТЕЛЬНОГО ПРОЦЕССА инициирует параллельную обработку и следит, пока она не закончится. Широкие стрелки указывают, где начинается параллельная работа, когда два ФИЛЬТРА получают одно и то же СОБЫТИЕ

Другие способы проектирования длительных процессов

Ниже перечислены три подхода к разработке ДЛИТЕЛЬНОГО ПРОЦЕССА, хотя их может быть больше.

Проектирование процесса как составной задачи, за выполнением которой следит компонент-исполнитель, который регистрирует этапы и моменты завершения задачи с помощью объекта постоянного хранения. Этот подход подробно рассматривается в текущем разделе.

Проектирование процесса как набора партнерских АГРЕГАТОВ, взаимодействующих между собой. Один или несколько АГРЕГАТОВ действуют как исполнители и поддерживают все состояние процесса. Этот подход предложил Пат Хелланд (Pat Helland) из компании Amazon [Helland].

Проектирование процесса без фиксации состояния, в котором компонент-обработчик каждого сообщения, получающий сообщение о СОБЫТИИ, должен дополнить его информацией о ходе выполнения процесса и послать в виде следующего сообщения. Состояние всего процесса поддерживается только в теле каждого сообщения, посылаемого от одного компонента другому.

Поскольку теперь на первичное СОБЫТИЕ подписаны два компонента, оба ФИЛЬТРА получают одно и то же СОБЫТИЕ практически одновременно. Исходный ФИЛЬТР работает как обычно, выполняя сравнение с текстовым шаблоном 303. Новый ФИЛЬТР только подсчитывает все строки, а после завершения подсчета посылает СОБЫТИЕ `AllPhoneNumbersCounted`. СОБЫТИЕ содержит общее количество всех контактов. Если, например, в базе есть 15 телефонных номеров, то свойство СОБЫТИЯ `count` устанавливается равным 15.

Теперь компонент `PhoneNumberExecutive` должен подписаться на два СОБЫТИЯ: `MatchedPhoneNumbersCounted` и `AllPhoneNumbersCounted`. Параллельная обработка не считается завершенной, пока не будут получены оба эти СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ. Когда параллельная обработка завершается, ее результат объединяется в один результат. Исполнитель теперь регистрирует запись

```
3 of 15 phone numbers matched on July 15, 2012 at 11:27 PM
```

Как видите, в полученный результат кроме числа избранных контактов, даты и времени, было добавлено общее количество контактов. Несмотря на то, что задачи, позволяющие получить конечный объединенный результат очень просты, они выполняются параллельно. Если хотя бы некоторые из компонентов-подписчиков развернуты на разных вычислительных узлах, параллельная обработка также должна быть распределенной.

Однако с ДЛИТЕЛЬНЫМИ ПРОЦЕССАМИ связана одна проблема. Компонент `PhoneNumberExecutive` не знает, что он должен получить два СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, свидетельствующие о завершении соответствующих параллельных процессов. Если таких параллельных процессов будет много и СОБЫТИЯ, свидетельствующие о завершении процессов, будут поступать беспорядочно, то исполнитель не сможет определить, какой именно параллельный процесс был завершен. В нашем искусственном примере регистрация несоответствующих событий не приводит к серьезным последствиям. Однако в корпоративной среде неправильно согласованный ДЛИТЕЛЬНЫЙ ПРОЦЕСС может привести к катастрофе.

Первое, что нужно сделать, для того чтобы решить эту опасную проблему, — *присвоить* ПРОЦЕССУ *уникальный идентификатор*, который должен содержаться в каждом соответствующем СОБЫТИИ ПРЕДМЕТНОЙ ОБЛАСТИ. Им может быть идентификатор, который был присвоен исходному СОБЫТИЮ ПРЕДМЕТНОЙ ОБЛАСТИ, инициировавшему ДЛИТЕЛЬНЫЙ ПРОЦЕСС (например, `AllPhoneNumbersListed`). Кроме того, мы могли бы использовать универсальный уникальный идентификатор (UUID), приписанный конкретному ПРОЦЕССУ. Обсуждение вопроса об уникальном идентификаторе можно найти в описании **СУЩНОСТЕЙ (ENTITIES) (5)** и **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENTS) (8)**. Компонент `PhoneNumberExecutive` теперь мог бы делать запись в журнале регистрации при получении СОБЫТИЙ, свидетельствующих о завершении ПРОЦЕССОВ, имеющих одинаковые идентификаторы. Однако мы не можем рассчитывать на то, что исполнитель дождется получения всех СОБЫТИЙ, свидетельствующих о завершении ПРОЦЕССОВ. Он также является подписчиком входящих и исходящих СОБЫТИЙ, получая и обрабатывая их при каждом получении.

Исполнитель или регистратор?

Может показаться, что объединение концепций *исполнителя* и *регистратора* в одном объекте — АГРЕГАТЕ — самый простой подход. Реализация такого АГРЕГАТА в виде части модели предметной области, выполняющей регистрацию естественным образом как часть ПРОЦЕССА, может оказаться удобным методом. С одной стороны, нам не придется разрабатывать отдельный регистратор как конечный автомат в дополнение к АГРЕГАТАМ, которые должны существовать. Фактически большую часть ДЛИТЕЛЬНЫХ ПРОЦЕССОВ лучше всего реализовать именно так.

В ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ обработчик сообщения ПОРТ-АДАПТЕР мог бы просто направлять его СЛУЖБЕ ПРИЛОЖЕНИЯ (или ОБРАБОТЧИКУ КОМАНД), которая могла бы загрузить целевое ПРИЛОЖЕНИЕ и делегировать его соответствующему командному методу. Поскольку АГРЕГАТ, в свою очередь, мог бы инициировать СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, это СОБЫТИЕ можно было бы опубликовать, как признак того, что АГРЕГАТ выполнил свою роль в ПРОЦЕССЕ.

Описанный выше подход очень напоминает подход, предложенный Патом Хелландом, который назвал его *партнерскими действиями* (partner activities) [Helland]. Он является вторым подходом, описанным во врезке "Другие способы проектирования длительных процессов". Однако в идеале обсуждение отдельных исполнителей и регистраторов — более эффективный и интуитивный способ освоения всей технологии.

В реальной предметной области каждый экземпляр исполнителя ПРОЦЕССА создает новый объект состояния, похожий на АГРЕГАТ, для отслеживания его возможного завершения. Объект состояния создается в момент начала ПРОЦЕССА и содержит тот же уникальный идентификатор, который должно нести каждое связанное с ним СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. В него полезно включить также метку времени, когда ПРОЦЕСС начался (причины будут обсуждаться позже в этой главе). Объект регистратора состояния ПРОЦЕССА проиллюстрирован на рис. 4.10.

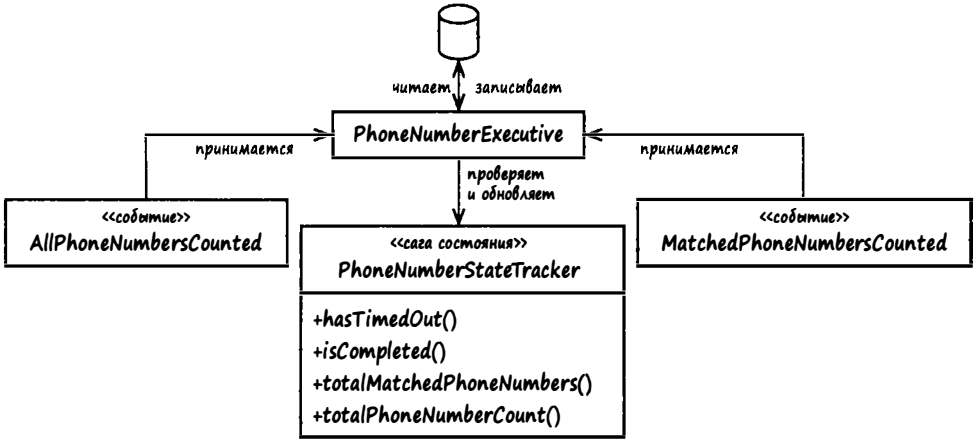


Рис. 4.10. Компонент `PhoneNumberStateTracker` играет роль объекта состояния ДЛИТЕЛЬНОГО ПРОЦЕССА, предназначенного для слежения за процессом; регистратор реализован как АГРЕГАТ

Как только каждый конвейер в параллельной обработке завершает работу, исполнитель получает соответствующее СОБЫТИЕ, свидетельствующее о завершении. ИСПОЛНИТЕЛЬ получает экземпляр объекта отслеживания состояния, проверяя уникальный идентификатор ПРОЦЕССА, который содержится в полученном СОБЫТИИ, и устанавливает свойство, которое представляет завершенный процесс.

У экземпляра состояния ПРОЦЕССА обычно есть метод, такой как `isCompleted()`. Как только все этапы завершены и зафиксированы регистратором состояний (state tracker), исполнитель выполняет метод `isCompleted()`. Этот метод проверяет, зарегистрировано ли завершение всех требуемых параллельных процессов. Если этот метод возвращает значение `true`, исполнитель может опубликовать финальное СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, если этого требует бизнес-логика. Это СОБЫТИЕ может потребоваться, например, если завершившийся ПРОЦЕСС представлял собой простое ответвление более крупного параллельного процесса.

Данный механизм передачи сообщений не может гарантировать *однократную доставку* каждого СОБЫТИЯ.[9] Если механизм передачи сообщений может доставлять СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ два или более раз, мы можем использовать объект состояния ПРОЦЕССА для дедупликации. Значит ли это, что механизм передачи сообщений должен иметь специальные функции для этого? Посмотрим, как это можно сделать без них.

Когда получены все СОБЫТИЯ, свидетельствующие о завершении процесса, *исполнитель ищет в объекте состояния существующую запись о завершении, соответствующую этому конкретному СОБЫТИЮ*. Если индикатор завершения уже установлен, СОБЫТИЕ считают копией и игнорируют.[10] Другая возможность заключается в том, чтобы *разработать идемпотентный объект события*. Таким образом, если исполнитель получил двойные сообщения, то объект состояния поглощает двойные записи. В то время как только второй вариант предусматривает разработку идемпотентного регистратора состояния, оба из этих подходов поддерживают идемпотентный обмен сообщениями. Дальнейшее обсуждение дедупликации изложено в описании **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**.

Некоторые механизмы отслеживания завершения ПРОЦЕССА могут быть чувствительными к задержкам. Мы можем обрабатывать простои ПРОЦЕССА пассивно или активно. Напомним, что механизм отслеживания состояния ПРОЦЕССА может содержать метку времени своего начала. Добавьте к этому общее допустимое постоянное время (параметр конфигурации), и руководитель сможет управлять чувствительными к задержкам ДЛИТЕЛЬНЫМИ ПРОЦЕССАМИ.

Пассивная проверка простоя выполняется каждый раз, когда исполнитель получает СОБЫТИЕ, свидетельствующее о завершении параллельной обработки. Исполнитель получает регистратор состояния и спрашивает его, произошла ли задержка. Этой цели может служить такой метод, как `hasTimed()`. Если пассивная проверка задержки показывает, что допустимый порог времени был превышен, регистратор состояния ПРОЦЕССА может быть отмечен как прекративший выполнение (abandoned). Также возможно опубликовать соответствующее СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, свидетельствующее об отказе. Обратите внимание на недостаток пассивной проверки задержки: он заключается в том, что ПРОЦЕСС может остаться активным надолго после превышения его временного порога, если по некоторым причинам исполнитель никогда не получит одно или более СОБЫТИЙ, свидетельствующих о завершении процесса. Такой подход может быть недопустим, если от успеха или неуспеха этого ПРОЦЕССА зависит более крупный параллельный процесс.

---

[9] Это вовсе не означает, что доставка сообщения не гарантируется, а означает лишь то, что не гарантируется доставка одного и только одного сообщения.

[10] Если механизм передачи сообщений получает подтверждение получения, то сообщение не может быть доставлено снова.

Активной проверкой простоев ПРОЦЕССА можно управлять, используя внешний таймер. Например, получить управляемый таймер в Java можно с помощью экземпляра JMX `TimerMBean`. Как только начинается ПРОЦЕСС, таймер устанавливается на максимальный порог простоя. Когда таймер срабатывает, слушатель получает доступ к регистратору состояния ПРОЦЕССА. Если ПРОЦЕСС не был завершен (этот факт всегда проверяется в случае срабатывания таймера, а также когда асинхронное СОБЫТИЕ завершает ПРОЦЕСС), он помечается как прекративший выполнение, и публикуется соответствующее СОБЫТИЕ, свидетельствующее об отказе. Если регистратор состояния процесса получает сигнал о завершении до срабатывания таймера, работу таймера можно прекратить. Недостатком активной проверки простоя является то, что для него требуется больше системных ресурсов, которые могут сильно нагружать среду с напряженным трафиком. Кроме того, состояние конкуренции между таймером и поступающим СОБЫТИЕМ, свидетельствующим о завершении, может ошибочно вызвать отказ.

ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ часто ассоциируются с распределенной параллельной обработкой, но не имеют никакого отношения к распределенным транзакциям. Они требуют учета итоговой согласованности. Необходимо направить все усилия на разработку ДЛИТЕЛЬНОГО ПРОЦЕССА, трезво рассчитывая на то, что, когда инфраструктура или сами задачи перестанут работать, хорошо продуманная процедура восстановления системы после ошибки сыграет свою роль. Каждая система, участвующая в единственном экземпляре ДЛИТЕЛЬНОГО ПРОЦЕССА, должна считаться несогласованной со всеми другими участниками, пока руководитель не получит уведомление о его полном завершении. Правда, некоторые ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ могут находиться в состоянии частичного завершения или даже простаивать в течение многих дней перед полным завершением. Но если ПРОЦЕСС близок к завершению и системы-участники остаются в несогласованных состояниях, может потребоваться компенсация. Если компенсация обязательна, по сложности она может превышать сложность разработки самой системы. Иногда в логике приложения допускаются отказы и предпринимаются попытки по восстановлению выполнения потока операций.

Команды SaaSOvation разворачивают СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ **в нескольких** ОГРАНИЧЕННЫХ КОНТЕКСТАХ, **а команда ProjectOvation будет использовать простейшую форму длительного процесса для управления созданием экземпляров класса** `Discussions`, **присвоенных экземплярам класса** `Product`. **Основным стилем обработки внешних сообщений и публикации** СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ **на предприятии является** ГЕКСАГОНАЛЬНАЯ АРХИТЕКТУРА.

Не следует забывать, что исполнитель ДЛИТЕЛЬНОГО ПРОЦЕССА может опубликовать один, два или больше СОБЫТИЙ, чтобы инициировать параллельную обработку. Кроме того, могут существовать не два, а три или больше подписчиков на любое инициирующее СОБЫТИЕ или СОБЫТИЯ. Иначе говоря, ДЛИТЕЛЬНЫЙ ПРОЦЕСС может состоять из многих отдельных действий бизнес-процесса, выполняющихся одновременно. Таким образом, наш искусственный пример ограничен в сложности только ради демонстрации фундаментальных понятий, связанных с ДЛИТЕЛЬНЫМ ПРОЦЕССОМ.

ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ часто полезны, когда интеграция с устаревшими системами приводит к продолжительным задержкам. Даже если задержка и использование устаревших систем — не главные проблемы, мы все еще можем извлечь выгоду из распределения и параллелизма с элегантностью, которая может привести к хорошо масштабируемым, высоконадежным бизнес-системам.

У некоторых механизмов передачи сообщений есть встроенная поддержка ДЛИТЕЛЬНЫХ ПРОЦЕССОВ, которые могут значительно ускорить их адаптацию. Один такой механизм описан в работе [NServiceBus], в которой они называются САГАМИ. Другая реализация САГ описана в работе [MassTransit].

ПОРОЖДЕНИЕ СОБЫТИЙ

Иногда компании заботятся об отслеживании изменений, происходящих с объектами в модели предметной области. Существуют разные уровни интереса к отслеживанию изменений и способы поддержки этих уровней. Обычно компании принимают решение следить только за тем, когда объект создан, а также когда и кем он был изменен в последний раз. Это относительно простой и прямой подход к отслеживанию изменений. Однако он не предоставляет информации об отдельных изменениях в модели.

Если компании требуется более высокий уровень слежения за изменениями, то ей понадобится больше метаданных. В этом случае компания начинает заботиться также об отдельных операциях, которые выполнялись в течение долгого времени. Возможно, она даже захочет понять, сколько времени заняло выполнение определенных операций. Эти желания приводят к необходимости вести системный журнал или журнал, содержащий показатели, подробно характеризующие выполнение сценария использования. Но у журнала есть свои ограничения. Он может содержать определенную информацию о том, что произошло в системе, возможно, даже допуская некоторую отладку. Но это не позволяет нам исследовать состояние отдельных объектов предметной области до и после изменений определенных видов. Что было бы, если бы мы могли извлечь больше из отслеживания изменений?

Как разработчики мы все имеем опыт подробного отслеживания изменений в том или ином виде. Наиболее распространенный пример — использование хранилища исходного кода, такого как CVS, Subversion, Git или Mercurial. У всех этих вариантов систем контроля версий исходного кода есть общая черта — все они знают, как отследить изменения, которые происходят с исходным файлом. Отслеживание изменений, обеспеченное этими инструментами, позволяет нам выполнить откат, просмотреть исходный код от его самой первой версии, а затем продолжить исследование версии за версией, вплоть до самой последней. Фиксируя все исходные файлы в системе контроля версий, можно отследить изменения целого жизненного цикла разработки.

Теперь, если мы подумаем о применении этого понятия к единственной СУЩНОСТИ, затем к АГРЕГАТУ, а затем к каждому АГРЕГАТУ в модели, то сможем понять важность объектов отслеживания изменений и значение, которое они могут иметь для наших систем. Имея это в виду, мы хотим разработать средство, позволяющее узнавать то, что произошло в модели и что привело к созданию конкретного АГРЕГАТА, а также что произошло с этим АГРЕГАТОМ с течением времени, операция за операцией. Учитывая историю произошедшего, мы могли бы даже поддерживать временные модели. Этот уровень отслеживания изменений лежит в основе шаблона под названием ПОРОЖДЕНИЕ СОБЫТИЙ (EVENT SOURCING).[11] Высокоуровневое представление этого шаблона показано на рис. 4.11.

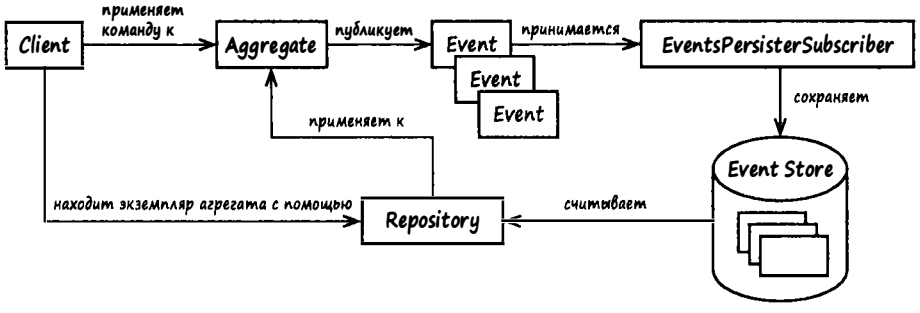


Рис. 4.11. Высокоуровневое представление ПОРОЖДЕНИЯ СОБЫТИЙ, в котором АГРЕГАТЫ публикуют СОБЫТИЯ, хранящиеся и использующиеся для слежения за изменениями состояния модели. ХРАНИЛИЩЕ считывает СОБЫТИЯ из ХРАНИЛИЩА СОБЫТИЙ и применяет их для воссоздания состояния АГРЕГАТА

Существуют разные определения ПОРОЖДЕНИЯ СОБЫТИЙ, поэтому требуется некоторое разъяснение. Мы обсуждаем способ использования, в котором каждая операционная команда, выполняемая с любым конкретным АГРЕГАТОМ в

[11] Для анализа ПОРОЖДЕНИЯ СОБЫТИЙ обычно необходимо понимать шаблон CQRS, который описан в предыдущем разделе.

модели предметной области, публикует по крайней мере одно СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, которое описывает результат ее выполнения. Каждое из событий сохраняется в **ХРАНИЛИЩЕ СОБЫТИЙ (EVENT STORE) (8)** в том порядке, в котором они произошли. Когда каждый АГРЕГАТ извлекается из его ХРАНИЛИЩА, воссоздается его экземпляр и СОБЫТИЯ воспроизводятся в порядке, в котором они ранее произошли.[12] Другими словами, сначала воспроизводится самое раннее СОБЫТИЕ, затем АГРЕГАТ применяет СОБЫТИЕ к самому себе, изменяя свое состояние. Затем второе по возрасту СОБЫТИЕ воспроизводится таким же образом. Это продолжается, пока не будут полностью воспроизведены и применены все СОБЫТИЯ, от самого старого до самого нового. В этот момент АГРЕГАТ существует в состоянии, которое он имел после окончания последнего выполнения некоторого поведения команды.

Движущаяся цель?

Определение шаблона ПОРОЖДЕНИЕ СОБЫТИЙ тщательно изучалось и уточнялось и во время написания книги еще не было сформулировано окончательно. Оно все еще требует уточнения, как и большинство передовых технологий. Здесь описана сущность шаблона с точки зрения шаблона DDD, которая, вероятно, в большой степени отражает общую тенденцию.

Не вызовет ли воспроизведение сотен, тысяч и даже миллионов СОБЫТИЙ серьезных задержек и перегрузки при обработке модели, в которой на протяжении долгого периода происходит изменение всех без исключения экземпляров АГРЕГАТОВ? По крайней мере, в некоторых моделях с напряженным трафиком это, определенно, может произойти.

Для того чтобы избежать этого, можно применить оптимизацию, использующую *снимки* (snapshots) состояний АГРЕГАТА. Для этого создается специальный процесс, который в фоновом режиме создает снимки состояний АГРЕГАТА для заданных точек истории ХРАНИЛИЩА СОБЫТИЙ. При этом АГРЕГАТ загружается в память, используя все СОБЫТИЯ, предшествовавшие текущему моменту времени. Затем состояние АГРЕГАТА сериализуется, и сериализированный образ снимка сохраняется в ХРАНИЛИЩЕ СОБЫТИЙ. Начиная с этого момента сначала создается экземпляр АГРЕГАТА с использованием самого свежего снимка, а затем воспроизводятся все новые СОБЫТИЯ, произошедшие после создания снимка АГРЕГАТА, как было описано выше.

Снимки не создаются случайным образом. Более того, они могут быть созданы в точках, где произошло предопределенное количество новых СОБЫТИЙ. Этот порог можно установить на основе эвристических данных или других наблюдений.

[12] Состояние АГРЕГАТА — это объединение предыдущих СОБЫТИЙ, в которое они включаются в том порядке, в котором происходили.

Например, мы могли бы обнаружить, что выборка АГРЕГАТА выполняется оптимально при наличии не более 50 или 100 СОБЫТИЙ между снимками.

Шаблон ПОРОЖДЕНИЕ СОБЫТИЙ в большой степени имеет технический характер. Мы можем создавать модели предметной области, которые публикуют СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, без необходимости поддерживать шаблон ПОРОЖДЕНИЕ СОБЫТИЙ. Как механизм постоянного хранения шаблон ПОРОЖДЕНИЕ СОБЫТИЙ является альтернативой использованию инструмента ORM. Поскольку СОБЫТИЯ часто сохраняются в ХРАНИЛИЩЕ СОБЫТИЙ в двоичном виде, они не могут (оптимально) использоваться для запросов. Фактически ХРАНИЛИЩА, разработанные для модели ПОРОЖДЕНИЯ СОБЫТИЙ, требуют только одной операции `get/find`, и этому методу передается в качестве параметра только уникальный идентификатор АГРЕГАТА. Более того, АГРЕГАТЫ проекта не имеют никаких методов запроса (`get`-методов). В результате возникает необходимость в другом способе запроса, который обычно приводит к использованию шаблона CQRS (рассмотренного ранее) в сочетании с шаблоном ПОРОЖДЕНИЕ СОБЫТИЙ.[13]

Поскольку ПОРОЖДЕНИЕ СОБЫТИЙ ведет нас по иному пути размышлений о способе, которым разработаны модели предметной области, мы должны обосновать свое решение. В основном историю СОБЫТИЙ можно использовать для устранения ошибок в системе. У отладки с использованием явной истории всего, что когда-либо происходило с моделью, есть большое преимущество. ПОРОЖДЕНИЕ СОБЫТИЙ может привести к высокопроизводительным моделям предметной области, достигая чрезвычайно большого количества транзакций в секунду. Добавление отдельной таблицы в базу данных, например, выполняется чрезвычайно быстро. Более того, это позволяет масштабировать модель запроса CQRS, потому что обновления источника данных выполняются в фоновом режиме после того, как в ХРАНИЛИЩЕ СОБЫТИЙ записываются новые СОБЫТИЯ. Это еще больше способствует репликации модели запроса на большее количество экземпляров источника данных для поддержки растущего числа клиентов.

Однако технические преимущества не означают преимущества для бизнеса. Рассмотрим деловые преимущества использования ПОРОЖДЕНИЯ СОБЫТИЙ, возникающие благодаря техническое реализации.

- "Латание" ХРАНИЛИЩА СОБЫТИЙ новыми или модифицированными СОБЫТИЯМИ, которые решают проблемы. Оно может влиять на бизнес, но если это легально в данной ситуации, "заплатка" может защитить систему от серьезных проблем, которые могли бы произойти из-за ошибок в модели. Так как у заплаток есть встроенный регистрационный журнал, их

[13] Несмотря на то что шаблон CQRS можно использовать без шаблона ПОРОЖДЕНИЕ СОБЫТИЙ, шаблон ПОРОЖДЕНИЕ СОБЫТИЙ без шаблона CQRS обычно не используется.

использование может смягчить любые юридические последствия, делая их явными и прослеживаемыми.

- Помимо латания, можно отменять и повторять изменения в модели, воспроизводя разнообразные наборы СОБЫТИЙ. Это может привести к техническим и деловым последствиям. Впрочем, эта возможность существует не всегда.
- Имея точную историю всего, что произошло в модели предметной области, бизнес может рассматривать вопросы "что если". Иначе говоря, воспроизводя сохраненные СОБЫТИЯ на множестве АГРЕГАТОВ, имеющих экспериментальные модификации, бизнес может получить точные ответы на гипотетические вопросы. Бизнес извлек бы выгоду, если бы можно было моделировать концептуальные сценарии, используя реальные исторические данные? Очень вероятно, что да. Это альтернативный способ бизнес-анализа.

Может ли бизнес извлечь выгоду из технических и нетехнических улучшений?

В приложении А приведено много деталей реализации АГРЕГАТОВ с ПОРОЖДЕНИЕМ СОБЫТИЙ и обсуждение его проекции на шаблон CQRS. Более подробную информацию можно найти в работах [Dahan, CQRS] и [Nijof, CQRS].

ФАБРИКА ДАННЫХ и РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ

Автор: Вес Уильямс (Wes Williams)

Поскольку по мере расширения баз пользователей и требований, относящихся к "большим данным", системы программного обеспечения становятся все более сложными, традиционные решения для баз данных могут стать узкими местами с точки зрения производительности. У организаций, сталкивающихся с реалиями информационных систем колоссального размера, нет альтернативы, кроме поиска решений вычислительных проблем. Шаблон ФАБРИКА ДАННЫХ (DATA FABRIC) — также иногда называемый шаблоном РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ (GRID COMPUTING)[14] — обеспечивает производительность и возможности гибкого масштабирования, которых требуют такие бизнес-ситуации.

[14] Это не значит, что ФАБРИКИ и РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ — идентичные концепции, но с точки зрения архитектуры эти названия часто означают одно и то же. Маркетинг также часто ограничивает эти концепции одним и тем же смыслом. В любом случае в этом разделе термин *фабрика данных* означает более широкий выбор возможностей, чем РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ.

Ковбойская логика

AJ: Хочешь информацию в обмен на выпивку?

LB: Извини. Я принимаю только кеш.[15]

Хорошая новость о ФАБРИКАХ ДАННЫХ заключается в том, что они поддерживают модели предметной области естественным способом, почти устраняя любое рассогласование. Фактически их распределенные кеши легко обеспечивают постоянное хранение объектов предметной области в целом и, в частности, действуют как ХРАНИЛИЩА АГРЕГАТОВ.[16] Проще говоря, АГРЕГАТ, хранящийся в фабричном кеше, основанном на отображении[17], является значением в паре "ключ–значение". Ключ формируется из глобального уникального идентификатора АГРЕГАТА, а состояние АГРЕГАТА сериализуется к некоторому бинарному или текстовому представлению, которое служит значением.

```
String key = product.productId().id();

byte[] value = Serializer.serialize(product);

// регион (GemFire) или кеш (Coherence)
region.put(key, value);
```

Таким образом, положительным результатом использования ФАБРИКИ ДАННЫХ с функциями, хорошо согласованными с техническими аспектами модели предметной области, является возможность сокращения циклов разработки.[18]

Примеры, приведенные в этом разделе, демонстрируют, как ФАБРИКА ДАННЫХ может разместить модель предметной области в кеше и добавлять системные функциональные возможности в распределенном масштабе. При этом мы исследуем способы поддержки образца архитектуры CQRS и СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЫ, использующей ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ.

---

[15] Игра слов: *cash* — наличные деньги и *cash* — буфер с быстрым доступом. — *Примеч. пер.*

[16] Мартин Фаулер (Martin Fowler) предложил термин "хранилище агрегатов" совсем недавно, хотя эта концепция уже существовала некоторое время.

[17] В системе GemFire она называется регионом, а в системе Coherence — кешем. Для согласованности я использую термин "кеш".

[18] Некоторые хранилища NoSQL функционируют как естественные "ХРАНИЛИЩА АГРЕГАТОВ", упрощая технические аспекты реализации принципов DDD.

Репликация данных

Размышляя о кеше данных в оперативной памяти, мы можем сразу рассмотреть реальную возможность потери всего или части состояния нашей системы, если кеш перестанет работать. Это реальная проблема, но совсем не сложная, если в ФАБРИКУ встроена избыточность.

Рассмотрим кеш-память, обеспеченную ФАБРИКОЙ, использующей стратегию "один кеш на АГРЕГАТ". В этом случае ХРАНИЛИЩЕ АГРЕГАТОВ данного типа поддерживается специальным кешем. Кеш, поддерживающий только один узел, был бы довольно уязвимым для отказов в единственной точке. Однако ФАБРИКА, обеспечивающая кеши, распределенные по многим узлам с возможностью репликации, была бы довольно надежной. Уровень избыточности можно выбрать, оценив вероятность отказа большого количества узлов, которые могут перестать работать в любой момент времени. Чем больше узлов включено, тем меньше эта вероятность. Можно также обменять избыточность на производительность, поскольку на производительность может влиять количество репликаций узла, требуемых для полной фиксации АГРЕГАТА.

Рассмотрим пример того, как может работать избыточность кеша (или региона, в зависимости от конкретной ФАБРИКИ). Один узел действует как *основной* кеш/регион, а все остальные *вторичны*. Если основное хранилище перестало работать, происходит переключение на резервные мощности, и одно из вторичных хранилищ становится новым основным хранилищем. Когда прежнее основное хранилище восстанавливается, все данные, хранившиеся в новом основном хранилище, реплицируются на восстановленный узел, и он становится вторичным хранилищем.

Дополнительное преимущество переключения узлов состоит в том, что они обеспечивают гарантируемую доставку событий, опубликованных ФАБРИКОЙ. Таким образом, обновления АГРЕГАТОВ и любые опубликованные СОБЫТИЯ ФАБРИКИ никогда не теряются. Очевидно, избыточность кеша и репликация — существенные функциональные возможности для хранения критически важных для бизнеса объектов модели предметной области.

Событийно-ориентированные фабрики и события предметной области

Основная функция ФАБРИКИ — поддержка событийно-ориентированного стиля с гарантируемой доставкой сообщений. У большинства ФАБРИК есть встроенная обработка событий технического характера, т.е. автоматического уведомления о событиях, которые сообщают о состоянии кеша и записей. Их не следует путать с СОБЫТИЯМИ ПРЕДМЕТНОЙ ОБЛАСТИ. Например, событие на уровне кеша сообщает о таких ситуациях, как повторная инициализация кеша, и событие на уровне записи сообщает о таких случаях, как создание и обновление записи.

Однако для ФАБРИКИ, поддерживающей открытую архитектуру, должен быть способ поддерживать публикацию СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ непосредственно АГРЕГАТАМИ. СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ может выделить подкласс конкретного каркасного события, такой как `EntryEvent` (например, GemFire), но это лишь малая цена за мощь, которую они обеспечивают.

Как реально использовать СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ в ФАБРИКЕ? Как указывалось при обсуждении **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**, **АГРЕГАТЫ** могли бы использовать простой компонент `DomainEventPublisher`. В кеше ФАБРИКИ этот издатель может просто поместить опубликованные СОБЫТИЯ в определенный кеш/регион. Кешируемые СОБЫТИЯ были бы тогда доставлены подписчикам (слушателям) синхронно или асинхронно. Для того чтобы не тратить впустую драгоценную память в этом специальном кеше/регионе СОБЫТИЙ, после того как каждое СОБЫТИЕ подтверждается всеми подписчиками, его запись удаляется из отображения. Разумеется, каждое СОБЫТИЕ подтверждается только полностью, как только оно опубликовано одним или более подписчиками на очередь или шину сообщений и/или использовано для обновления модели запроса CQRS.

Поскольку подписчики СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ могут использовать СОБЫТИЯ для синхронизации с другими АГРЕГАТАМИ, такой архитектурный стиль обеспечивает итоговую согласованность.

Непрерывные запросы

Некоторые ФАБРИКИ поддерживают особую разновидность уведомлений — НЕПРЕРЫВНЫЙ ЗАПРОС (CONTINUOUS QUERY). Он позволяет клиентам регистрировать запрос к ФАБРИКЕ, которая гарантирует, что клиент получит уведомление об изменениях в кеше, которые удовлетворяют запросу. НЕПРЕРЫВНЫЙ ЗАПРОС можно использовать, например, в компонентах пользовательского интерфейса, позволяя прослушивать изменения, которые могут повлиять на текущее представление.

Вы видите, что получается? Если ФАБРИКА поддерживает модель запроса, то шаблон CQRS точно соответствует свойствам НЕПРЕРЫВНОГО ЗАПРОСА. Вместо того чтобы заставлять представление следить за обновлениями табличного представления, можно получать уведомления в соответствии с НЕПРЕРЫВНЫМИ ЗАПРОСАМИ, обеспечивая своевременное изменение представлений. Рассмотрим пример, в котором клиент регистрируется на получение событий НЕПРЕРЫВНОГО ЗАПРОСА в системе Gemfire.

```
CqAttributesFactory factory = new CqAttributesFactory();
CqListener listener = new BacklogItemWatchListener();

factory.addCqListener(listener);
```

```
String continuousQueryName = "BacklogItemWatcher";

String query = "select * from /queryModelBacklogItem qmbli "
 + "where qmbli.status = 'Committed'";

CqQuery backlogItemWatcher = queryService.newCq(
 continuousQueryName, query, factory.create());
```

Если при модификации АГРЕГАТА выполняются критерии запроса, то ФАБРИКА ДАННЫХ теперь будет поставлять модель запроса CQRS объекту обратного вызова клиента, предоставленному классом `CqListener`, добавляя, модифицируя или удаляя метаданные.

Распределенная обработка

Мощь ФАБРИКИ ДАННЫХ заключается в распределении обработки через реплицированные кеши и возвращении агрегированных результатов клиенту. Это позволяет ФАБРИКЕ выполнять событийно-ориентированную распределенную параллельную обработку, возможно, используя ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ.

Для того чтобы проиллюстрировать эту функцию, мы должны будем упомянуть некоторые конкретные подходы в системах Gemfire и Coherence. Исполнителя ПРОЦЕССА можно реализовать как ФУНКЦИЮ в системе Gemfire или ПРОЦЕССОР ЗАПИСИ в системе Coherence. Они оба могут быть обработчиками **КОМАНДЫ (COMMAND)** [Гамма и др.], которые функционируют параллельно через распределенный реплицированный кеш. (Вместо этого можно рассматривать эту концепцию как СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ, но ее предназначение может не зависеть от предметной области). Для сохранения согласованности будем называть эту функциональную возможность ФУНКЦИЕЙ. Эта ФУНКЦИЯ может дополнительно получать фильтр, чтобы ограничивать свое выполнение соответствующими экземплярами АГРЕГАТА.

Рассмотрим пример ФУНКЦИИ, реализующей ДЛИТЕЛЬНЫЙ ПРОЦЕСС под названием "Подсчет телефонных номеров". Этот ПРОЦЕСС будет выполняться параллельно несколькими реплицированными кешами с помощью ФУНКЦИИ в системе Gemfire.

```
public class PhoneNumberCountSaga extends FunctionAdapter {
    @Override
    public void execute(FunctionContext context) {
        Cache cache = CacheFactory.getAnyInstance();
        QueryService queryService = cache.getQueryService();

        String phoneNumberFilterQuery = (String) context.getArguments();
        ...
        // Псевдокод
```

```
        // - Выполняет ФУНКЦИЮ, чтобы получить MatchedPhoneNumbersCounted.
        // - Посылает ответ сборщику, вызывая метод
        //   aggregator.sendResult(MatchedPhoneNumbersCounted).
        // - Выполняет ФУНКЦИЮ, чтобы получить AllPhoneNumbersCounted.
        // - Посылает ответ сборщику, вызывая метод
        //   aggregator.sendResult(AllPhoneNumbersCounted).
        // - Сборщик автоматически накапливает ответы,
        //   полученные от каждого вызова распределенных ФУНКЦИЙ,
        //   и возвращает клиенту один накопленный ответ.
    }
}
```

Ниже приведен образец кода, параллельно выполняющего ДЛИТЕЛЬНЫЙ ПРОЦЕСС в реплицированном распределенном кеше.

```
PhoneNumberCountProcess phoneNumberCountProcess =
        new PhoneNumberCountProcess();

String phoneNumberFilterQuery =
        "select phoneNumber from /phoneNumberRegion pnr "
        + "where pnr.areaCode = '303'";
```

```
Execution execution =
        FunctionService.onRegion(phoneNumberRegion)
                .withFilter(0)
                .withArgs(phoneNumberFilterQuery)
                .withCollector(new PhoneNumberCountResultCollector());

PhoneNumberCountResultCollector resultCollector =
                 execution.execute(phoneNumberCountProcess);

List allPhoneNumberCountResults = (List) resultsCollector.getResult();
```

Конечно, ПРОЦЕСС мог быть намного более сложным или намного более простым, чем этот. Он также демонстрирует, что ПРОЦЕСС не обязательно является СОБЫТИЙНО-ОРИЕНТИРОВАННЫМ ПОНЯТИЕМ, но может работать с другими параллельными подходами распределенной обработки. Полное обсуждение распределенной и параллельной обработки с помощью ФАБРИК можно найти в документе [GemFire Functions].

Резюме

Мы рассмотрели несколько архитектурных стилей и шаблонов, которые можно использовать в рамках подхода DDD. Это далеко не исчерпывающий список, поскольку в этой области существует слишком много возможностей, что еще раз подчеркивает универсальность принципов DDD. Например, мы не рассмотрели, как подход DDD сочетается с концепцией "отображение–свертка". Это будет темой дальнейших обсуждений.

- Мы обсудили традиционную МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ и показали, как ее можно улучшить с помощью ПРИНЦИПА ИНВЕРСИИ ЗАВИСИМОСТЕЙ.
- Вы осознали потенциал, вероятно, неподвластной времени ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ, лежащей в основе архитектуры многих приложений.
- Мы показали, как следует использовать принципы DDD в среде SOA с помощью архитектуры REST и как использовать ФАБРИКУ ДАННЫХ или РАСПРЕДЕЛЕННЫЙ КЕШ ДЛЯ КОЛЛЕКТИВНОЙ РАБОТЫ.
- Мы привели обзор шаблона CQRS и показали, как он может упростить некоторые аспекты приложений.
- Мы рассмотрели разные аспекты событийно-ориентированной архитектуры, включая КАНАЛЫ И ФИЛЬТРЫ, ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ, и даже вскользь упомянули ПОРОЖДЕНИЕ СОБЫТИЙ.

Теперь мы переходим к ряду глав о тактическом моделировании с помощью принципов DDD. Эти главы помогут вам лучше освоить методы моделирования.

Глава 5

Сущности

Я — Чеви Чейз... а ты — нет.
Чеви Чейз

Разработчики часто фокусируются на данных, а не предметной области. Это может произойти с теми, кто плохо знаком с принципами DDD и отдает предпочтение традиционным подходам к разработке программного обеспечения, которые помещают во главу угла базу данных. Вместо того чтобы разрабатывать понятия предметной области с содержательным поведением, они думают прежде всего об атрибутах (столбцах) и связях (внешних ключах) данных. В этом случае модель данных отображается в объекты и в результате почти каждое понятие в нашей "модели предметной области" кодируется как **СУЩНОСТЬ**, изобилующая методами `set` и `get`. Найти инструменты, которые генерируют все это для нас, очень просто. Несмотря на то что в методах доступа к свойствам нет ничего плохого, это не единственное поведение, которое должны иметь СУЩНОСТИ DDD.

В этом и заключается ловушка, в которую попали разработчики из компании SaaSOvation. Извлеките уроки из их ошибок, сделанных при проектировании СУЩНОСТИ.

Назначение главы

- Показать пользу СУЩНОСТЕЙ в ситуациях, когда необходимо моделировать уникальные вещи.
- Показать, как можно сгенерировать уникальный идентификатор для сущности.
- Рассказать, как команда вырабатывает **ЕДИНЫЙ ЯЗЫК (1)** при проектировании СУЩНОСТЕЙ.
- Научить выражать роли и функции СУЩНОСТЕЙ.
- Продемонстрировать примеры оценки СУЩНОСТЕЙ и способы их хранения в хранилище.

Зачем нужны СУЩНОСТИ

Мы проектируем понятие предметной области как СУЩНОСТЬ, когда нас интересует его индивидуальность и его отличие от всех других объектов в системе является обязательным условием. СУЩНОСТЬ является уникальной вещью, способной непрерывно изменяться в течение длительного периода времени. Изменения могут быть столь значительными, что объект может казаться очень отличающимся от того, каким он был раньше. И все же по сути это тот же самый объект.

Когда СУЩНОСТЬ изменяется, мы можем следить за тем, когда, как и кем были внесены эти изменения. Кроме того, если нас устраивает его текущая форма, возникающая в результате изменения его предыдущих состояний, то явное отслеживание изменений становится ненужным. Даже если мы решим не отслеживать всю историю изменений ОБЪЕКТОВ, мы можем обсуждать последовательность допустимых изменений, которые могли произойти с этими объектами за все время существования. Именно уникальный идентификатор и характеристики изменчивости отличают СУЩНОСТИ от **ОБЪЕКТОВ-ЗНАЧЕНИЙ (VALUE OBJECTS) (6)**.

Иногда СУЩНОСТЬ не подходит для моделирования. Некорректное использование СУЩНОСТЕЙ происходит намного чаще, чем многие думают. Часто концепцию следует моделировать как ЗНАЧЕНИЕ. Если вы с этим не согласны, то, возможно, принципы DDD не соответствуют вашим бизнес-потребностям. Вполне возможно, что система, основанная на операциях CRUD, больше соответствовала бы ситуации. В этом случае такое решение должно сэкономить время и деньги для вашего проекта. Проблема состоит в том, что следование альтернативам, основанным на операциях CRUD, не всегда сохраняет эти драгоценные ресурсы.

Компании регулярно прикладывают слишком много усилий для разработки хваленых редакторов таблицы базы данных. Без правильного выбора инструмента решения, основанные на операциях CRUD, оказываются слишком дорогими. Когда целесообразно использовать операции CRUD, следует применять такие языки и платформы, как Groovy, Grails, Ruby on Rails и т.п. Если выбор сделан правильно, он должен сэкономить время и деньги.

Ковбойская логика

AJ: На какую это гадость[1] я наступил?

LB: Это коровья лепешка!

AJ: Я знаю, что такое лепешка. Бывает хлебная лепешка, бывает сырная. Это — не лепешка.

LB: Как говорится, не бросайся коровьими лепешками в жаркий день. Хорошо, что ты ее не бросил.

[1] Игра слов: одно из значений слова "crud" — "гадость". — *Примеч. ред.*

С другой стороны, если применять операции CRUD неправильно — т.е. в более сложных системах, которые требуют использования принципов DDD, — можно пожалеть об этом. По мере возрастания сложности мы будем все сильнее испытывать ограниченность выбранного нами инструмента. Системы CRUD не позволяют создать точную бизнес-модель лишь на основе данных.

Если вы можете позволить себе инвестиции в подход DDD, то правильно применяйте СУЩНОСТИ.

> Если объект определяется уникальным индивидуальным существованием, а не набором атрибутов, это свойство следует считать главным при определении объекта в модели. Определение класса должно быть простым и строиться вокруг непрерывности и уникальности цикла существования объекта. Найдите способ различать каждый объект независимо от его формы или истории существования. В модели должно даваться точное определение, ***что такое одинаковые объекты*** [Эванс, с. 98].

В этой главе мы покажем, как сделать правильный акцент на СУЩНОСТЯХ, и продемонстрируем методы проектирования СУЩНОСТЕЙ.

Уникальный идентификатор

На ранних стадиях проектирования СУЩНОСТЕЙ мы в основном сосредоточиваемся на их основных атрибутах и функциях, благодаря которым его можно однозначно идентифицировать и найти по запросу, и мы намеренно будем игнорировать все остальные атрибуты и функции, пока не опишем главные.

> Вместо того чтобы сосредоточиться на атрибутах или даже на рабочих функциях объекта, следует ограничить определение **ОБЪЕКТА-СУЩНОСТИ** наиболее неотъемлемыми его характеристиками, т.е. теми, которые однозначно выделяют его среди других и обычно используются для его поиска и сравнения. Задавайте только те рабочие функции, которые существенны для создания понятия об объекте, и только те атрибуты, которых требуют эти функции [Эванс, с. 98].

Итак, вот что мы должны сделать в первую очередь. Наличие диапазона доступных параметров для реализации идентификатора действительно важно, как и обеспечение уникальности с течением времени.

Уникальный идентификатор СУЩНОСТИ может быть практичным инструментом для поиска и сравнения, а может и не быть им. Использование уникального идентификатора для сравнения обычно зависит от того, насколько он удобочитаем. Например, если приложение выполняет поиск имен людей, доступных для

пользователей, то очень маловероятно, что это имя будет использоваться как уникальный идентификатор СУЩНОСТИ класса `Person`. Имена разных людей часто совпадают. С другой стороны, если приложение осуществляет поиск налогового идентификационного номера компаний, то этот номер может служить главным уникальным идентификатором СУЩНОСТИ класса `Company`. Уникальные налоговые идентификаторы устанавливаются правительствами.

Уникальные идентификаторы можно хранить в ОБЪЕКТАХ-ЗНАЧЕНИЯХ. Эти объекты являются неизменяемыми, что обеспечивает постоянство идентификаторов, а любая функция, связанная с обработкой идентификаторов, является централизованной. Впрочем, можно достаточно просто локализовать функцию для работы с идентификаторами, чтобы предотвратить утечку информации в другие части модели и к другим клиентам.

Рассмотрим несколько распространенных стратегий создания идентификаторов, от самой простой до более сложных.

- Пользователь вводит в приложение одно или несколько уникальных значений. Приложение должно проверить, являются ли они уникальными.
- Приложение автоматически генерирует идентификатор, используя алгоритм, гарантирующий его уникальность. Для этого можно использовать библиотеку или каркас, а можно сделать это и в самом приложении.
- Приложение обращается к постоянному хранилищу, например к базе данных, и генерирует уникальный идентификатор.
- Уникальный идентификатор уже определен другим **ОГРАНИЧЕННЫМ КОНТЕКСТОМ (2)** (системой или приложением). Он может вводиться пользователем или выбираться им из доступного множества вариантов.

Рассмотрим отдельные стратегии вместе со специфичными проблемами, которые с ними связаны. При рассмотрении диапазона технических решений почти всегда возникают побочные эффекты. Один такой побочный эффект возникает, когда мы используем реляционные базы данных для хранения объектов, которые проникают в наши модели предметной области. Мы не заостряем внимание на проблемах создания идентификаторов, выборе моментов для генерации идентификатора, идентичности ссылок на объекты предметной области в реляционной базе данных и роли объектно-реляционного отображения (object-relational mapping — ORM) в этой ситуации. Мы дадим также несколько практических советов, касающихся поддержки постоянства уникальных идентификаторов.

Идентификатор вводится пользователем

Кажется очевидным разрешить пользователям вручную вводить детали уникального идентификатора. Пользователь вводит распознаваемое значение или символ в поле ввода или выбирает из ряда доступных характеристик, в результате чего создается СУЩНОСТЬ. Правда, это довольно простой подход. Однако могут возникнуть осложнения.

Одно из осложнений состоит в том, что качество идентификаторов зависит от пользователей. Идентификатор может быть уникальным, но некорректным. В большинстве случаев идентификатор должен быть неизменным, следовательно, пользователи не должны изменять их. Так бывает не всегда, поэтому иногда целесообразно позволить пользователям исправлять значения идентификатора. Приведем пример. Что произойдет, если мы использовали заголовки объектов классов `Forum` и `Discussion` как уникальные идентификаторы, а пользователь набрал заголовок неправильно или позже решил, что заголовок недостаточно точный, как показано в рис. 5.1? Какова стоимость этих изменений? Несмотря на то что предоставление пользователям возможности вводить идентификатор может показаться весьма бюджетным вариантом, это может оказаться неправильным. Можно ли положиться на пользователей, чтобы создать уникальные и корректные, постоянные идентификаторы?

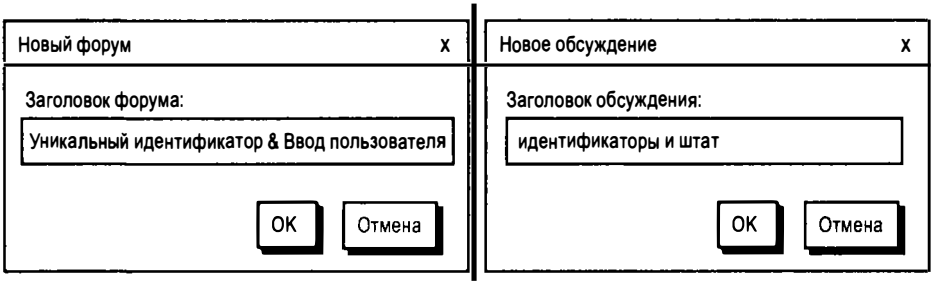


Рис. 5.1. Заголовок форума содержит опечатки, а заголовок обсуждения совершенно неприемлем

Предотвращение этой проблемы начинается с обсуждения проекта. Команды должны рассмотреть отказоустойчивые методы, позволяющие пользователям определять уникальные идентификаторы. Проверка идентификаторов в ходе бизнес-процесса неприемлема в предметных областях с высокой пропускной способностью, но отлично работает в ситуациях, когда удобочитаемые идентификаторы являются необходимостью. Если для создания и утверждения идентификаторов, которые будут использоваться в бизнесе в течение многих последующих лет, требуются дополнительное время и усилия и к тому же возможна поддержка

бизнес-процесса, то добавление нескольких дополнительных циклов, чтобы гарантировать качество идентификаторов, стоит затрат.

Всегда существует возможность включить введенные пользователями значения как свойства СУЩНОСТИ, доступные для проверки соответствия, но не использовать их для создания уникальных идентификаторов. Простые свойства легче модифицировать как часть нормального рабочего состояния СУЩНОСТИ, которое со временем изменяется. В этом случае мы должны будем использовать другие средства для получения уникальных идентификаторов.

Идентификатор генерируется приложением

Существуют высоконадежные способы автоматического генерирования уникальных идентификаторов, несмотря на необходимость соблюдать осторожность, когда приложение кластеризируется или иначе распределяется по многочисленным вычислительным узлам. Существуют шаблоны создания идентификатора, которые с намного большей степенью уверенности могут создавать абсолютно уникальные идентификаторы. Одним из таких подходов является универсально уникальный идентификатор (universally unique identifier — UUID) или глобально уникальный идентификатор (globally unique identifier — GUID). Ниже приведен общепринятый вариант, в котором результаты каждого шага объединяются в единое текстовое представление.

1. Время в миллисекундах на вычислительном узле.
2. IP-адрес вычислительного узла.
3. Идентификатор фабричного объекта в виртуальной машине (Java).
4. Случайное число, сгенерированное тем же самым генератором в виртуальной машине (Java).

В результате возникает 128-битовое уникальное значение. Чаще всего оно выражается в виде 32- или 36-байтовой шестнадцатеричной текстовой строки. Текст имеет 36-байтовый вид, когда в качестве разделителя сегмента используется дефис в формате `f36ab21c-67dc-5274-c642-1de2f4d5e72a`. Если дефисы не используются, текстовая строка имеет 32-байтовый формат. В любом случае идентификатор большой и неудобочитаемый.

В среде Java эта формула была заменена стандартным генератором UUID, доступным начиная с версии Java 1.5. Он обеспечивается классом `java.util.UUID`. Эта реализация поддерживает четыре различных алгоритма генератора на основе варианта Лича–Зальца (Leach–Salz). Используя стандартный прикладной интерфейс языка Java, мы можем легко генерировать псевдослучайные уникальные идентификаторы.

```
String rawId = java.util.UUID.randomUUID().toString();
```

Здесь используется тип 4, предусматривающий криптографически сильный генератор случайных чисел, основанный на генераторе `java.security.SecureRandom`. Тип 3 реализует шифрование имени, используя класс `java.security.MessageDigest`. Получить имя, основанное на идентификаторе UUID, можно примерно так.

```
String rawId = java.util.UUID.nameUUIDFromBytes(
 "Некоторый текст".getBytes()).toString();
```

Можно также использовать генератор псевдослучайных чисел с шифрованием.

```
SecureRandom randomGenerator = new SecureRandom();

int randomNumber = randomGenerator.nextInt();

String randomDigits = new Integer(randomNumber).toString();

MessageDigest encryptor = MessageDigest.getInstance("SHA-1");

byte[] rawIdBytes = encryptor.digest(randomDigits.getBytes());
```

Теперь осталось только решить задачу преобразования массива `rawIdBytes` в шестнадцатеричное текстовое представление. Мы могли бы получить это преобразование просто так. После генерации случайного числа и его преобразования в объект класса `String` мы передаем этот текст методу `UUIDnameUUIDFromBytes()` из шаблона **ФАБРИКА** [Гамма и др.].

Существуют и другие возможности генерирования идентификаторов, такие как `java.rmi.server.UID` и `java.rmi.dgc.VMID`, но они уступают классу `java.util.UUID` и поэтому здесь не рассматриваются.

Идентификатор UUID генерируется относительно быстро, не требуя никакого взаимодействия с внешней стороной, такой как механизм постоянного хранения. Даже если определенный вид СУЩНОСТИ создается много раз в секунду, генератор идентификатора UUID может поддерживать высокий темп. Для предметных областей более высокой производительности мы можем кешировать любое число экземпляров UUID, пополняя кеш в фоновом режиме. Если кешированные экземпляры UUID потеряны из-за перезапуска сервера, то между идентификаторами не возникает никаких разрывов, потому что они все основаны на случайных, искусственных значениях. Повторное заполнение кеша на перезапуске сервера не создает никаких негативных последствий, связанных с потерянными значениями.

Генерирование таких больших идентификаторов непрактично из-за перегрузки памяти. В таких случаях проблему могут решить 8-байтовые идентификаторы, сгенерированные механизмом постоянного хранения. Еще более эффективным может оказаться меньшее, 4-байтовое целое число с приблизительно двумя миллиардами уникальных значений. Эти подходы обсуждаются далее.

Глядя на приведенную ниже строку, мы, очевидно, не будем отображать значения идентификаторов UUID в окнах нашего пользовательского интерфейса.

```
f36ab21c-67dc-5274-c642-1de2f4d5e72a
```

Полный идентификатор UUID обычно удобен, когда его можно скрыть от пользователей и применить удобочитаемые и понятные человеку ссылки. Например, мы можем разработать ресурсы гипермедиа с идентификаторами URI, которые можно посылать по электронной почте или отправлять с помощью другого механизма передачи сообщений пользователям. Текстовая часть ссылки может использоваться для маскировки таинственно выглядящего идентификатора UUID, так же, как строка `text` в дескрипторе `<a> text </a>` маскирует детали HTML-ссылки.

В зависимости от уровня доверия к уникальности отдельных сегментов шестнадцатеричного текста идентификатора UUID, вы можете использовать всего один или несколько сегментов целого идентификатора. Сокращенный идентификатор лучше защищен, когда он используется только как *локальный идентификатор* СУЩНОСТЕЙ в границах **АГРЕГАТА (10)**. Локальный идентификатор означает, что СУЩНОСТИ, сохраненные в АГРЕГАТЕ, являются уникальными лишь относительно других СУЩНОСТЕЙ в том же АГРЕГАТЕ. С другой стороны, СУЩНОСТЬ, служащая КОРНЕМ АГРЕГАТА, требует наличия глобального уникального идентификатора.

Наш собственный генератор идентификатора мог бы использовать один или несколько конкретных сегментов идентификатора UUID. Рассмотрим искусственный пример: `APM-P-08-14-2012-F36AB21C`. Этот 25-символьный идентификатор представляет объект класса `Product` (`P`) из *Контекста управления гибким проектированием* (`APM`), созданного 14 августа 2012 года. Дополнительный текст `F36AB21C` — это первый сегмент сгенерированного идентификатора UUID, который отличает данный объект от всех остальных СУЩНОСТЕЙ класса `Product`, созданных в этот день. Это вполне удобочитаемый идентификатор с высокой вероятностью глобальной уникальности. Это удобно не только для пользователей. Когда такие идентификаторы оказываются в ОГРАНИЧЕННЫХ КОНТЕКСТАХ, разработчики сразу видят, откуда они появились. Для компании SaaSOvation этот подход может оказаться практичным, поскольку АГРЕГАТЫ еще более разделяются по арендаторам.

Хранить такие идентификаторы в объектах класса `String`, вероятно, — не лучшее решение. Целесообразнее использовать специальный ОБЪЕКТ-ЗНАЧЕНИЕ.

```
String rawId = "APM-P-08-14-2012-F36AB21C"; // генерируется
ProductId productId = new ProductId(rawId);
...
Date productCreationDate = productId.creationDate();
```

Клиент может запросить детали идентификатора, такие как дата создания продукта, и этот запрос можно легко выполнить. Клиенты не обязаны поддерживать необработанный формат идентификатора. Теперь КОРЕНЬ АГРЕГАТА класса `Product` может представить свою дату создания, не указывая клиентам, как он получен.

```
public class Product extends Entity {
    private ProductId productId;
    ...
    public Date creationDate() {

        return this.productId().creationDate();
    }
    ...
}
```

Задачу генерирования идентификатора можно решить с помощью сторонних библиотек и каркасов. Проект Apache Commons содержит компонент Commons Id ("песочница"), в котором предусмотрены пять разных генераторов идентификаторов.

Некоторые постоянные хранилища, такие как NoSQL Riak и MongoDB, также могут генерировать идентификаторы. Обычно для того чтобы сохранить значение в системе Riak, используется команда HTTP `PUT`, получающая ключ.

```
PUT /riak/bucket/key

[объект сериализации]
```

Вместо этого можно использовать команду `POST`, не передавая ей ключ и вынуждая систему Riak генерировать уникальный идентификатор. Впрочем, эти возможности заслуживают более подробного обсуждения, которое мы проведем в этой главе позднее.

Что служит ФАБРИКОЙ идентификаторов, генерируемых приложением? Для генерирования идентификатора КОРНЯ АГРЕГАТА я предпочитаю использовать его **ХРАНИЛИЩЕ (12)**.

```
public class HibernateProductRepository
        implements ProductRepository {
    ...
    public ProductId nextIdentity() {
        return new ProductId(
                java.util.UUID.randomUUID().toString().toUpperCase());
    }
    ...
}
```

ХРАНИЛИЩЕ кажется естественным местом для генерирования идентификаторов.

Идентификатор генерируется механизмом постоянного хранения

Делегирование задачи генерирования уникального идентификатора механизму постоянного хранения имеет преимущество, которого нет у других способов. Если мы обратимся к базе данных, чтобы получить последовательность или инкрементированное значение, они всегда будут уникальными.

В зависимости от диапазона, база данных может генерировать уникальные 2-, 4- или 8-байтовые значения. В языке Java 2-байтовое короткое целое число допускает до 32 767 уникальных идентификаторов, 4-байтовое обычное целое число позволяет получить 2 147 483 647 уникальных значений, а 8-байтовое длинное целое число обеспечивает до 9 223 372 036 854 775 807 разных идентификаторов. Даже заполненные незначащими нулями представления этих диапазонов являются достаточно короткими и содержат пять, десять и 19 символов соответственно. Это свойство можно использовать для создания составных идентификаторов.

Единственным возможным недостатком этого способа является производительность. Обращение к базе данных за каждым значением может отнять намного больше времени, чем генерирование идентификаторов в приложении. Многое зависит от загрузки сервера базы данных и требований приложения. Одно из решений этой проблемы состоит в том, чтобы кешировать значения последовательности или инкрементированных значений в приложении, например в ХРАНИЛИЩЕ. Это может оказаться хорошим решением, но обычно мы сталкиваемся с потерей большого количества неиспользованных значений при перезагрузке сервера базы данных. Если эти разрывы в нумерации, вызванные потерянным кешем, недопустимы или если вы запланировали лишь относительно небольшое количество значений (2-байтовое короткое целое), кеширование предварительно определенных значений может оказаться непрактичным или даже ненужным. Вероятно, собрать и восстановить потерянные идентификаторы и можно, но игра часто не стоит свеч.

Предварительное определение и кеширование не являются проблемой, если модель допускает позднее генерирование идентификаторов. Посмотрим, как получить последовательность идентификаторов в библиотеке Hibernate системы Oracle.

```
<id name="id" type="long" column="product_id">
    <generator class="sequence">
        <param name="sequence">product_seq</param>
    </generator>
</id>
```

Приведем пример того же самого подхода, но с использованием столбца с автоматической инкрементацией в системе MySQL.

```
<id name="id" type="long" column="product_id">
    <generator class="native"/>
</id>
```

Этот подход работает хорошо и допускает простое конфигурирование при определении отображений в системе Hibernate. Правда, может возникнуть проблема выбора времени генерирования, которая будет обсуждаться далее. В оставшейся части этого подраздела мы рассмотрим требование раннего генерирования идентификаторов.

Порядок может иметь значение

Иногда имеет значение момент, когда происходят генерирование идентификатора и его присваивание СУЩНОСТИ. *Раннее* генерирование и присваивание происходят до того, как сохранена СУЩНОСТЬ. *Позднее* генерирование и присваивание происходят, когда СУЩНОСТЬ уже сохранена.

Рассмотрим поддержку раннего генерирования с помощью ХРАНИЛИЩА, обслуживающего следующую доступную последовательность в системе Oracle по запросу.

```
public ProductId nextIdentity() {
    Long rawProductId = (Long)
        this.session()
            .createSQLQuery(
                "select product_seq.nextval as product_id from dual")
            .addScalar("product_id", Hibernate.LONG)
            .uniqueResult();

    return new ProductId(rawProductId);
}
```

Поскольку система Oracle возвращает последовательность значений, которые библиотека Hibernate отображает в экземпляры класса `BigDecimal`, необходимо информировать библиотеку Hibernate о том, что мы хотим преобразовать результат `product_id` в объект класса `Long`.

Что же делать с базами данных, такими как MySQL, которые не поддерживают последовательности? База MySQL поддерживает столбцы с автоматической нумерацией. Обычно автоматическая нумерация не происходит, пока в таблицу не будет вставлена новая строка. Однако существует способ заставить базу MySQL выполнить автоматическую нумерацию в стиле Oracle.

```
mysql> CREATE TABLE product_seq (nextval INT NOT NULL);
Query OK, 0 rows affected (0.14 sec)

mysql> INSERT INTO product_seq VALUES (0);
Query OK, 1 row affected (0.03 sec)

mysql> UPDATE product_seq SET nextval=LAST_INSERT_ID(nextval + 1);
Query OK, 1 row affected (0.03 sec)
Rows matched: 1 Changed: 1 Warnings: 0

mysql> SELECT LAST_INSERT_ID();
+------------------+
| LAST_INSERT_ID() |
+------------------+
|                1 |
+------------------+
1 row in set (0.06 sec)

mysql> SELECT * FROM product_seq;
+---------+
| nextval |
+---------+
|       1 |
+---------+
1 row in set (0.00 sec)
```

Мы создали в базе данных MySQL таблицу с именем `product_seq`. Затем мы вставили в таблицу одну строку, инициализировав нулями ее единственный столбец. Эти первые два шага устанавливают эмулятор последовательности для ОБЪЕКТА класса `Product`. Следующие два оператора демонстрируют генерацию единственного значения последовательности. Мы обновляем единственную строку, увеличивая значения в столбце `nextval` на единицу. Инструкция обновления использует функцию MySQL `LAST_INSERT_ID()`, чтобы постепенно увеличить значение `INT` в столбце. Сначала вычисляется выражение, заданное параметром, а его результат присваивается столбцу `nextval`. Результат выражения `nextval + 1` в функции `LAST_INSERT_ID()` остается постоянным, и после выполнения инструкции `SELECT LAST_INSERT_ID()` возвращается значение

nextval. Наконец для проверки мы можем вычислить оператор SELECT * FROM product_seq, чтобы доказать, что текущее значение nextval совпадает с результатом, возвращенным функцией.

В библиотеке Hibernate 3.2.3 для генерирования переносимых последовательностей используется генератор org.hibernate.id.enhanced.SequenceStyleGenerator, который поддерживает только позднее генерирование идентификаторов (когда СУЩНОСТЬ уже вставлена в хранилище). Для поддержки раннего генерирования последовательности в ХРАНИЛИЩЕ потребуется создать специальный запрос к библиотеке Hibernate или интерфейсу JDBC. Рассмотрим повторную реализацию метода nextIdentity() из класса ProductRepository для базы MySQL.

```
public ProductId nextIdentity() {
    long rawId = -1L;
    try {
        PreparedStatement ps =
            this.connection().prepareStatement(
               "update product_seq "
               + "set next_val=LAST_INSERT_ID(next_val + 1)");

        ResultSet rs = ps.executeQuery();

       try {
           rs.next();
           rawId = rs.getLong(1);
       } finally {
           try {
           rs.close();
           } catch(Throwable t) {
             // игнорируем
           }
       }
    } catch (Throwable t) {
        throw new IllegalStateException(
                "Cannot generate next identity", t);
    }

    return new ProductId(rawId);
}
```

Если используется интерфейс JDBC, нет необходимости выполнять второй запрос к базе данных, чтобы получить результаты функции LAST_INSERT_ID(). Всю работу выполняет запрос на обновление. Мы получаем значение типа long из объекта класса ResultSet, используя его для создания объекта класса ProductId.

Последний трюк обеспечивает связь с интерфейсом JDBC из библиотеки Hibernate. Это сложно, но возможно.

```
private Connection connection() {
    SessionFactoryImplementor sfi =
           (SessionFactoryImplementor)sessionFactory;

    ConnectionProvider cp = sfi.getConnectionProvider();
    return cp.getConnection();
}
```

Без объекта Connection мы не можем получить объект класса ResultSet, выполнив инструкцию PreparedStatement. А без него невозможно получить переносимую последовательность.

Используя переносимые последовательности из Oracle, MySQL и других баз данных, мы получаем инструмент для генерирования более компактных и гарантированно уникальных идентификаторов, поддерживающих раннее генерирование.

Идентификатор присваивается другим ограниченным контекстом

Если идентификатор присваивается другим ОГРАНИЧЕННЫМ КОНТЕКСТОМ, необходимо выполнить интеграцию, чтобы найти, сравнить и присвоить каждый идентификатор. Интеграция в подходе DDD объясняется в главах, посвященных **КАРТАМ КОНТЕКСТОВ (3)** и **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**.

Наиболее желательным является точное совпадение. В этом случае пользователи должны указать один или несколько атрибутов, таких как номер счета, имя пользователя, адрес электронной почты или другое уникальное значение, чтобы точно определить намеченный результат.

Часто соответствие допускает нечеткую вводную информацию, приводящую к многозначным результатам поиска, предусматривающего выбор с участием человека. Это продемонстрировано на рис. 5.2. Пользователь вводит критерий "найти нечто подобное" с использованием знака подстановки для искомой СУЩНОСТИ. Мы получаем доступ к прикладному интерфейсу API внешнего ОГРАНИЧЕННОГО КОНТЕКСТА, который выполняет поиск одинаково описанных объектов. Затем пользователь выбирает определенный результат из нескольких вариантов. Идентификатор выбранного объекта используется в качестве локального идентификатора. Из внешней СУЩНОСТИ в локальную СУЩНОСТЬ можно также скопировать некоторое дополнительное состояние (свойство).

Рис. 5.2. Результаты поиска идентификатора во внешней системе. Поисковый интерфейс пользователя не обязательно отображает идентификатор. В этом примере он его отображает

Это создает последствия для синхронизации. Что произойдет, если изменение объектов, на которые есть внешние ссылки, оказывает влияние на локальные СУЩНОСТИ? Как узнать, что ассоциированный объект изменился? Эту проблему можно решить с помощью **СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЫ (4)** и **СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**. Наш локальный ОГРАНИЧЕННЫЙ КОНТЕКСТ подписывается на СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, опубликованные внешними системами. После получения соответствующего уведомления наша локальная система выполняет изменение своих СУЩНОСТЕЙ АГРЕГАТА, чтобы отразить их состояние во внешних системах. Иногда синхронизация должна инициироваться локальным ОГРАНИЧЕННЫМ КОНТЕКСТОМ, а изменения должны передаваться в инициирующую внешнюю систему.

Обычно это непросто сделать, но в результате возникает большее количество автономных систем. После достижения автономии поиск можно сузить до локальных объектов. Это не локальное кеширование внешних объектов. Скорее это трансляция внешних концепций в концепции внутри локального ОГРАНИЧЕННОГО КОНТЕКСТА, как объяснено в главе, посвященной **КАРТАМ КОНТЕКСТА (3)**.

Это одна из самым сложных стратегий создания идентификаторов. Поддержка локальной СУЩНОСТИ зависит не только от изменений, вызванных локальными функциями предметной области, но и, возможно, от поведения одной или нескольких внешних систем. Используйте этот подход максимально консервативно.

Если время генерирования идентификатора имеет значение

Генерация идентификатора может произойти либо рано, как часть процесса создания объекта, либо поздно, как часть механизма хранения объекта. Иногда необходимо генерировать идентификаторы рано, а иногда поздно. Если это имеет значение, мы должны понять, что с этим связано.

Рассмотрим простейший вариант, в котором допускается позднее определение идентификатора, когда новая СУЩНОСТЬ уже сохранена, т.е. новая строка уже вставлена в базу данных. Эта ситуация демонстрируется на рис. 5.3. Клиент просто создает новый экземпляр класса `Product` и добавляет его в объект класса `ProductRepository`. Если экземпляр класса `Product` создан недавно, клиент не нуждается в его идентификаторе. Это хорошо, в том числе и потому, что идентификатор не нужен. Он понадобится только после того, как экземпляр будет сохранен.

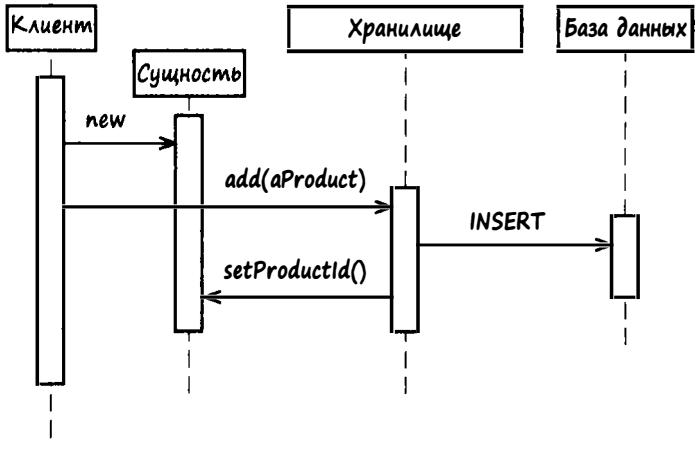


Рис. 5.3. Простейший способ определения уникального идентификатора — заставить хранилище данных генерировать его при первом сохранении объекта

Почему время может иметь значение? Рассмотрим сценарий, в котором клиент подписывается на исходящие СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ. Когда выполняется создание нового экземпляра класса `Product`, происходит СОБЫТИЕ. Клиент сохраняет опубликованное СОБЫТИЕ в **ХРАНИЛИЩЕ СОБЫТИЙ (EVENT STORE) (8)**. В конечном счете сохраненные СОБЫТИЯ публикуются как уведомления, которые достигают подписчиков за пределами ОГРАНИЧЕННОГО КОНТЕКСТА. Если применяется подход, представленный на рис. 5.3, клиент получает СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ прежде, чем у него появляется возможность добавить новый экземпляр класса `Product` в экземпляр класса `ProductRepository`. Таким образом, СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ не

может содержать корректный идентификатор нового экземпляра класса `Product`. Для того чтобы СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ было правильно инициализировано, генерация идентификатора должна быть выполнена рано. Этот подход продемонстрирован на рис. 5.4. Клиент запрашивает из экземпляра класса `ProductRepository` следующий идентификатор, передавая его конструктору класса `Product`.

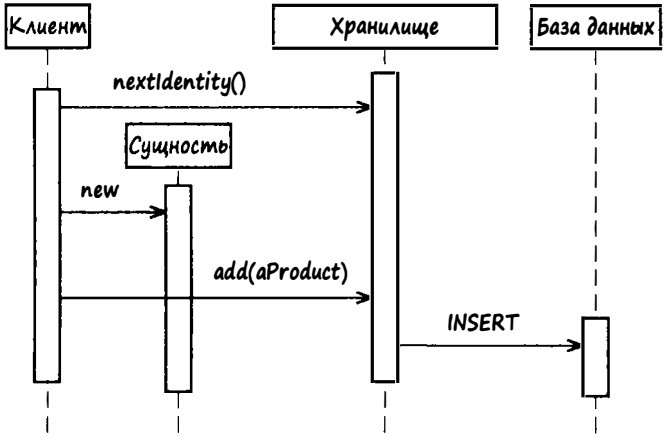


Рис. 5.4. В данном случае идентификатор запрашивается из ХРАНИЛИЩА и присваивается во время создания объекта. Сложность генерирования идентификатора скрыта в реализации ХРАНИЛИЩА

Существует другая проблема, которая может возникнуть при позднем генерировании идентификатора, когда СУЩНОСТЬ еще не сохранена. Она возникает, когда две или более новые СУЩНОСТИ должны быть добавлены в множество `java.util.Set`, но их идентификаторы еще не были присвоены и поэтому совпадают с другими новыми идентификаторами (например, `null`, `0` или `-1`). Если метод `equals()`, принадлежащий СУЩНОСТИ, сравнит идентификаторы, то СУЩНОСТИ, недавно добавленные в множество `Set`, окажутся одинаковыми. В множестве будет содержаться только первый добавленный объект, а все другие будут удалены. В результате возникает ошибка, первопричину которой трудно понять и устранить.

Для того чтобы избежать этой ошибки, мы должны сделать одну из двух вещей. Следует либо изменить проект, чтобы рано определять и присваивать идентификаторы, либо осуществить рефакторинг метода `equals()`, чтобы сравнивать атрибуты, отличающиеся от идентификаторов. При выборе рефакторинга метода `equals()` его следует реализовывать так, будто СУЩНОСТЬ является ОБЪЕКТОМ-ЗНАЧЕНИЕМ. В этом случае метод `hashCode()` следует согласовать с методом `equals()` в одном и том же объекте.

```
public class User extends Entity {
   ...
   @Override
   public boolean equals(Object anObject) {
       boolean equalObjects = false;
       if (anObject != null &&
             this.getClass() == anObject.getClass()) {
       User typedObject = (User) anObject;
       equalObjects =
            this.tenantId().equals(typedObject.tenantId()) &&
            this.username().equals(typedObject.username()));
       }
       return equalObjects;
    }

    @Override
    public int hashCode() {
       int hashCode =
           + (151513 * 229)
           + this.tenantId().hashCode()
           + this.username().hashCode();

      return hashCode;
    }
    ...
}
```

В многоарендной среде экземпляр класса `TenantId` также рассматривается как часть уникального идентификатора. Никакие два объекта класса `User` при разных подписчиках класса `Tenant` не должны считаться одинаковыми.

Следует подчеркнуть, что сталкиваясь с ситуацией, требующей добавления сущностей в множество `Set`, я предпочитаю подход, основанный на раннем определении и присвоении. Для СУЩНОСТЕЙ более желательно иметь методы `equals()` и `hashCode()`, основанные на уникальных идентификаторах объектов, а не на других атрибутах.

Суррогатный идентификатор

Некоторые инструменты ORM, такие как Hibernate, хотят иметь дело с идентичностью объекта на собственных условиях. Библиотека Hibernate в качестве основного идентификатора каждой СУЩНОСТИ предпочитает оригинальный тип базы данных, такой как числовая последовательность. Если предметная область требует другого вида идентичности, это вызывает нежелательный конфликт для библиотеки Hibernate. Для того чтобы исправить этот недостаток, мы должны использовать два идентификатора. Один из них предназначен для модели предметной области и подчиняется ее требованиям. Другой предназначен для библиотеки Hibernate и известен как *суррогатный идентификатор* (surrogate identity).

Создать суррогатный идентификатор довольно просто. Создайте атрибут СУЩНОСТИ для хранения типа суррогата. Обычно на эту роль подходят типы `long` и `int`. Кроме того, создайте столбец в базе данных сущностей, чтобы сохранить в нем уникальный идентификатор, и наложите на него ограничения первичного ключа. Затем включите в определение отображения в библиотеке Hibernate элемент `<id>`. Напоминаем, что в этом случае с идентификатором предметной области ничего не происходит. Он создается только для удовлетворения требования инструмента ORM, т.е. библиотеки Hibernate.

Суррогатный идентификатор лучше всего скрыть от внешнего мира. Поскольку этот суррогат не является частью модели предметной области, его визуальное представление нарушило бы условия постоянного хранения. Несмотря на то что нарушения некоторых из этих условий избежать невозможно, все же можно предпринять несколько мер для сокрытия суррогата от разработчиков модели и ее клиентов.

Одна из мер предосторожности использует шаблон **СУПЕРТИП УРОВНЯ (LAYER SUPERTYPE)** [Fowler, P of EAA].

```
public abstract class IdentifiedDomainObject
        implements Serializable {

    private long id = -1;

    public IdentifiedDomainObject() {
        super();
    }

    protected long id() {
        return this.id;
    }

    protected void setId(long anId) {
        this.id = anId;
    }
}
```

Этим СУПЕРТИПОМ УРОВНЯ является класс `IdentifiedDomainObject`, абстрактный базовый класс, скрывающий суррогатный первичный ключ от представления клиентов с помощью защищенных (`protected`) методов доступа. Клиентов не должно интересовать, предназначены ли эти методы для них, потому что они невидимы за пределами **МОДУЛЯ (9)** СУЩНОСТИ, расширяющей данный базовый класс. Эти методы можно даже объявить закрытыми (`private`). Библиотека Hibernate свободно использует отражение метода или поля на любом уровне видимости, `public` или `private`. Дополнительные СУПЕРТИПЫ УРОВНЯ могут добавлять значения, например для поддержки оптимистичных схем параллельной работы, как показано в главе, посвященной **АГРЕГАТАМ (10)**.

Необходимо отобразить атрибут суррогата `id` в столбец базы данных с помощью определения библиотеки Hibernate. В данном случае класс `User` имеет атрибут `id`, который отображается в столбец базы данных с именем `id`.

```
<hibernate-mapping default-cascade="all">
    <class
    name="com.saasovation.identityaccess.domain.model.identity.User"
   table="tbl_user" lazy="true">

      <id
          name="id"
          type="long"
          column="id"
          unsaved-value="-1">

          <generator class="native"/>
       </id>
       ...

    </class>
</hibernate-mapping>
```

Вот как выглядит определение таблицы MySQL, предназначенной для хранения объектов класса `User`.

```
CREATE TABLE 'tbl_user' (
     'id' int(11) NOT NULL auto_increment,
     'nablement_enabled' tinyint(1) NOT NULL,
     'enablement_end_date' datetime,
     'enablement_start_date' datetime,
     'password' varchar(32) NOT NULL,
     'tenant_id_id' varchar(36) NOT NULL,
     'username' varchar(25) NOT NULL,
     KEY 'k_tenant_id_id' ('tenant_id_id'),
     UNIQUE KEY `k_tenant_id_username` ('tenant_id_id','username'),
     PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

Первый столбец, `id`, является суррогатным идентификатором. В последней инструкции в определении таблицы столбец `id` объявляется первичным ключом. Мы можем различить суррогатный идентификатор и идентификатор предметной области. Существует два столбца, `tenant_id_id` и `username`, обеспечивающие уникальный идентификатор для предметной области. Они объединяются в одно целое, образуя уникальный ключ с именем `k_tenant_id_username`.

Идентификатор предметной области не обязан играть роль первичного ключа базы данных. Мы разрешили использовать суррогатный идентификатор `id` в качестве первичного ключа базы данных, чтобы удовлетворить требования библиотеки Hibernate.

Суррогатные первичные ключи баз данных можно использовать во всей модели данных как внешние ключи других таблиц, обеспечивая ссылочную целостность. Для этого на предприятии, возможно, придется организовать систему управления данными (например, аудиторскими) или другую инструментальную поддержку. Ссылочная целостность важна и для библиотеки Hibernate, когда таблицы переплетаются между собой для реализации разнообразных отображений "многие-ко-многим" (например, 1:M). Они также поддерживают объединение таблиц для оптимизации запросов при чтении АГРЕГАТОВ из баз данных.

Постоянство идентификатора

В большинстве случаев уникальный идентификатор необходимо защитить от модификации, обеспечив его постоянство на протяжении существования сущности, которой он присвоен.

Для предотвращения модификации идентификатора можно предпринять простые меры. Мы можем скрыть методы-установщики идентификатора от клиентов. Мы можем также создать предохранители в этих методах для предотвращения изменения состояния, в том числе и самой СУЩНОСТИ, если она уже существует. Предохранители кодируются как утверждения в методах-установщиках СУЩНОСТИ. Рассмотрим пример метода-установщика идентификатора.

```
public class User extends Entity {
   ...
   protected void setUsername(String aUsername) {
       if (this.username != null) {
           throw new IllegalStateException(
                   "The username may not be changed.");
       }
       if (aUsername == null) {
           throw new IllegalArgumentException(
                    "The username may not be set to null.");
       }
       this.username = aUsername;
    }
    ...
}
```

В этом примере атрибут `username` является идентификатором предметной области по отношению к СУЩНОСТИ класса `User`. Он изменяется только один раз и только автоматически. Метод-установщик `setUsername()` обеспечивает инкапсуляцию, скрытую от клиентов. Когда открытая функция СУЩНОСТИ делегирует себя методу установщику, этот метод проверяет атрибут `username`, чтобы проверить, не равен ли он `null`. Если он уже не равен `null`, что свидетельствует о неизменяемом инвариантном состоянии, генерируется исключение

IllegalStateException. Исключение свидетельствует о том, что атрибут username поддерживается как состояние, которое изменяется только один раз.

Заметки на доске

- Укажите настоящие СУЩНОСТИ из вашей предметной области и запишите их имена.
- Назовите их уникальные идентификаторы, как предметной области, так и суррогатный. Не считаете ли вы, что для генерирования идентификаторов лучше было бы использовать другие способы или другие моменты времени?
- Затем для каждой СУЩНОСТИ укажите, не следовало ли использовать другой подход к присвоению идентификаторов — с помощью пользователя, приложения, механизма постоянного хранения или другого ОГРАНИЧЕННОГО КОНТЕКСТА — и почему (даже если сейчас вы не можете ничего изменить)?

Затем отметьте для каждой сущности, необходимо ли ей раннее генерирование идентификатора или достаточно позднего, и объясните, почему.

- Оцените стабильность каждого идентификатора и решите, не следует ли ее повысить.

Данный метод-установщик не мешает библиотеке Hibernate, когда она воссоздает состояние объекта, извлеченного их хранилища. Так как объект сначала создается с помощью конструктора по умолчанию, не имеющего аргументов, атрибут username изначально равен null. Это позволяет выполнить повторную инициализацию, и метод-установщик разрешает библиотеке Hibernate одноразовое присваивание. Без этого можно обойтись, если библиотека Hibernate получает указание использовать поле (атрибут) для постоянного хранения и восстановления, а не для доступа.

Тест подтверждает, что предохранитель, допускающий одноразовое изменение, правильно защищает состояние идентификатора объекта класса User.

```
public class UserTest extends IdentityTest {
    ...
    public void testUsernameImmutable() throws Exception {
         try { User user = this.userFixture();
         user.setUsername("testusername");
         fail("The username must be immutable after. ¿
initialization.");
         } catch (IllegalStateException e) {
```

```
            // ожидаемая неудача
        }
    }
    ...
}
```

Этот демонстрационный пример показывает, как работает модель. При успешном выполнении он доказывает, что метод `setUsername()` предотвращает изменение существующего идентификатора, не равного `null`. (Предохранители СУЩНОСТЕЙ и тесты более подробно будут рассмотрены в разделе, посвященном проверке корректности.)

Выявление сущностей и их внутренних характеристик

Теперь перейдем к выводам, которые можно извлечь из опыта компании SaaSOvation...

Сначала команда, разрабатывающая проект CollabOvation, совершила ошибку, погрузившись в программирование на языке Java модели "сущность–связь" (ER — entity-relationship). Члены команды слишком сосредоточились на базе данных, таблицах и столбцах, а также их отображении в объекты. Это привело к созданию АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ [Fowler, Anemic], состоящей из множества методов получателей и установщиков. Им следовало больше думать о подходе DDD. К тому времени, когда им потребовалось разработать систему безопасности, как описано в главе, посвященной ОГРАНИЧЕННЫМ КОНТЕКСТАМ (2), они поняли, что следует сосредоточиться на выработке ЕДИНОГО ЯЗЫКА. Это привело к хорошим результатам. В этом разделе мы покажем, какие уроки усвоила команда *Контекста идентификации и доступа*.

ЕДИНЫЙ ЯЗЫК в отдельном ОГРАНИЧЕННОМ КОНТЕКСТЕ содержит все концепции и термины, необходимые для разработки модели предметной области. Этот язык не появляется внезапно. Он вырабатывается в ходе подробных обсуждений с экспертами предметной области и после обдумывания требований, предъявляемых к системе. Часть терминологии состоит из имен существительных, называющих вещи, прилагательных, которые их описывают, и глаголов, указывающих, что эти вещи делают. Было бы ошибкой думать, что все сводится только к набору имен классов и глаголов, обозначающих операции. Такой ограниченный подход

снижает гибкость и глубину модели. Отсутствие ограничений на дискуссии и изучение спецификаций поможет выработать ЯЗЫК, отражающий важные концепции и являющийся результатом усилий, соглашений и компромиссов. В итоге команда будет разговаривать на ЯЗЫКЕ полноценными предложениями, а модель будет четко отражать высказывания на этом ЯЗЫКЕ.

В этом сценарии очень важно зафиксировать результаты дискуссий в неформальном документе. На первых этапах ваш ЕДИНЫЙ ЯЗЫК будет иметь форму глоссария и набора простых сценариев использования. Впрочем, ошибочно предполагать, что этого достаточно. В конце концов ЯЗЫК будет моделироваться кодом, и сохранить согласованность документации окажется трудно или даже невозможно.

Выявление сущностей и свойств

Рассмотрим очень простой пример. Команда SaaSOvation пришла к выводу, что в *Контексте идентификации и доступа* необходимо моделировать сущность `User`. Этот пример не относится к **СМЫСЛОВОМУ ЯДРУ (2)**, но к этому примеру мы перейдем позднее, а пока будем игнорировать дополнительную сложность, которая присуща СМЫСЛОВОМУ ЯДРУ, и сосредоточимся на более простой СУЩНОСТИ. В качестве учебного пособия этот пример вполне подходит.

Посмотрим, что известно компании о СУЩНОСТИ `User` в терминах требования к программному обеспечению (а не пользовательских сценариев или пользовательских историй), которые примерно соответствуют выражениям на ЕДИНОМ ЯЗЫКЕ. Им необходимо уточнить следующие моменты.

- Пользователи находятся под контролем арендаторов и связаны с ними.
- Пользователи и системы должны быть авторизованы.
- Пользователи имеют персональную информацию, включая имя и контактную информацию.
- Персональная информация о пользователе может быть изменена самим пользователем или менеджером.
- Сертификаты безопасности пользователей (пароли) можно изменять.

Команда внимательно читала и слушала. Как только члены команды видели или слышали различные формы слова "*изменить*", они были вполне уверены, что имели дело по крайней мере с одной СУЩНОСТЬЮ. Конечно, слово "изменить" могло также означать "заменить ЗНАЧЕНИЕ", а не "изменить СУЩНОСТЬ". Было ли что-либо еще, что мешало команде сделать выбор структурного элемента? Было. Ключевое

слово "*аутентифицируется*", которое является верным признаком того, что следует разрешить поиск. Если у вас есть набор вещей, и одна из вещей должна быть найдена среди многих, вам нужны уникальные идентификаторы, позволяющие отличить ее от остальных. В результате поиска должен быть найден один из многих пользователей, связанных с арендатором.

А что можно сказать об инструкции, относящейся к арендаторам, управляющими пользователями? Не следует ли из этого, что реальная СУЩНОСТЬ — это объект класса `Tenant`, а не `User`? Это открывает дискуссию об **АГРЕГАТАХ (10)**, которую мы ненадолго отложим. Короче говоря, ответ — "и да, и нет". Да, есть СУЩНОСТЬ класса `Tenant`, и нет, это не означает, что нет СУЩНОСТИ класса `User`. Они обе являются СУЩНОСТЯМИ. Объяснение, почему объекты класса `Tenant` и `User` являются **КОРНЯМИ (10)** двух различных АГРЕГАТОВ, будет приведено ниже в этой главе. И да, классы `User` и `Tenant` в конечном счете являются типами АГРЕГАТОВ, но команда сначала пренебрегла этим обстоятельством.

Объяснение состоит в том, что каждый объект класса `User` должен быть однозначно идентифицирован и четко отличаться от остальных. Объект класса `User` должен также изменяться со временем, таким образом, совершенно очевидно, что это СУЩНОСТЬ. В данный момент не имеет значения, как мы моделируем персональные данные в классе `User`.

Команда должна уделить внимание уточнению смысла первого ограничения.

- Пользователи находятся под контролем арендаторов и связаны с ними.

Сначала команда должна была просто добавить примечание или как-то изменить формулировку инструкции, чтобы отразить тот факт, что арендаторы владеют пользователями, *но не собирают и не содержат их*. Команда должна быть осторожной, чтобы не попасть в паутину технических и тактических проблем. Инструкции должны иметь смысл для всей команды. Они формулируются следующим образом.

- Арендаторы допускают регистрацию многих пользователей по приглашению.
- Арендаторы могут быть активными и неактивными.
- Пользователи системы должны быть авторизованы, но могут авторизоваться, только если арендатор является активным.
- . . .

Вот это было сюрпризом! В ходе дальнейшего обсуждения команда аккуратно решила лингвистические проблемы и одновременно намного прояснила смысл требований. Члены команды пришли к выводу, что исходная инструкция, касающаяся пользователей, находящихся под управлением арендаторов, была неполной. Фактом является то, что пользователи регистрируются на правах аренды и только по приглашению. Было также важно указать, что арендаторы могут быть активными

или неактивными и что пользователи могут аутентифицироваться, только когда их аренда активна. Полное повторение одного требования, добавление другого и разъяснение третьего позволили намного точнее определить то, что фактически происходит.

Это усилие устранило любые возможные осложнения, связанные с факторами, управляющими жизненным циклом пользователей, но прояснило, что независимо от того, кто владеет пользователями, некоторые пользователи при определенных обстоятельствах могут быть недоступными. На текущий момент эти сценарии были важными.

В этот момент казалось, что команда заложила основы глоссария терминов ЕДИНОГО ЯЗЫКА. Однако у них не было достаточной информации для формулировки подробных определений. Для этого ей необходимо некоторое время делать больше записей в глоссарии.

У членов команды было несколько известных СУЩНОСТЕЙ, как показано на рис. 5.5. Важно было узнать, как их однозначно определить и какие дополнительные свойства могут потребоваться, для того чтобы найти их среди многих возможных объектов того же типа.

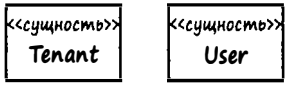


Рис. 5.5. Две СУЩНОСТИ: `Tenant` и `User`, выявленные на фазе раннего обнаружения

Команда решила использовать полный идентификатор UUID для уникальной идентификации каждого объекта класса `Tenant`. В этом случае идентификатор генерируется приложением. Обосновать выбор большого текстового значения было легко. Это не только гарантировало уникальность идентификатора, но и повышало безопасность каждого подписчика. Было бы трудно случайно воспроизвести идентификатор UUID для получения полного доступа к приватным данным. Они также обнаружили потребность явно выделить СУЩНОСТИ, принадлежащие каждому объекту класса `Tenant`, от тех, которые принадлежат кому-то другому. Такое требование направлено на повышение безопасности приложений и служб подписчиков, принадлежащих арендаторам — конкурентоспособным компаниям. Таким образом, каждая СУЩНОСТЬ во всех системах "привязывалась" к этим уникальным идентификаторам и каждый запрос требовал уникального идентификатора для поиска СУЩНОСТИ, какой бы она ни была.

Уникальный идентификатор арендатора не является СУЩНОСТЬЮ. Это ЗНАЧЕНИЕ определенного вида. Вопрос заключается в том, должен ли этот идентификатор иметь специальный тип или он должен иметь просто тип `String`?

Похоже, что нет необходимости моделировать применение **ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS) (6)** к идентификаторам. Идентификатор представляет собой обычное шестнадцатеричное текстовое представление большого числа. Однако это представление должно использоваться широко. Оно должны быть установлено во всех остальных СУЩНОСТЯХ в каждом КОНТЕКСТЕ. В этом случае привлекательной может оказаться строгая типизация. Определив

ОБЪЕКТ-ЗНАЧЕНИЕ типа `TenantId`, команда может усилить гарантии того, что все СУЩНОСТИ, принадлежащие подписчику, будут связаны с правильным идентификатором. Моделирование этой ситуации показано на рис. 5.6, на котором продемонстрированы сущности классов `Tenant` и `User`.

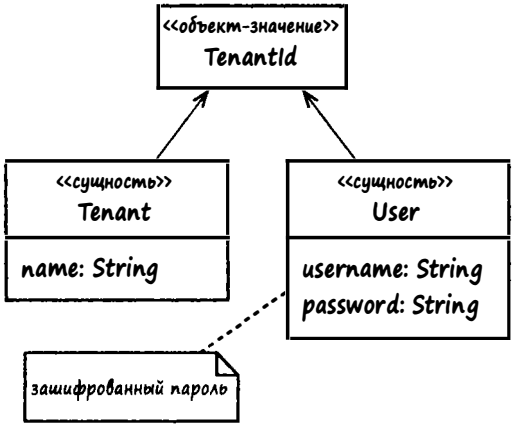


Рис. 5.6. После выявления и именования СУЩНОСТИ следует выяснить атрибуты и свойства, однозначно идентифицирующие ее, и обеспечить их поиск

Объект класса `Tenant` должен иметь имя. Атрибут `name` должен иметь тип `String`, потому что у него нет других специальных функций. Атрибут `name` помогает выполнять запросы. Сотрудник справочного бюро должен сначала найти объект класса `Tenant` по атрибуту `name`, прежде чем сможет оказать помощь. Это необходимый атрибут и "внутренняя характеристика". Атрибут `name` можно сделать уникальным среди всех остальных подписчиков, но пока это не важно.

С каждым подписчиком могут быть связаны и другие атрибуты. К ним относятся контракт на обслуживание, PIN-код активизации вызова, информация о счетах и платежах и, возможно, адрес компании и другая контактная информация. Но все это относится к бизнесу, а не к безопасности. Попытка слишком сильно расширить *Контекст идентификации и доступа* может оказаться неудачным решением.

Поддержкой можно управлять с помощью другого КОНТЕКСТА. Найдя арендатора по имени, программное обеспечение может использовать его уникальный идентификатор `TenantId`. Затем его можно использовать для доступа, например, к *Контексту поддержки*, *Контексту счетов* или *Контексту управления взаимоотношениями с клиентами*. Контракты на обслуживание, адрес компании и контактная информация клиентов имеют мало отношения к вопросам безопасности. Впрочем, связывание имени подписчика с объектом класса `Tenant` поможет персоналу справочного бюро быстро оказать необходимую поддержку. В то же время атрибут `name` относится к вопросам безопасности.

Завершив анализ СУЩНОСТИ класса `Tenant`, команда переключила свое внимание на СУЩНОСТЬ класса `User`. Что может служить его уникальным идентификатором? Большинство систем идентификации используют уникальные имена

пользователей. Поскольку имя пользователя внутри арендатора является уникальным, не имеет значения, из чего он состоит. (У разных арендаторов имена пользователей не обязаны быть уникальными.) Выбор имени предоставляется самим пользователям. Если подписчики придерживаются определенных правил при выборе имен или имена выбираются интегрированной системой безопасности, решение этих вопросов относится к компетенции пользователя. Команда просто объявляет атрибут `username` в классе `User`.

Одно из ограничений утверждает, что существует сертификат безопасности. Оно утверждает, что им является пароль. Команда уточнила терминологию и объявила атрибут `password` в классе `User`. Она пришла к выводу, что атрибут `password` не следует хранить в виде открытого текста. Команда решила зашифровать атрибут `password`. Таким образом, ей необходим способ шифрования каждого пароля, прежде чем он будет связан с объектом класса `User`. Это похоже на разновидность **СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICE) (7).** Команда зарезервировала место в глоссарии ЕДИНОГО ЯЗЫКА, к разработке которого она приступила. Глоссарий должен быть небольшим, но полезным.

- **Арендатор.** Именованная организация, являющаяся подписчиком служб идентификации и доступа, а также других интерактивных служб. Поддерживает регистрацию пользователя с помощью приглашения.
- **Пользователь.** Зарегистрированное действующее лицо системы безопасности внутри арендатора, имеющее персональное имя и контактную информацию. Пользователь имеет уникальное имя и зашифрованный пароль.
- **Служба шифрования.** Обеспечивает средства шифрования паролей и других данных, которые нельзя хранить и использовать в виде открытого текста.

Остался один вопрос: следует ли рассматривать атрибут `password` как часть уникального идентификатора объекта класса `User`? В конце концов, он используется для поиска объектов класса `User`. Если да, то, вероятно, целесообразно объединить два этих атрибута в ЦЕЛОСТНОЕ ЗНАЧЕНИЕ, назвав его, например, `SecurityPrincipal`. В этом случае концепция станет намного более явной. Это интересная идея, но она противоречит важному ограничению: пароли могут изменяться. Кроме того, иногда службы должны искать объект класса `User`, не зная пароля. Это не относится к аутентификации. (Рассмотрите сценарий, в котором необходимо проверить, играет ли пользователь какую-либо роль в системе безопасности. Мы не можем каждый раз требовать пароль, когда нужно найти объект класса `User`, чтобы проверить его полномочия.) Это не идентификатор. При этом мы можем по-прежнему включать атрибуты `username` и `password` в один запрос аутентификации.

Идея создать тип ЗНАЧЕНИЯ `SecurityPrincipal` приводит к соблазнительному предположению, касающемуся моделирования. Этот вопрос был отложен. Кроме того, неисследованными остались некоторые другие концепции, например как обеспечить приглашение к регистрации, а также детали персонального имени и контактную информацию. Команда должна решить эти проблемы на следующей итерации.

Поиски важных функций

После того как были выявлены важные атрибуты, команда должна выявить функции, без которых нельзя обойтись. . .

Оглядываясь на основные требования, полученные командой, разработчики задумались о поведении объектов класса `Tenant` и `User`.

- Арендаторы могут быть активными и неактивными.

Размышляя об активизации и деактивизации объекта класса `Tenant`, вы, вероятно, представляете себе булев переключатель. Впрочем, как именно его реализовать, пока неважно. Какая польза будет от того, что мы включим атрибут `active` в класс `Tenant` на диаграмме классов? Раскрывает ли следующее объявление атрибутов в файле `Tenant.java` намерения проектировщиков?

```
public class Tenant extends Entity {
   ...
   private boolean active;
   ...
```

Вероятно, не совсем. Для начала мы хотим сосредоточиться только на атрибутах и свойствах, определяющих идентификатор и позволяющих выполнять запросы. Детали этой поддержки будут приведены далее.

Команда может решить объявить метод `setActive(boolean)` без учета терминологии требования. Это не значит, что открытые методы-установщики никогда нельзя использовать. Это значит лишь, что их следует использовать только тогда, когда ЯЗЫК это допускает и когда нежелательно использовать несколько методов-установщиков для выполнения одного запроса. Несколько методов-установщиков вызывают неопределенность в намерениях разработчиков. Они также усложняют публикацию отдельного осмысленного СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ как результат выполнения отдельной логической команды.

Работая над ЯЗЫКОМ, команда отметила, что эксперты предметной области говорят об активизации и деактивизации. Для того чтобы учесть эту терминологию, они могли бы предусмотреть операции `activate()` и `deactivate()`.

Следующий исходный код реализует шаблон **ИНФОРМАТИВНЫЙ ИНТЕРФЕЙС (INTENTION REVEALING INTERFACE)** [Эванс] и соответствует терминологии ЕДИНОГО ЯЗЫКА.

```
public class Tenant extends Entity {
    ...
    public void activate() {

    // ЗАДАНИЕ: реализовать
    }

    public void deactivate() {

    // ЗАДАНИЕ: реализовать
}
...
```

Для иллюстрации своих идей команда сначала разработала тест, чтобы увидеть, как используются новые функции.

```
public class TenantTest ... {
    public void testActivateDeactivate() throws Exception {
        Tenant tenant = this.tenantFixture();
        assertTrue(tenant.isActive());

        tenant.deactivate();

        assertFalse(tenant.isActive());

        tenant.activate();
        assertTrue(tenant.isActive());
    }
}
```

После этого теста команда почувствовала уверенность в качестве интерфейса. Разработка теста заставила их понять, что был необходим другой метод, `isActive()`. Они сосредоточились на этих трех новых методах, как указано на рис. 5.7. Глоссарий ЕДИНОГО ЯЗЫКА также расширился.

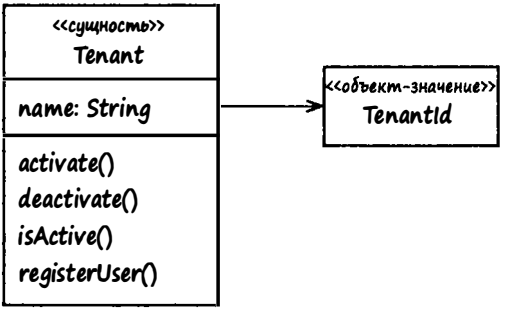


Рис. 5.7. На первой итерации в класс `Tenant` включена необходимая функция. Некоторые функции пока пропущены из-за их сложности

- **Активизация арендатора.** С помощью этой операции можно активизировать арендатора и подтвердить текущее состояние.
- **Деактивизация арендатора.** С помощью этой операции можно деактивизировать арендатора. После деактивизации арендатора пользователи не могут аутентифицироваться.
- **Служба аутентификации.** Координирует аутентификацию пользователей, в первую очередь, проверяя активность владеющего ими арендатора.

Последняя запись, добавленная в глоссарий, свидетельствует о выявлении новой СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Перед попыткой сравнения экземпляра класса `User` кто-то должен сначала проверить с помощью функции `isActive()`, что объект класса `Tenant` является активным. Это стало ясным после анализа следующего требования.

- Пользователи системы должны аутентифицироваться, но это возможно, только если арендатор является активным.

Поскольку аутентификация не сводится только к простому поиску объекта класса `User` с заданными атрибутами `username` и `password`, необходим координатор более высокого уровня. Для этого хорошо подходят СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Детали будут раскрыты позднее. Пока важно отметить, что команда зафиксировала имя службы `AuthenticationService` и добавила его в ЕДИНЫЙ ЯЗЫК. Подход, основанный на первоочередном тестировании, принес свои плоды.

Команда рассмотрела также следующее требование.

- Арендаторы допускают регистрацию многих пользователей по приглашению.

Когда члены команды приступили к более глубокому анализу этого ограничения, они поняли, что ситуация немного сложнее, чем хотелось бы на первой итерации. Казалось, что в систему требовалось включить что-то вроде объекта класса `Invitation`. Однако требование было сформулировано недостаточно четко. Неясной также оставалась функция, управляющая приглашениями. Таким образом, команда отложила моделирование этого требования до получения разъяснений от экспертов предметной области и первых клиентов. И все же члены команды определили метод `registerUser()`. Это было важно для создания экземпляров класса `User` (см. раздел "Конструирование" далее в главе).

После этого они вернулись к классу `User`.

- Пользователи имеют персональную информацию, включая имя и контактную информацию.
- Персональная информация о пользователе может быть изменена самим пользователем или менеджером.
- Сертификаты безопасности пользователей (пароли) можно изменять.

ПОЛЬЗОВАТЕЛЬ и ОСНОВНОЙ ИДЕНТИФИКАТОР — это два шаблона безопасности, которые часто используются вместе.[2] Из термина *персональный* следует, что объект класса `User` имеет отдельное концептуальное представление. На основе предыдущих утверждений команда разработала композицию шаблонов и функций.

[2] См. мои опубликованные шаблоны: `http://vaughnvernon.co/`.

Person моделируется как отдельный класс, чтобы не перегружать обязанностями класс User. Слово *персональный* натолкнуло команду на мысль добавить класс Person в ЕДИНЫЙ ЯЗЫК.

- **Класс** Person**.** Содержит персональные данные объекта класса User, включая имя и контактную информацию, и управляет ими.

Является ли объект класса Person СУЩНОСТЬЮ или ОБЪЕКТОМ-ЗНАЧЕНИЕМ? Здесь снова ключевым является слово *изменяет*. Кажется нецелесообразным заменять весь объект класса Person только потому, что у человека может измениться номер телефона. Как показано на рис. 5.8, команда спроектировала СУЩНОСТЬ, в которой хранятся два ЗНАЧЕНИЯ, ContactInformation и Name. В настоящее время эти понятия являются нечеткими и со временем должны быть уточнены.

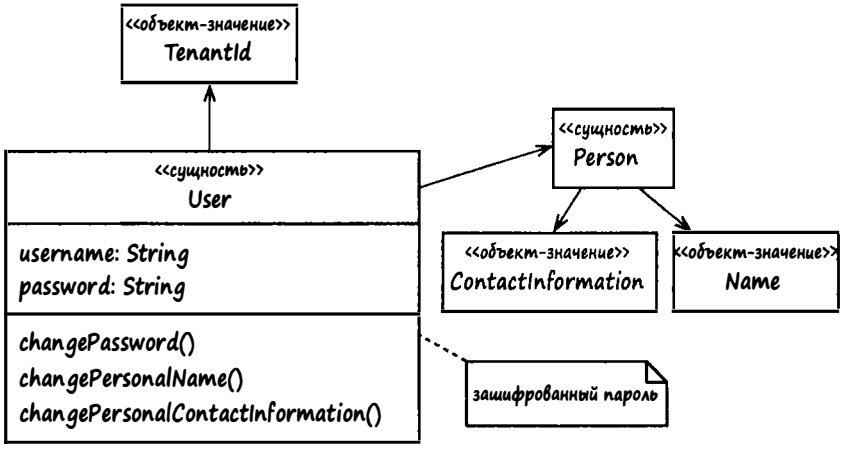


Рис. 5.8. Основные функции класса User избавляют от лишних ассоциаций. Не стремясь к излишней специализации, команда разработала немного больше объектов и операций

Для обсуждения способов управления изменениями персонального имени и контактной информации было организовано новое совещание. Должны ли клиенты получить доступ к объекту класса Person внутри класса User? Один из разработчиков спросил, всегда ли объект класса User является человеком. Что если он представляет собой внешнюю систему? В настоящее время такая возможность не рассматривалась и ее следовало изучить в будущем, но в любом случае она заслуживала внимания. Если клиент получит доступ к конфигурации класса User и сможет перейти к объекту класса Person, чтобы выполнить функцию, потребуется дальнейший рефакторинг.

Если бы вместо этого члены команды смоделировали персональное поведение класса User, делая его более обобщенным по отношению к субъекту системы безопасности, то они, вероятно, избежали бы части последующих волнений. После того как они написали некоторые иллюстративные тесты для исследования этого понятия, решение казалось правильным. Они смоделировали класс User, как показано на рис. 5.8.

Были и другие соображения. Должна ли команда вообще открыть класс Person или скрыть его от всех клиентов? Пока она решила оставить класс Person открытым для запроса информации. Впоследствии функцию доступа можно было бы перепроектировать, чтобы обслуживать интерфейс класса Principal, а классы User и System должны были бы стать специализированными версиями класса Principal. Более глубоко вникнув в проблему, команда должна была быть в состоянии осуществить рефакторинг.

Работая согласованно, команда быстро выработала ЕДИНЫЙ ЯЗЫК с учетом последнего из рассматриваемых требований.

- Сертификаты безопасности пользователей (пароли) можно изменять.

В классе User есть функция changePassword(). Она отражает термин, используемый в требованиях, и одобрена экспертами предметной области. Клиентам никогда не предоставляется доступ даже к зашифрованным паролям. После того как в объекте класса User устанавливается пароль, он никогда не раскрывается за пределами АГРЕГАТА. Все, кто хотят быть аутентифицированными, должны использовать только службу AuthenticationService.

Команда также решила, что все функции, которые могут вызвать успешную модификацию, должны публиковать СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. Это решение оказалось более сложным, чем команда предполагала. Однако они действительно распознали потребность в СОБЫТИЯХ. События служили бы по крайней мере двум целям. Во-первых, они позволили бы отслеживать изменения на протяжении жизненного цикла всех объектов (эта тема обсуждается ниже). Во-вторых, они позволили бы внешним подписчикам сохранять синхронизацию с изменениями, предоставляя внешним объектам возможность автономии.

Эти темы обсуждаются в главах, посвященных **СОБЫТИЯМ (8)** и **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**.

Роли и обязанности

Один из аспектов моделирования предусматривает выявление ролей и обязанностей объектов. В принципе, анализ ролей и обязанностей применяется к предметной области, однако здесь мы специально рассмотрим роли и обязанности СУЩНОСТЕЙ.

Для термина *роль* нужен некоторый контекст. Например, при обсуждении *Контекста идентификации и доступа* объект класса Role — это СУЩНОСТЬ и КОРЕНЬ АГРЕГАТА, предназначенного для решения широкого спектра проблем, связанных с системой безопасности. Клиенты могут спросить, играет ли пользователь какую-то роль в системе безопасности. Это полностью отличается от того, о чем я говорю. В этом разделе я говорю о том, какие роли могут играть объекты в вашей модели.

Объекты предметной области играют несколько ролей

В объектно-ориентированном программировании роль класса обычно определяют интерфейсы. Если класс разработан правильно, у него есть одна роль для каждого интерфейса, который он реализует. Если у класса нет явно объявленных ролей — он не реализует явных интерфейсов, — то по умолчанию у него все равно есть роль его класса, т.е. неявный интерфейс его открытых методов. Класс `User` в предыдущих примерах не реализует явных интерфейсов, но при этом играет свою роль — класса `User`.

Можно создать объект, который одновременно будет играть две роли — класса `User` и `Person`. Не считайте это рекомендацией, но предположим, что мы одобрили эту идею. В этом случае нет причин агрегировать отдельный объект класса `Person` как ссылочную связь с объектом класса `User`. Вместо этого достаточно иметь один объект, играющий обе роли.

Почему мы можем это сделать? Обычно потому что мы видим сходство и различия между двумя или более объектами. Перекрытие характеристик можно реализовать путем смешения нескольких интерфейсов в одном объекте. Например, мы могли бы иметь объект, который одновременно является объектом классов `User` *и* `Person`, назвав класс реализации `HumanUser`.

```
public interface User {
    ...
}

public interface Person {
    ...
}

public class HumanUser implements User, Person {
    ...
}
```

Есть ли в этом смысл? Возможно, но могут возникнуть и осложнения. Если оба интерфейса сложные, то может быть трудно реализовать их в одном объекте. Кроме того, объект класса `User` может оказаться системой, что увеличит количество необходимых интерфейсов до трех. Разработка одного объекта, играющего роль классов `User`, `Person` и `System`, может оказаться еще сложнее. Возможно, ее можно было бы упростить, создав универсальный интерфейс `Principal`.

```
public interface User {
    ...
}

public interface Principal {
    ...
}
```

```
public class UserPrincipal implements User, Principal {
   ...
}
```

В рамках этого проекта мы пытаемся определить реальный тип субъекта (principal) на этапе выполнения (позднее связывание). У субъекта-человека и субъекта-системы разные реализации. Системы не нуждаются в том же виде контактной информации, что и человек. Однако так или иначе мы могли бы попробовать разработать реализацию пересыльного делегирования. Для этого мы проверили бы существование одного или другого типа на этапе выполнения и делегировали бы функции существующему объекту.

```
public interface User {
   ...
}

public interface Principal {
   public Name principalName();
   ...
}

public class PersonPrincipal implements Principal {
   ...
}

public class SystemPrincipal implements Principal {
   ...
}

public class UserPrincipal implements User, Principal {
   private Principal personPrincipal;
   private Principal systemPrincipal;
   ...
   public Name principalName() {

      if (personPrincipal != null) {
         return personPrincipal.principalName();
      } else if (systemPrincipal != null) {
         return systemPrincipal.principalName();
      } else {
         throw new IllegalStateException(
                 "The principal is unknown.");
      }
    }
    ...
}
```

Эта схема порождает несколько проблем. Во-первых, она страдает от так называемой *объектной шизофрении* (object schizophrenia).[3] Функция делегируется с помощью механизма, который называется пересылкой или диспетчеризацией. Ни объект `personPrincipal`, ни объект `systemPrincipal` не отражает идентичности СУЩНОСТИ `UserPrincipal`, который в итоге выполняет эту функцию. Объектная шизофрения описывает ситуацию, в которой делегируемый объект не знает, кто его делегирует. Делегаты не знают, кем они в действительности являются. Это не значит, что для получения идентификатора базового объекта в каждом из конкретных классов потребуется метод делегата, но такая ситуация вполне возможна. Мы могли бы передать ссылку на объект класса `UserPrincipal`. Однако это усложняет проект и требует изменения интерфейса класса `Principal`. Это плохо. Как утверждают авторы книги [Гамма и др.], "Делегирование можно считать хорошим выбором только тогда, когда оно позволяет достичь упрощения, а не усложнения".

Мы не будем здесь решать эту проблему моделирования. Она лишь иллюстрирует сложности, которые могут возникать при использовании ролей объектов, и подчеркивает, что стиль моделирования следует выбирать осторожно. Улучшить ситуацию позволяют правильно выбранные инструменты, такие как Qi4j [Öberg].

Иногда целесообразно уточнить интерфейсы, как предлагает Уди Дахан (Udi Dahan) [Dahan, Roles]. Укажем два требования, которые стимулируют уточнение интерфейса.

- Добавлять новые заказы к покупателю.
- Делать покупателя привилегированным (без объяснений, как это сделать).

Класс `Customer` реализует две детализированные роли интерфейса: `IAddOrdersToCustomer` и `IMakeCustomerPreferred`. Как показано на рис. 5.9, каждая из них определяет только одну операцию. Мы даже можем реализовывать другие интерфейсы, например `Ivalidator`.

[3] Эта схема описывает объект с несколькими личностями, что не является определением шизофрении с медицинской точки зрения. Реальная проблема, скрывающаяся за путаницей имен, заключается к путанице идентичностей объекта.

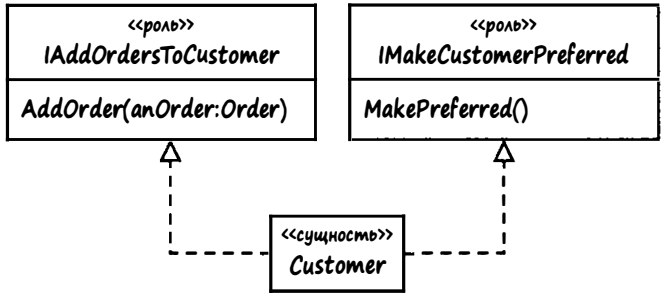


Рис. 5.9. Используя соглашения об именах C#.NET, СУЩНОСТЬ класса `Customer` реализует две роли объекта: `IAddOrdersToCustomer` и `IMakeCustomerPreferred`

Как указывается в главе, посвященной **АГРЕГАТАМ (10)**, обычно мы не можем собирать в классе `Customer` большое количество объектов, например все заказы. По этой причине данный пример следует считать искусственным и предназначенным только для иллюстрации использования ролей объектов.

Префикс `I` в имени интерфейса — это стиль, широко используемый в программировании на платформе .NET. Кроме того, он подсказывает такой вариант прочтения имени: "я добавляю заказы к покупателю" и "я делаю покупателя привилегированным". Без префикса `I` мы получили бы менее информативные имена, звучащие, как приказы: `AddOrdersToCustomer` и `MakeCustomerPreferred`. Впрочем, можно было бы подумать об именах, состоящих только из существительных или прилагательных. Такой стандарт тоже вполне возможен.

Рассмотрим некоторые преимущества этого стиля. Роль СУЩНОСТИ может изменяться в зависимости от пользовательского сценария. Если клиенту необходимо добавить в объект класса `Customer` новый объект класса `Order`, его роль отличается от роли, когда он хочет сделать объект класса `Customer` привилегированным. Кроме того, существуют технические преимущества. Разные пользовательские сценарии могут требовать специализированных стратегий получения данных.

```
IMakeCustomerPreferred customer =
    session.Get<IMakeCustomerPreferred>(customerId);
customer.MakePreferred();



IAddOrdersToCustomer customer =
    session.Get<IAddOrdersToCustomer>(customerId);
customer.AddOrder(order);
```

Механизм постоянного хранения запрашивает параметризованное имя типа `T` для метода `Get<T>()`. Он использует этот тип для поиска ассоциированной стратегии получения данных, зарегистрированной в инфраструктуре. Если у интерфейса нет специальной стратегии получения данных, то используется стратегия, установленная по умолчанию. Реализуя стратегию получения данных, идентифицированный объект класса `Customer` загружается в форме, необходимой для конкретного пользовательского сценария.

Мы можем видеть технические преимущества, поскольку интерфейсы, маркирующие роли, предоставляют возможность использования закулисных механизмов. Поведение, характерное для другого пользовательского сценария, можно связать с любой другой ролью, предусматривая выполнение специального метода для проверки модификаций СУЩНОСТИ, предназначенной для постоянного хранения.

Детализированные интерфейсы облегчают реализацию поведения самим классом реализации, например классом `Customer`. В этом случае нет необходимости делегировать реализацию отдельным классам, избегая объектной шизофрении.

Возникает вопрос: дает ли разделение функций класса `Customer` *преимущество при моделировании предметной области*? Сравните предыдущий вариант класса `Customer` с вариантом, показанным на рис. 5.10; какой из них лучше? Легче ли клиенту ошибочно вызвать метод `AddOrder()`, в то время как на самом деле он должен вызвать метод `MakePreferred()`? Вероятно, нет. Однако на этот вопрос нельзя ответить однозначно.

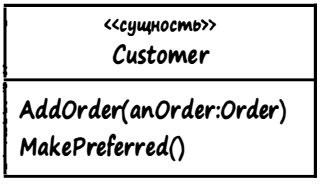


Рис. 5.10. Здесь класс `Customer` моделируется с операциями, которые ранее были разбросаны по разным интерфейсам, а теперь собраны в одном интерфейсе класса СУЩНОСТИ

Возможно, наиболее практичное применение ролевых интерфейсов является также самым простым. Мы можем усилить интерфейсы, чтобы скрыть детали реализации, которые не хотим открывать для клиентов. Разработайте интерфейс, в котором открыто только то, что мы хотим позволить клиентам использовать, и ничего больше. Класс реализации может быть намного более сложным, чем интерфейс. Он может иметь все виды поддержки свойств с помощью методов доступа (`get-` и `set-`) и реализованные функциональные возможности, о которых клиенты никогда не будут получать никакой информации. Например, возможно,

что инструмент или платформа вынуждает создавать открытые методы, и мы не хотим, чтобы клиенты их использовали. Несмотря на это, интерфейс модели предметной области не зависит от технических деталей реализации. В этом заключается определенное преимущество моделирования предметной области.

Принимая любое проектное решение, гарантируйте, что ЕДИНЫЙ ЯЗЫК господствует над любыми техническими предпочтениями. В рамках подхода DDD главной является модель деловой предметной области.

Конструирование

Когда мы вновь создаем СУЩНОСТЬ, мы хотим использовать конструктор, который получает достаточно информации, чтобы полностью идентифицировать сущность и позволить клиентам найти ее. Когда используется раннее генерирование идентификаторов, правильно разработанный конструктор получает в качестве параметра по крайней мере уникальный идентификатор. Если бы СУЩНОСТЬ запрашивалась другими средствами, например по имени или описанию, мы также включали бы эту информацию в список параметров конструктора.

Иногда СУЩНОСТЬ имеет один или несколько инвариантов. Инвариант — это состояние, которое должно сохранять транзакционную согласованность на протяжении всего времени существования СУЩНОСТИ. Инварианты входят в компетенцию АГРЕГАТОВ, но, поскольку КОРЕНЬ АГРЕГАТА всегда является СУЩНОСТЬЮ, мы обсуждаем эту тему в данном разделе. Если СУЩНОСТЬ имеет инвариант, который удовлетворяется ненулевым состоянием содержащегося объекта или вычисляется по другому состоянию, это состояние должно обеспечиваться одним или несколькими параметрами конструктора.

Каждый объект класса `User` должен содержать переменные `tenantId`, `username`, `password` и `person`. Иначе говоря, для успешного создания объекта необходимо, чтобы ссылки на эти объявленные переменные экземпляра никогда не равнялись значению `null`. Конструктор класса `User` и его методы-установщики переменных экземпляра этому условию удовлетворяют.

```
public class User extends Entity {
   ...
   protected User(TenantId aTenantId, String aUsername,
          String aPassword, Person aPerson) {
       this();
       this.setPassword(aPassword);
       this.setPerson(aPerson);
       this.setTenantId(aTenantId);
       this.setUsername(aUsername);
       this.initialize();
   }
   ...
```

```
    protected void setPassword(String aPassword) {
        if (aPassword == null) {
            throw new IllegalArgumentException(
                  "The password may not be set to null.");
        }
        this.password = aPassword;
}

   protected void setPerson(Person aPerson) {
      if (aPerson == null) {
          throw new IllegalArgumentException(
                "The person may not be set to null.");
      }
      this.person = aPerson;
   }

   protected void setTenantId(TenantId aTenantId) {
      if (aTenantId == null) {
          throw new IllegalArgumentException(
                "The tenantId may not be set to null.");
      }
      this.tenantId = aTenantId;
   }

   protected void setUsername(String aUsername) {
      if (this.username != null) {
          throw new IllegalStateException(
                "The username may not be changed.");
      }
      if (aUsername == null) {
          throw new IllegalArgumentException(
                "The username may not be set to null.");
      }
          this.username = aUsername;
   }
   ...
}
```

Подход к проектированию класса `User` демонстрирует мощь самоинкапсуляции. Конструктор делегирует присваивание переменной экземпляра собственным внутренним методам-установщикам атрибутов/свойств, обеспечивающим самоинкапсуляцию для этих переменных. Самоинкапсуляция позволяет каждому методу-установщику самостоятельно определять подходящие контрактные условия для установки части состояния. Каждый из методов-установщиков индивидуально устанавливает ненулевое ограничение от имени СУЩНОСТИ, регламентирующей контракт экземпляра. Эти утверждения называются *предохранителями* (guards) (см. раздел "Проверка корректности"). Как указывалось ранее, в разделе "Постоянство идентификатора", самоинкапсуляция этих методов-установщиков при необходимости может быть более сложной.

Для создания сложных СУЩНОСТЕЙ следует использовать ФАБРИКУ. Это понятие рассмотрено в главе, посвященной **ФАБРИКАМ (11)**. Заметили ли вы в предыдущем примере, что конструктору класса `User` назначен атрибут `protected`? Сущность класса `Tenant` служит ФАБРИКОЙ экземпляров класса `User`, и только классы, находящиеся в том же МОДУЛЕ, могут иметь доступ к конструктору класса `User`. В таком случае ни один объект, кроме объектов класса `Tenant`, не имеет права создавать экземпляры класса `User`.

```
public class Tenant extends Entity {
   ...
   public User registerUser(
          String aUsername,
          String aPassword,
          Person aPerson) {

      aPerson.setTenantId(this.tenantId());

      User user =
             new User(
                    this.tenantId(),
                    aUsername,
                    aPassword,
                    aPerson);
       return user;
     }
     ...
}
```

Здесь метод `registerUser()` является ФАБРИКОЙ. Эта ФАБРИКА упрощает создание состояния, принятого по умолчанию для объекта класса `User`, и гарантирует, что объект класса `TenantId` для СУЩНОСТЕЙ классов `User` и `Person` всегда является корректным. Все это происходит под контролем метода ФАБРИКИ, который включен в ЕДИНЫЙ ЯЗЫК.

Проверка корректности

Основной причиной, по которой производится проверка корректности модели, является желание проверить правильность всех атрибутов и свойств, целостных объектов или композиций объектов. Рассмотрим три уровня проверки корректности модели. Несмотря на многочисленность способов проверки корректности, включая специализированные каркасы и библиотеки, мы не будем рассматривать их в нашей книге. Вместо этого мы опишем универсальные подходы, которые, впрочем, могут привести к более сложным методам.

Проверка корректности означает разные вещи. Просто из того, что все атрибуты и свойства объекта предметной области корректны по отдельности, не

следует, что объект в целом является корректным. Комбинация двух корректных атрибутов может сделать некорректным целый объект. Аналогично из того, что каждый отдельный объект в целом является корректным, не следует, что композиция объектов является корректной. Комбинация двух СУЩНОСТЕЙ, каждая из которых находится в корректном состоянии, может сделать некорректным всю композицию. Следовательно, чтобы решить все возможные проблемы, нам может потребоваться несколько уровней проверки корректности.

Проверка корректности атрибутов и свойств

Как защитить от установки некорректного значения отдельный атрибут или свойство? Разница между ними описана в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (VALUE OBJECTS) (6)**? Я настоятельно рекомендую самоинкапсуляцию. Она позволяет выработать первое решение.

Процитируем Мартина Фаулера: "Самоинкапсуляция — это такая разработка классов, при которой весь доступ к данным, даже внутри самого класса, осуществляется посредством методов доступа" [Fowler, Self Encap]. Использование этого метода дает несколько преимуществ. Он допускает абстрагирование статических переменных и переменных экземпляра. Он обеспечивает легкое наследование атрибутов и свойств от любого количества объектов. И не в последнюю очередь он упрощает проверку корректности модели.

На самом деле я люблю называть использование самоинкапсуляции для защиты корректного состояния объекта *валидацией* (validation). Это название отпугивает некоторых разработчиков, потому что они считают, что валидация — это отдельная задача, которая относится к классам, а не к объектам предметной области. Я согласен. Впрочем, я говорю немного о другом. Я обсуждаю *утверждения* (assertions), которые лежат в основе *контрактного проектирования* (design-by-contract).

По определению контрактное проектирование позволяет сформулировать предусловия, постусловия и инварианты проектируемых компонентов. Этот подход разрабатывался Бертраном Мейером (Bertrand Meyer) и был последовательно реализован им в языке программирования Eiffel. Применение этого подхода в языках Java и C# описано в книге *Design Patterns and Contracts* [Jezequel et al.]. Здесь мы рассмотрим только валидацию, основанную на проверке предусловий и применении предохранителей.

```
public final class EmailAddress {

    private String address;

    public EmailAddress(String anAddress) {
        super();
        this.setAddress(anAddress);
```

```
    }
    ...

    private void setAddress(String anAddress) {
        if (anAddress == null) {
            throw new IllegalArgumentException(
                    "The address may not be set to null.");
        }
        if (anAddress.length() == 0) {
            throw new IllegalArgumentException(
                    "The email address is required.");
        }
        if (anAddress.length() > 100) {
            throw new IllegalArgumentException(
                     "Email address must be 100 characters or less.");
        }
        if (!java.util.regex.Pattern.matches(
            "\\w+([-+.']\\w+)*@\\w+([-.]\\w+)*\\.\\w+([-.]\\w+)*",
                  anAddress)) {
            throw new IllegalArgumentException(
                  "Email address and/or its format is invalid.");
        }

        this.address = anAddress;
    }
    ...
}
```

Контракт метода `setAddress()` содержит четыре предусловия. Все предохранители предусловия проверяют условие с аргументом `anAddress`.

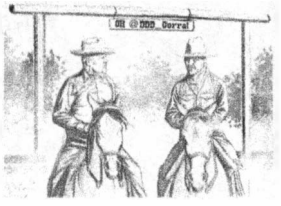
- Параметр не должен быть равным `null`.
- Параметр не должен быть пустой строкой.
- Длина параметра не должна превышать 100 символов (но не нулевых).
- Параметр должен соответствовать основному формату электронного адреса.

Если все эти предусловия выполняются, свойство `address` устанавливается равным значению аргумента `anAddress`. Если одно из них не выполняется, то генерируется исключение `IllegalArgumentException`. Класс `EmailAddress` не является СУЩНОСТЬЮ. Он является ОБЪЕКТОМ-ЗНАЧЕНИЕМ. Мы используем его по нескольким причинам. Во-первых, это хороший пример реализации предохранителей разных уровней: от проверки равенства нулю до форматирования значений (об этом чуть позже). Во-вторых, это ЗНАЧЕНИЕ хранится СУЩНОСТЬЮ класса `Person` как одно из его свойств, а именно: косвенно с помощью ЗНАЧЕНИЯ класса `ContactInformation`. Итак, на самом деле оно является частью СУЩНОСТИ точно так же, как простой атрибут, объявленный в классе СУЩНОСТИ является ее частью. При реализации методов-установщиков для простых атрибутов мы

используем точно такие же предусловия. Когда свойству СУЩНОСТИ присваивается ЦЕЛОСТНОЕ ЗНАЧЕНИЕ, не существует способа защиты от установки некорректного состояния, если внутри ЗНАЧЕНИЯ нет защиты более мелких атрибутов.

Ковбойская логика

LB: Я думал, что у меня есть хороший аргумент для моей миссис, но она внезапно создала исключительную ситуацию для меня своим нелегальным аргументом.



Некоторые разработчики называют эти виды проверок предварительного условия *безопасным программированием* (defensive programming). Разумеется, предотвращение ввода в модель абсолютно недопустимых значений представляет собой безопасное программирование. Некоторые специалисты могут не согласиться с увеличивающейся степенью специфики таких предохранителей. Некоторые программисты, придерживающиеся безопасного программирования, соглашаются с проверкой нулей и даже пустых строк, но могут уклониться от проверки таких условий, как длина строки, числовые диапазоны, форматы значения и т.п. Некоторые думают, например, что проверку размера значений лучше всего поручить базе данных. Они полагают, что проверка таких условий, как максимальная длина строки, следует поручить кому-то другому, а не объектам модели. И все же эти предварительные условия могут рассматриваться как законные проверки работоспособности.

В некоторых ситуациях проверка длины строк может оказаться ненужной. Это может произойти, например, если в базе данных максимальный размер столбца NVARCHAR никогда не достигается. Текстовые столбцы на платформе Microsoft SQL Server можно объявить с помощью ключевого слова `max`.

```
CREATE TABLE PERSON (
   ...
   CONTACT_INFORMATION_EMAIL_ADDRESS_ADDRESS
           NVARCHAR(max) NOT NULL,
   ...
) ON PRIMARY
GO
```

Вряд ли можно ожидать, что длина электронного адреса составит 1 073 741 822 символа. Просто мы хотим объявить такой размер столбца, который невозможно превзойти, чтобы больше об этом не беспокоиться.

В некоторых базах данных это невозможно. Например, в таблицах типа MyISAM базы данных MySQL установлен максимальный размер строки, равный

65 535 байт. Однако это размер *строки*, а не столбца. Если мы объявим хотя бы один столбец типа VARCHAR с максимальным размером, равным 65 535, в таблице не останется места для еще одного столбца. В зависимости от количества столбцов типа VARCHAR в данной таблице, мы должны ограничивать размер каждого столбца исходя из практических соображений, чтобы в таблице поместились все столбцы. В таких ситуациях можно объявить символьные столбцы типа TEXT, поскольку столбцы типа TEXT и BLOB хранятся в разных сегментах. Следовательно, в зависимости от базы данных существуют способы обойти ограничения, связанные с предельным размером столбца, и устранить необходимость проверки длины строки в модели.

Если существует возможность переполнения столбца, то простая проверка длины строки в модели вполне оправдана. Насколько практичным было бы транслировать следующую ошибку в осмысленную ошибку предметной области?

```
ORA-01401: inserted value too large for column
ORA-01401: введенное значение превышает размер столбца
```

Мы не можем даже определить, какой именно столбец оказался переполненным. Возможно, лучше всего было бы избежать появления этой проблемы, предварительно проверив размер текста в предусловии методов-установщиков. Кроме того, проверка размера относится не только к ограничениям столбцов в базе данных. Сама предметная область может накладывать ограничения на длину текста, исходя из вполне разумных причин, например из-за ограничений наследуемой системы, с которой мы выполняем интеграцию.

Мы должны также рассмотреть возможность выполнения проверок диапазонов и других вариантов. Даже простая проверка формата, например адреса электронной почты, является целесообразной, если мы хотим предотвратить ассоциацию с СУЩНОСТЬЮ совершенно некорректного значения. Если основные значения отдельной СУЩНОСТИ являются корректными, то грубая проверка целостных объектов и композиций объектов значительно облегчается.

Валидация целостных объектов

Даже если СУЩНОСТЬ имеет совершенно корректные атрибуты и свойства, это не значит, что сама СУЩНОСТЬ является корректной. Для валидации целостной СУЩНОСТИ необходимо иметь доступ к состоянию всего объекта, т.е. ко всем его атрибутам и свойствам. Кроме того, нам нужна **СПЕЦИФИКАЦИЯ (SPECIFICATION)** [Evans & Fowler, Spec] или **СТРАТЕГИЯ (STRATEGY)** [Гамма и др.] проверки корректности.

В своем языковом шаблоне **ПРОВЕРКИ (CHECKS)** Уорд Каннингем (Ward Cunningham) [Cunningham, Checks] исследовал несколько подходов к валидации. Для целостных проверок полезной является **ОТЛОЖЕННАЯ ВАЛИДАЦИЯ**

(DEFERRED VALIDATION). Уорд говорит, что это "класс проверки, которая должна быть отложена до последнего возможного момента". Она откладывается, потому что представляет собой очень подробную проверку, которая может затянуться при проверке сложного объекта или даже композиции объектов. По этой причине мы обсудим ОТЛОЖЕННУЮ ВАЛИДАЦИЮ позднее, когда будем рассматривать более крупные композиции объектов. В этом разделе я ограничиваю проверки тем, что Уорд назвал "проверкой простых действий".

Поскольку во время валидации мы должны иметь доступ ко всему состоянию СУЩНОСТИ, может показаться, что это подходящий повод для внедрения логики проверки непосредственно в СУЩНОСТЬ. Здесь следует быть осторожным. Во многих случаях логика валидации объекта предметной области изменяется чаще, чем сам объект. Внедрение логики проверки в саму СУЩНОСТЬ накладывает на нее слишком много обязанностей. Поддерживая свое состояние, она уже несет ответственность за поведение предметной области.

Компонент, связанный с валидацией, обязан проверять, является ли состояние СУЩНОСТИ корректным. Разрабатывая отдельный класс для валидации на языке Java, поместите его в тот же самый МОДУЛЬ (пакет), что и СУЩНОСТЬ. Если используется язык Java, следует объявить методы-получатели атрибутов и свойств в защищенной области видимости или в пакете, а еще лучше — сделать их открытыми. Закрытая область видимости не позволяет классу прочитать требуемое состояние. Если класс для валидации не поместить в тот же МОДУЛЬ, что и СУЩНОСТЬ, все методы доступа к атрибутам и свойствам придется сделать открытыми, что во многих ситуациях нежелательно.

Класс для валидации может реализовывать шаблон СПЕЦИФИКАЦИЯ или СТРАТЕГИЯ. Обнаружив некорректное состояние, он информирует клиента или как-то иначе делает запись о результатах валидации, которые можно просмотреть позже (например, после обработки пакета данных). В процессе валидации важнее собрать полный набор результатов, чем генерировать исключение при первых признаках опасности. Рассмотрим следующий абстрактный класс `validator` и его конкретные подклассы.

```
public abstract class Validator {
   private ValidationNotificationHandler notificationHandler;
   ...
   public Validator(ValidationNotificationHandler aHandler) {
       super();
       this.setNotificationHandler(aHandler);
   }

   public abstract void validate();

   protected ValidationNotificationHandler notificationHandler() {
       return this.notificationHandler;
   }
```

```
   private void setNotificationHandler(
           ValidationNotificationHandler aHandler) {
      this.notificationHandler = aHandler;
   }
}

public class WarbleValidator extends Validator {

   private Warble warble;

   public Validator(
          Warble aWarble,
          ValidationNotificationHandler aHandler) {
      super(aHandler);
      this.setWarble(aWarble);
   }
   ...
   public void validate() {
      if (this.hasWarpedWarbleCondition(this.warble())) {
          this.notificationHandler().handleError(
                   "The warble is warped.");
      }
      if (this.hasWackyWarbleState(this.warble())) {
          this.notificationHandler().handleError(
                   "The warble has a wacky state.");
      }
      ...
   }
}
```

Экземпляр класса `WarbleValidator` создается методом `ValidationNotificationHandler`. Если обнаруживается некорректное условие, метод `ValidationNotificationHandler` получает команду обработать его. Метод `ValidationNotificationHandler` представляет собой универсальную реализацию метода `handleError()`, получающего уведомление класса `String`. Вместо этого мы могли бы создать специализированные реализации, в которых для каждого вида некорректных условий предусмотрен отдельный метод.

```
class WarbleValidator extends Validator {
   ...
   public void validate() {
       if (this.hasWarpedWarbleCondition(this.warble())) {
           this.notificationHandler().handleWarpedWarble();
       }
       if (this.hasWackyWarbleState(this.warble())) {
           this.notificationHandler().handleWackyWarbleState();
       }
    }
    ...
}
```

Это позволяет не накапливать в процессе проверки сообщения об ошибках или ключах свойств, а также другие виды уведомлений. Еще лучше поместить обработку уведомлений внутрь метода проверки.

```
class WarbleValidator extends Validator {
   ...
   public Validator(
          Warble aWarble,
          ValidationNotificationHandler aHandler) {
      super(aHandler);
      this.setWarble(aWarble);
   }
   ...
   public void validate() {
      this.checkForWarpedWarbleCondition();
      this.checkForWackyWarbleState();
      ...
   }
   ...
   protected checkForWarpedWarbleCondition() {
      if (this.warble()...) {
          this.warbleNotificationHandler().handleWarpedWarble();
      }
   }
   ...
   protected WarbleValidationNotificationHandler
        warbleNotificationHandler() {
        return (WarbleValidationNotificationHandler)
            this.notificationHandler();
   }
}
```

В этом примере мы используем метод `ValidationNotificationHandler` для условия типа `Warble`. Этот тип вначале является стандартным, а потом приводится к конкретному типу, который используется внутри класса, Для поставки корректного типа модель должна содержать контракт между ней и клиентами.

Как клиенты могут гарантировать выполнение валидации СУЩНОСТИ? В каком месте начинается процесс проверки?

Например, можно поместить метод `validate()` во все СУЩНОСТИ, требующие проверки, и вызвать его с помощью шаблона СУПЕРТИП УРОВНЯ (LAYER SUPERTYPE).

```
public abstract class Entity
       extends IdentifiedDomainObject {
```

```
    public Entity() {
       super();
    }
```

```
    public void validate(
       ValidationNotificationHandler aHandler) {
    }
}
```

Любой подкласс `Entity` может совершенно безопасно вызывать свой метод `validate()`. Если конкретная СУЩНОСТЬ поддерживает специализированную проверку, она выполняется. Если она ее не поддерживает, то поведение становится холостым. Если проверку выполняют только некоторые СУЩНОСТИ, то, возможно, лучше объявить метод `validate()` только там, где это необходимо.

Однако должны ли СУЩНОСТИ проверять себя сами? То, что они имеют собственный метод `validate()`, еще не значит, что СУЩНОСТЬ сама выполняет валидацию. Да, этот факт позволяет СУЩНОСТИ определять, *что* именно будет ее проверять, освобождая клиентов от необходимости самим делать этот выбор.

```
public class Warble extends Entity {
   ...
   @Override
   public void validate(ValidationNotificationHandler aHandler) {
       (new WarbleValidator(this, aHandler)).validate();
   }
   ...
}
```

При необходимости каждый специализированный подкласс `Validator` может выполнять любое количество детальных проверок. СУЩНОСТЬ ничего не обязана знать о том, *как* выполняется проверка. Она знает только, что она может быть проверена. Кроме того, отдельный класс `Validator` позволяет изменять темп процесса проверки разных СУЩНОСТЕЙ, а также выполнять сложные проверки.

Валидация композиций объектов

Мы можем использовать ОТЛОЖЕННУЮ ВАЛИДАЦИЮ для того, что Уорд Каннингем называет "более сложными действиями, требующими проверки всех более простых действий и многого другого". В этом случае мы определяем, является ли корректной не только отдельная СУЩНОСТЬ, но и кластер или композиция СУЩНОСТЕЙ в совокупности, включая один или несколько экземпляров

АГРЕГАТОВ. Для этого могли бы создать подходящее количество экземпляров конкретного подкласса Validator. Однако, возможно, лучше всего управлять валидацией этого вида с помощью СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, которая может использовать ХРАНИЛИЩЕ для чтения экземпляров АГРЕГАТОВ, требующих проверки. Она могла бы выполнить проверку каждого экземпляра в требуемом темпе, как по отдельности, так и в совокупности.

Следует решить, всегда ли необходима проверка. Иногда АГРЕГАТ или множество АГРЕГАТОВ находятся во временном, промежуточном состоянии. Возможно, в этом случае мы могли бы моделировать состояние АГРЕГАТА, чтобы предотвратить ненужную проверку. Если условия не подходят для проверки, модель могла бы информировать клиентов об этом, публикуя СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ.

```
public class SomeApplicationService ... {
   ...
   public void doWarbleUseCaseTask(...) {
      Warble warble =
          this.warbleRepository.warbleOfId(aWarbleId);

      DomainEventPublisher
          .instance()
          .subscribe(new DomainEventSubscriber<WarbleTransitioned>(){
              public void handleEvent(DomainEvent aDomainEvent) {
                 ValidationNotificationHandler handler = ...;
                 warble.validate(handler);
                 ...
              }
              public Class<WarbleTransitioned>
                     subscribedToEventType() {
                 return WarbleTransitioned.class;
              }
          });

      warble.performSomeMajorTransitioningBehavior();
   }
}
```

Когда клиент получает сообщение WarbleTransitioned, он понимает, что условия вновь стали приемлемыми. До этого момента клиенты воздерживаются от проверки.

Наблюдение за изменениями

По определению СУЩНОСТИ в слежении за изменениями его состояния на протяжении всего времени существования нет необходимости. Мы должны

поддерживать только непрерывно изменяющееся состояние. Однако иногда эксперты в предметной области интересуются важными событиями, происходящими в модели с течением времени. В этом случае может помочь механизм слежения за конкретными изменениями СУЩНОСТЕЙ.

Практичнее всего реализовать точное и полезное слежение за изменениями с помощью СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ и ХРАНИЛИЩА СОБЫТИЙ. Для этого мы создаем уникальный тип СУЩНОСТИ для каждой важной команды, изменяющей состояние и примененной к каждому АГРЕГАТУ, который интересует экспертов в предметной области. Сочетание имени СОБЫТИЯ и его свойств делает явной запись об изменении. СОБЫТИЯ публикуются после завершения выполнения командных методов. Подписчик регистрирует получение каждого события, созданного моделью. Получив сообщение, подписчик сохраняет его в ХРАНИЛИЩЕ СОБЫТИЙ.

Эксперты в предметной области могут не интересоваться всеми изменениями. происходящими в модели, но техническую группу это должно интересовать в любом случае. Обычно это происходит по техническим причинам и сопровождается применением шаблона **ИСТОЧНИКИ СОБЫТИЙ (EVENT SOURCING) (4)**.

Резюме

Мы рассмотрели целый ряд тем, связанных с СУЩНОСТЯМИ. Напомним, что мы изучили в этой главе.

- Рассмотрели четыре основных способа генерирования уникальных идентификаторов СУЩНОСТЕЙ.
- Осознали важность правильного выбора момента для генерирования идентификатора и научились использовать суррогатные идентификаторы.
- Теперь мы знаем, как обеспечить стабильность идентификаторов.
- Узнали, как выявить внутренние характеристики СУЩНОСТЕЙ с помощью ЕДИНОГО ЯЗЫКА в КОНТЕКСТЕ. Мы показали также, как выявить свойства и функции.

- Вместе с основным поведением мы рассмотрели сильные и слабые стороны моделирования СУЩНОСТЕЙ с помощью многочисленных ролей.
- В заключение мы изучили детали создания СУЩНОСТЕЙ, проверки их корректности и слежения за изменениями при необходимости.

Далее мы рассмотрим очень важный компонент тактического моделирования — ОБЪЕКТЫ-ЗНАЧЕНИЯ.

Глава 6

Объекты-значения

Цена — это то, что вы платите.
Стоимость — это то, что вы получаете.
Уоррен Баффет

Несмотря на то что это понятие часто оказывается в тени СУЩНОСТЕЙ, **ОБЪЕКТЫ-ЗНАЧЕНИЕ (VALUE OBJECTS)** представляют собой жизненно важный компонент предметно-ориентированного программирования. Примерами объектов, которые обычно моделируются как ЗНАЧЕНИЯ, являются числа, такие как 3, 10 и 293,51; текстовые строки, такие как "Привет, мир!" и "Предметно-ориентированное проектирование"; даты; времена; объекты, содержащие персональные данные, такие как полное имя человека, состоящее из имени, отчества, фамилии и атрибутов звания; и другие, такие как валюта, цвета, телефонные номера и почтовые адреса. Есть и более сложные виды. Я буду обсуждать ЗНАЧЕНИЯ, моделирующие концепции вашей предметной области, используя **ЕДИНЫЙ ЯЗЫК (1)** и преследуя цели предметно-ориентированного проектирования.

Преимущества значений

Типы значений, представляющие собой результаты измерений, количественной оценки или описания предметов, проще создавать, тестировать, использовать, оптимизировать и поддерживать.

Возможно, вы удивитесь, но при моделировании по возможности следует использовать ОБЪЕКТЫ-ЗНАЧЕНИЯ, а не СУЩНОСТИ. Даже если понятие предметной области должно моделироваться как СУЩНОСТЬ, следует стремиться проектировать эту СУЩНОСТЬ так, чтобы она была больше похожа на контейнер ЗНАЧЕНИЙ, а не на дочерний контейнер СУЩНОСТЕЙ. Этот совет — не прихоть и не дело вкуса. Типы значений, представляющие собой результаты измерений, количественной оценки или описания, проще создавать, тестировать, использовать, оптимизировать и поддерживать.

Назначение главы

- Научить выявлять характеристики понятий предметной области для моделирования в виде ЗНАЧЕНИЙ.
- Показать, как ОБЪЕКТЫ-ЗНАЧЕНИЯ минимизируют сложность интеграции.
- Продемонстрировать использование СТАНДАРТНЫХ ТИПОВ предметной области, выраженных как ЗНАЧЕНИЯ.
- Рассказать, как компания SaaSOvation осознала важность ЗНАЧЕНИЙ.
- Описать, как компания SaaSOvation тестировала, реализовывала и сохраняла свои типы ЗНАЧЕНИЙ.

Сначала команды SaaSOvation увлеклись использованием СУЩНОСТЕЙ. Фактически это стало происходить задолго до того, как понятия ПОЛЬЗОВАТЕЛЯ и РАЗРЕШЕНИЯ были связаны друг с другом с помощью сотрудничества. С самого начала проекта они следовали популярному стереотипу, согласно которому каждый элемент их модели предметной области следовало отобразить в собственную таблицу базы данных, и что все их атрибуты должны быть легко заданы и получены с помощью открытых методов доступа. Поскольку у каждого объекта был первичный ключ базы данных, модель была плотно скомпонована в виде большого, сложного графа. Это объяснялось желанием моделировать данные, которое возникает у большинства разработчиков, чрезмерно увлеченных реляционными базами данных, где все нормализовано и имеет ссылки в виде внешних ключей. Как позже узнали разработчики, не было никаких оснований погружаться в паутину СУЩНОСТЕЙ. Это было не только не нужным, но и слишком дорогостоящим с точки зрения времени разработки и прилагаемых усилий.

При правильном проектировании экземпляр ЗНАЧЕНИЯ может быть создан, передан и забыт. Мы не должны волноваться, что потребитель как-то неправильно его изменил или даже вообще модифицировал до неузнаваемости. У ЗНАЧЕНИЯ может быть как короткое, так и долгое время жизни. Это обычное целое и невредимое ЗНАЧЕНИЕ, которое появляется и исчезает по мере необходимости.

Использование ЗНАЧЕНИЙ позволяет значительно разгрузить мозг. Оно напоминает переход от языка программирования без средств управляемой памяти к языку с механизмом сбора мусора. Простота использования, которую предоставляют ЗНАЧЕНИЯ, предоставляет такое количество их видов, которое мы можем обосновать.

Итак, как понять, следует ли моделировать понятие предметной области как ЗНАЧЕНИЕ? Для этого необходимо обратить внимание на его характеристики.

> Если элемент модели полностью определяется своими атрибутами, то его следует считать ОБЪЕКТОМ-ЗНАЧЕНИЕМ. Сделайте так, чтобы он отражал смысл заложенных в него атрибутов, и придайте ему соответствующую функциональность. Считайте такой объект неизменяющимся. Не давайте ему индивидуальности, вообще избегайте любых сложностей, неизбежных при программном управлении СУЩНОСТЯМИ [Эванс, с. 103].

Кажущаяся простота создания ЗНАЧЕНИЯ иногда приводит людей, плохо разбирающихся в предметно-ориентированном проектировании, к неправильному решению при выборе предмета моделирования — СУЩНОСТИ или ЗНАЧЕНИЯ — в конкретном экземпляре. Иногда с этим соблазном борются даже опытные разработчики. Показывая, как реализовать ЗНАЧЕНИЕ, я надеюсь снять покров тайны с этого иногда запутанного процесса принятия решений.

Характеристики значений

При моделировании понятия предметной области в виде ОБЪЕКТА-ЗНАЧЕНИЯ в первую очередь следует убедиться, что вы используете ЕДИНЫЙ ЯЗЫК. Считайте это базовым принципом и основной характеристикой. Я буду следовать этому принципу на протяжении всей главы.

Решая, является ли понятие ЗНАЧЕНИЕМ, следует выяснить, обладает ли оно большинством из следующих характеристик.

- Оно измеряет, оценивает или описывает объект предметной области.
- Его можно считать неизменяемым.
- Оно моделирует нечто концептуально целостное, объединяя связанные атрибуты в одно целое.
- При изменении способа измерения или описания его можно полностью заменить.
- Его можно сравнивать с другими объектами с помощью отношения равенства ЗНАЧЕНИЙ.
- Оно предоставляет связанным с ним объектам ФУНКЦИЮ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ [Эванс].

Это поможет глубже понять перечисленные характеристики. Применяя этот подход для анализа элементов проекта в модели, вы можете обнаружить, что ОБЪЕКТЫ-ЗНАЧЕНИЯ следует использовать чаще, чем раньше.

Измерение, количественная оценка или описание

Истинный ОБЪЕКТ-ЗНАЧЕНИЕ в модели — это не элемент предметной области. На самом деле это понятие, которое *измеряет*, *определяет количество* или как-то иначе *описывает* элемент предметной области. У человека есть возраст. Возраст — это не элемент, а понятие, которое измеряет или определяет количество лет, которое человек (элемент) прожил. У человека есть имя. Имя — не элемент, а понятие, которое описывает то, как называется человек (объект).

Все это тесно связано с характеристикой КОНЦЕПТУАЛЬНОГО ЦЕЛОСТНОГО (CONCEPTUAL WHOLE).

Неизменяемость

Объект, являющийся ЗНАЧЕНИЕМ, после создания остается неизменным.[1] Например, в языках Java и C# для создания экземпляра используются конструкторы класса ЗНАЧЕНИЯ, которым в качестве параметров передаются все объекты, от состояния которых оно зависит. Этими параметрами могут быть объекты, являющиеся непосредственными атрибутами ЗНАЧЕНИЯ, или объекты, из которых во время создания выводится один или несколько атрибутов. Рассмотрим пример типа ОБЪЕКТ-ЗНАЧЕНИЕ, в котором хранится ссылка на другой ОБЪЕКТ-ЗНАЧЕНИЕ.

```
package com.saasovation.agilepm.domain.model.product;

public final class BusinessPriority implements Serializable {
    private BusinessPriorityRatings ratings;

    public BusinessPriority(BusinessPriorityRatings aRatings) {
        super();
        this.setRatings(aRatings);
        this.initialize();
    }
    ...
}
```

Само по себе создание экземпляра не гарантирует неизменности объекта. После создания и инициализации объекта ни один из его методов (открытых или скрытых) с этого момента не сможет изменить его состояние. В этом примере только методы `setRatings()` и `initialize()` могут изменять состояние, потому что они используются только в области видимости конструктора. Метод

[1] Иногда ОБЪЕКТ-ЗНАЧЕНИЕ можно проектировать изменяемым, но это требуется редко. Здесь я не буду рассматривать изменяемые ЗНАЧЕНИЯ. Если вас интересует, когда следует использовать изменяемые ЗНАЧЕНИЯ, обратитесь к книге Эванса.

setRatings() является закрытым/скрытым и не может вызываться извне экземпляра.[2] Далее, класс BusinessPriority должен быть реализован так, чтобы ни один из его методов, кроме конструкторов, открытых или скрытых, не мог вызывать метод-установщик. Позднее я объясню, как проверить неизменяемость ОБЪЕКТОВ-ЗНАЧЕНИЙ.

В зависимости от ваших вкусов иногда вы можете проектировать ОБЪЕКТЫ-ЗНАЧЕНИЯ, содержащие ссылки на СУЩНОСТИ. Однако здесь следует быть осторожным. Если СУЩНОСТЬ, на которую указывает ссылка, изменяет свое состояние (с помощью функции СУЩНОСТИ), то ЗНАЧЕНИЕ тоже изменяется, что нарушает свойство его неизменяемости. Таким образом, следует помнить, что ссылки на СУЩНОСТЬ, хранящиеся в типах ЗНАЧЕНИЙ, используются для обеспечения неизменяемости композиции, ее выразительности и удобства использования. В противном случае, если СУЩНОСТИ допускают изменение своих состояний посредством интерфейса ОБЪЕКТА-ЗНАЧЕНИЯ, то, вероятно, их не следовало объединять в одно целое. Анализируя ФУНКЦИЮ БЕЗ ПРОБОЧНЫХ ЭФФЕКТОВ, которая рассматривается ниже, следует тщательно взвешивать все "за" и "против".

> **Проверяйте свои предположения**
>
> Если вы считаете, что проектируемый объект должен изменяться с помощью его функции, спросите себя, зачем это нужно. Не следовало ли вместо этого использовать замену, а не изменение ЗНАЧЕНИЯ? Последовательное применение этого подхода может упростить проектирование.
>
> Иногда объект нецелесообразно делать неизменяемым. Это очень хорошо и свидетельствует о том, что объект следует моделировать как СУЩНОСТЬ. Если в ходе анализа вы пришли к этому заключению, обратитесь к **СУЩНОСТЯМ (5)**.

Концептуальное целое

ОБЪЕКТ-ЗНАЧЕНИЕ может иметь один, несколько или много индивидуальных атрибутов, каждый из которых связан с остальными. Каждый атрибут вносит свой важный вклад в целое, которое эти атрибуты описывают в совокупности. Взятые по отдельности каждый из атрибутов не обеспечивает согласованного описания. Только все атрибуты вместе взятые образуют полную меру или описание. Этим они отличаются от простой группировки атрибутов внутри объекта. Если целое неадекватно описывает сущность в модели, то сама по себе группировка значит мало.

---

[2] В некоторых ситуациях каркасы, такие как объектно-реляционные механизмы отображения или библиотеки сериализации (для форматов XML, JSON и т.д.), вынуждены использовать методы-установщики для изменения состояния ЗНАЧЕНИЯ из его сериализованной формы.

Как продемонстрировал Уорд Каннингем (Ward Cunningham) в своем шаблоне **ЦЕЛОЕ ЗНАЧЕНИЕ (WHOLE VALUE)**[3] [Cunningham, Whole Value aka Value Object], ЗНАЧЕНИЕ {50 000 000 долларов} имеет два атрибута: атрибут 50 000 000 и атрибут доллары. По отдельности эти атрибуты либо описывают что-то другое, либо ничего конкретного не значат. Особенно это относится к числу 50 000 000 и в некоторой степени к долларам. Вместе эти атрибуты образуют концептуальное целое, которое описывает денежную сумму. Итак, *не* следует ожидать, что вещь, которая стоит 50 000 000 долларов, должна иметь два отдельных атрибута, описывающие ее стоимость: `amount`, равный 50 000 000, и `currency`, представляющий собой доллары. Это объясняется тем, что вещи не стоят просто 50 000 000, а цена измеряется не просто в долларах. Рассмотрим неявный способ моделирования этой ситуации.

```
// некорректно моделируемая стоимость вещей
public class ThingOfWorth {
    private String name;          // атрибут
    private BigDecimal amount;    // атрибут
    private String currency;      // атрибут
    // ...
}
```

В этом примере модель и ее клиенты должны знать, когда и как использовать атрибуты `amount` и `currency` вместе, потому что они не образуют концептуальное целое. Для решения этой задачи лучше использовать другой подход.

Для правильного описания стоимости вещей она должна рассматриваться не как два отдельных атрибута, а как целое значение: {50 000 000 долларов}. Покажем, как стоимость моделируется в виде ЦЕЛОСТНОГО ЗНАЧЕНИЯ.

```
public final class MonetaryValue implements Serializable {
    private BigDecimal amount;
    private String currency;

    public MonetaryValue(BigDecimal anAmount, String aCurrency) {
        this.setAmount(anAmount);
        this.setCurrency(aCurrency);
    }
    ...
}
```

Это не значит, что класс `MonetaryValue` идеален и не может быть улучшен. Конечно, можно использовать дополнительный тип ЗНАЧЕНИЯ, например `Currency`. Мы могли бы заменить тип `String` атрибута `currency` более

[3] Он называется также ОСМЫСЛЕННОЕ ЦЕЛОЕ (MEANINGFUL WHOLE).

информативным типом Currency. Кроме того, для решения этой задачи можно применить шаблон **ФАБРИКА** или, возможно, **СТРОИТЕЛЬ (BUILDER)** [Гамма и др.]. Однако эти вопросы отвлекли бы нас от главной цели — концепции ЦЕЛОСТНОГО ЗНАЧЕНИЯ.

Поскольку целостность концепции в предметной области настолько важна, ссылка родительского объекта на ОБЪЕКТ-ЗНАЧЕНИЕ — это не просто атрибут. На самом деле это *свойство* объемлющего родительского объекта/вещи в модели, которая содержит ссылку на него. Несмотря на то, что тип ОБЪЕКТА-ЗНАЧЕНИЯ имеет один или несколько атрибутов (например, в классе MonetaryValue два атрибута), по отношению к вещи, которая содержит ссылку на экземпляр ОБЪЕКТА-ЗНАЧЕНИЯ, атрибут является свойством. Следовательно, вещь, которая стоит 50 000 000 долларов (назовем ее ThingOfWorth), должна иметь свойство (возможно, с именем worth), которое содержит ссылку на экземпляр объекта-значения, имеющего два атрибута, в совокупности описывающих меру {50 000 000 долларов}. Однако напомним, что имя свойства (возможно, worth) и имя типа ЗНАЧЕНИЯ (возможно, MonetaryValue) можно определить только после определения **ОГРАНИЧЕННОГО КОНТЕКСТА (2)** и ЕДИНОГО ЯЗЫКА. Рассмотрим улучшенную реализацию класса.

```
// правильно моделируемая стоимость вещей
public class ThingOfWorth {
    private ThingName name;        // свойство
    private MonetaryValue worth;   // свойство
    // ...
}
```

Как и ожидалось, я изменил класс ThingOfWorth так, чтобы он обладал свойством типа MonetaryValue с именем worth. Тем самым мы улучшили структуру дезорганизованных атрибутов. Что еще более важно, теперь ЗНАЧЕНИЕ выражает нечто целое.

Я бы хотел обратить внимание на второе изменение, которое, возможно, выглядит неожиданным. Имя экземпляра класса ThingOfWorth может оказаться таким же важным, как и имя worth. По этой причине я заменил тип String переменной name типом ThingName. Использование атрибута String для переменной name на первый взгляд может показаться обоснованным. Однако на последующих итерациях вы узнаете, что использование простого типа String вызывает проблемы. Он допускает утечку информации о переменной name в классе ThingOfWorth за пределы модели. Она утекает в другие части модели и в код клиента.

```
// Клиенты сталкиваются с проблемами, связанными с именами

String name = thingOfWorth.name();

String capitalizedName =
    name.substring(0, 1).toUpperCase()
     + name.substring(1).toLowerCase();
```

Здесь клиент предпринимает слабую попытку исправить имя, переведя его в верхний регистр. Определив тип ThingName, мы можем централизовать все вопросы, касающиеся переменной name в классе ThingOfWorth. Из этого примера следует, что класс ThingName может полностью форматировать текстовое имя при создании экземпляра, освобождая клиентов от этой обязанности. Это подчеркивает необходимость широкого использования ЗНАЧЕНИЙ в рамках всей модели, а не минимизации их влияния. Теперь вместо трех малозначимых атрибутов класс ThingOfWorth содержит два свойства, представляющих собой ЗНАЧЕНИЯ с правильными типами и именами.

Конструкторы класса ЗНАЧЕНИЯ влияют на эффективность концептуального целого. Наряду с неизменяемостью мы требуем, чтобы конструкторы класса ЗНАЧЕНИЯ гарантировали, что ЦЕЛОСТНОЕ ЗНАЧЕНИЕ создается с помощью одной операции. Следует запретить заполнение атрибутов экземпляра ЗНАЧЕНИЯ после его создания, чтобы ситуация не выглядела так, будто ЦЕЛОСТНОЕ ЗНАЧЕНИЕ создается шаг за шагом. Вместо этого финальное состояние должно инициализироваться моментально, т.е. атомарно. Эту особенность выражают описанные выше конструкторы классов BusinessPriority и MonetaryValue.

Рассмотрим другой аспект злоупотребления базовым типом ЗНАЧЕНИЙ (например, String, Integer или Double). Существуют языки программирования (такие, как Ruby), которые позволяют эффективно дополнять класс новыми, специализированными функциями. Благодаря таким возможностям можно рассмотреть представление валюты, например, в виде значений двойной точности с плавающей точкой. Если необходимо вычислить обменный курс валюты, можно было бы просто дополнить класс Double функцией convertToCurrency(Currency aCurrency). Это похоже на программистский трюк, но на самом ли деле идея использовать возможности языка в этом случае является удачной идеей? С одной стороны, эта специфичная для валюты функция, вероятно, затеряна в море универсальных функций для работы с числами с плавающей точкой. Это раз. Аналогично в классе Double нет никаких встроенных функций для работы с валютой. Таким образом, вы должны были бы создать в языке тип значений по умолчанию, чтобы понять больше о валютах. В конце концов вы должны перейти в класс Currency, чтобы узнать, что и во что она преобразовывает. Это два. Самое главное — класс Double не говорит ничего явного о вашей предметной области. Вы теряете след своих проблем предметной области, не применяя ЕДИНЫЙ ЯЗЫК. Это три.

Проверяйте свои предположения

Если вы испытываете соблазн включить в СУЩНОСТЬ несколько атрибутов, что в итоге свидетельствует об их ослабленных отношениях со всеми остальными атрибутами, то эти атрибуты, скорее всего, будут объединены в отдельный тип ЗНАЧЕНИЙ или в несколько типов ЗНАЧЕНИЙ. Каждый из этих типов должен представлять собой концептуальное целое, отображающее связность и названное по правилам ЕДИНОГО ЯЗЫКА. Если хотя бы один атрибут ассоциируется с описательным аспектом, то весьма вероятно, что централизация всех аспектов этого понятия повысит мощность модели. Если со временем один или несколько атрибутов должны изменяться, рассмотрите возможность замены ЦЕЛОГО ЗНАЧЕНИЯ поддержкой сущности на протяжении времени его существования.

Заменяемость

Неизменяемое ЗНАЧЕНИЕ в модели должно содержать ссылку на СУЩНОСТЬ, поскольку его константное состояние описывает корректное текущее ЦЕЛОСТНОЕ ЗНАЧЕНИЕ. Если это условие не выполняется, то все ЗНАЧЕНИЕ полностью заменяется новым ЗНАЧЕНИЕМ, которое также представляет корректное текущее ЦЕЛОСТНОЕ ЗНАЧЕНИЕ.

Концепцию заменяемости легко понять в контексте работы с числами. Предположим, что у нас есть понятие `total`, которое в вашей предметной области представляет собой целое число. Если текущее значение переменной `total` равно 3 и должно стать равным 4, вы, разумеется, не можете сделать так, чтобы число 3 стало числом 4. Вместо этого вы просто присваиваете переменной `total` целое число 4.

```
int total = 3;
// затем...
total = 4;
```

Это очевидно, но позволяет подчеркнуть один аспект. В нашем примере мы просто *заменили* значение 3 переменной `total` значением 4. Это не упрощенчество, а точный смысл замены, который сохраняется, когда тип ОБЪЕКТА-ЗНАЧЕНИЯ более сложен, чем обычное целое число. Рассмотрим более сложный комплексный тип ЗНАЧЕНИЙ.

```
FullName name = new FullName("Vaughn", "Vernon");
// затем...
name = new FullName("Vaughn", "L", "Vernon");
```

Переменная `name` вначале представляет собой описательное значение, задающее имя и фамилию. Затем ЦЕЛОСТНОЕ ЗНАЧЕНИЕ *заменяется* ЦЕЛОСТНЫМ

ЗНАЧЕНИЕМ, задающим имя, инициал отчества и фамилию автора этой книги. Я не использовал здесь метод FullName для изменения состояния значения переменной name, чтобы включить в нее инициал второго имени. Это нарушило бы свойство неизменности типа ЗНАЧЕНИЙ FullName. Я просто заменил ЦЕЛОСТНОЕ ЗНАЧЕНИЕ, присвоив объекту name ссылку на совершенно другой экземпляр типа FullName. (Правда, этот пример не слишком ярко выражает идею замены, и мы вскоре приведем более удачный способ.)

Проверяйте свои предположения

Если вы решили создать СУЩНОСТЬ из-за того, что атрибуты объекта должны изменяться, проверьте предположения о корректности модели. Не следует ли применить замену объектов? Рассматривая предыдущий пример, описывающий замену, вы могли подумать, что создание нового экземпляра непрактично и снижает эффективность. Даже если объект, с которым вы работаете, является сложным и часто изменяется, замена не обязательно оказывается непрактичной или неудачной идеей. Второй пример демонстрирует ФУНКЦИЮ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ и выразительный способ замены ЦЕЛОСТНОГО ЗНАЧЕНИЯ.

Равенство значений

Когда ОБЪЕКТ-ЗНАЧЕНИЕ сравнивается с другим экземпляром, необходимо выполнить проверку равенства объектов. В системе может быть очень много ЗНАЧЕНИЙ, которые равны друг другу, но не являются одним и тем же объектом. Равенство определяется путем сравнения типов обоих объектов, а затем их атрибутов. Если типы и атрибуты совпадают, то ЗНАЧЕНИЯ считаются равными. Далее, если два или больше ЗНАЧЕНИЙ равны между собой, то свойству СУЩНОСТИ данного типа можно присвоить (используя замену) любое из равных ЗНАЧЕНИЙ, причем присваивание не изменит значение свойства.

Рассмотрим пример реализации проверки равенства ЗНАЧЕНИЙ класса FullName.

```
public boolean equals(Object anObject) {
    boolean equalObjects = false;
    if (anObject != null &&
        this.getClass() == anObject.getClass()) {
        FullName typedObject = (FullName) anObject;
        equalObjects =
            this.firstName().equals(typedObject.firstName()) &&
            this.lastName().equals(typedObject.lastName());
    }
    return equalObjects;
}
```

Каждый из атрибутов двух экземпляров класса FullName сравнивается с другими (в этом примере используются только имя и фамилия, без второго имени). Если все атрибуты обоих объектов равны, то эти два экземпляра класса FullName считаются равными. Это конкретное ЗНАЧЕНИЕ во время создания присваивает атрибутам firstName и lastName значение null. Таким образом, проверять значение null в функции equals() при каждом сравнении свойств не обязательно. Кроме того, я предпочитаю самоинкапсуляцию, поэтому обращаюсь к атрибутам с помощью их методов доступа. Это позволяет выводить атрибуты, а не требовать от каждого атрибута существования явного состояния. Кроме того, для этого нужна реализация соответствующей функции hashCode() (она будет показана ниже).

Рассмотрим комбинацию характеристик ЗНАЧЕНИЯ, необходимых для поддержки уникального идентификатора **АГРЕГАТА (10)**. Например, при обращении к конкретному экземпляру АГРЕГАТА по идентификатору нам нужна функция сравнения равенства ЗНАЧЕНИЙ. Неизменность также очень важна. Уникальный идентификатор никогда не должен изменяться и это частично можно обеспечить неизменностью ЗНАЧЕНИЯ. Кроме того, можно извлечь пользу из понятия концептуального целого, поскольку идентификатор образуется по правилам ЕДИНОГО ЯЗЫКА и имеет все уникальные атрибуты в одном экземпляре. Однако в данном конкретном случае нам не нужно свойство заменяемости ОБЪЕКТА-ЗНАЧЕНИЯ, потому что уникальный идентификатор КОРНЯ АГРЕГАТА никогда не будет заменяться. Впрочем, свойство заменяемости не мешает использовать ЗНАЧЕНИЯ. Кроме того, если идентификатор требует наличия ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ, он реализуется как тип ЗНАЧЕНИЯ.

Проверяйте свои предположения

Спросите себя, следует ли идентифицировать проектируемую вами СУЩНОСТЬ уникальным образом или достаточно поддерживать возможность проверять равенство ЗНАЧЕНИЙ. Если понятие само по себе не требует наличия уникального идентификатора, его следует моделировать как ОБЪЕКТ-ЗНАЧЕНИЕ.

Функция без побочных эффектов

Метод внутри объекта можно проектировать как ФУНКЦИЮ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ [Эванс]. Функция — это операция над объектом, порождающая результат, но не модифицирующая его состояние. Если при выполнении конкретной операции не происходит модификации объекта, она называется операцией без побочных эффектов. Все методы неизменяемого ОБЪЕКТА-ЗНАЧЕНИЯ должны быть ФУНКЦИЯМИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ, потому что они не должны нарушать свойство неизменяемости. Отсутствие побочных эффектов и

неизменяемость тесно связаны друг с другом. Однако я предпочитаю рассматривать их как отдельные характеристики, чтобы подчеркнуть огромную выгоду от использования ОБЪЕКТОВ-ЗНАЧЕНИЙ. В противном случае ЗНАЧЕНИЯ могут оказаться простыми контейнерами атрибутов, потеряв бо́льшую часть преимуществ этого шаблона.

Функциональный подход

Эта характеристика обычно подразумевается в языках функционального программирования. Чисто функциональные языки не допускают никаких функций, кроме ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ, чтобы все замыкания получали и порождали только неизменяемые ОБЪЕКТЫ-ЗНАЧЕНИЯ.

Как показано в книге Мартина Фаулера (Martin Fowler) [Fowler, CQS], Бертран Мейер (Bertrand Meyer) описал ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ как методы *Запроса* в рамках его принципа РАЗДЕЛЕНИЯ КОМАНД И ЗАПРОСОВ (COMMAND-QUERY SEPARATION — **CQS**). Метод запроса запрашивает искомый объект. По определению запрос объекта не должен изменять ответ.

Рассмотрим пример использования ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ в типе `FullName` для создания нового замещающего значения объекта этого типа.

```
FullName name = new FullName("Vaughn", "Vernon");
// позднее...
name = name.withMiddleInitial("L");
```

В результате возникает такой же результат, как и в примере из раздела "Заменяемость", но при этом повышается выразительность программы. Данная ФУНКЦИЯ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ реализуется следующим образом.

```
public FullName withMiddleInitial(String aMiddleNameOrInitial) {
    if (aMiddleNameOrInitial == null) {
        throw new IllegalArgumentException(
        "Must provide a middle name or initial.");
    }

    String middle = aMiddleNameOrInitial.trim();

    if (middle.isEmpty()) {
        throw new IllegalArgumentException(
        "Must provide a middle name or initial.");
    }

    return new FullName(
           this.firstName(),
           middle.substring(0, 1).toUpperCase(),
           this.lastName());
}
```

В этом примере метод `withMiddleInitial()` не модифицирует состояние своего ЗНАЧЕНИЯ и, следовательно, не имеет побочных эффектов. Вместо этого он создает новый объект ЗНАЧЕНИЯ, состоящий из одной из частей полного имени и заданного отчества. Этот метод отражает важную бизнес-логику предметной области и не допускает утечку информации в клиентский код, что происходило в предыдущем примере.

> **Если ЗНАЧЕНИЕ ссылается на СУЩНОСТЬ**
>
> Следует ли разрешать методу ОБЪЕКТА-ЗНАЧЕНИЯ модифицировать сущность, передаваемую ему как параметр? Будет ли такой метод иметь побочный эффект, если не запретить ему изменять передаваемый объект? Легче ли тестировать такой метод? Определенно сказать невозможно. Следовательно, если методу ЗНАЧЕНИЯ передается в качестве параметра СУЩНОСТЬ, возможно, лучше всего, чтобы этот метод возвращал некий результат, который сама СУЩНОСТЬ может использовать для того, чтобы изменить саму себя по своим правилам.

Тем не менее такой подход порождает некоторые проблемы. Рассмотрим пример, в котором объект класса `Product`, который в рамках методологии Scrum представляет собой СУЩНОСТЬ, используется ОБЪЕКТОМ-ЗНАЧЕНИЕМ `BusinessPriority` для вычисления приоритета.

```
float priority = businessPriority.priorityOf(product);
```

Видите ли вы недостатки в таком подходе? Вероятно, вы придете к выводу, что с ним связаны по крайней мере несколько проблем.

- Обратите внимание на то, что мы вынудили ЗНАЧЕНИЕ не только зависеть от объекта класса `Product`, но и понимать форму этой СУЩНОСТИ. По возможности следует минимизировать зависимость и знания ЗНАЧЕНИЯ только своим типом и типами его атрибутов. Это не всегда возможно, но к этому следует стремиться.
- Люди, читающие этот код, не поймут, какие части объекта класса `Product` будут использоваться. Выражение носит неявный характер, ухудшая ясность модели. Было бы намного лучше передавать реальное или производное свойство объекта класса `Product`.
- Что еще более важно, любой метод ЗНАЧЕНИЯ, которому передается СУЩНОСТЬ в качестве параметра, не может гарантировать, что это не приведет к модификации сущности, что затрудняет тестирование данной операции. Итак, даже если ЗНАЧЕНИЕ обещает не модифицировать объекты, это нелегко доказать.

Проведя данный анализ, мы поняли, что нисколько не приблизились к решению задачи. Для того чтобы изменить ситуацию и сделать ЗНАЧЕНИЕ надежным, нам следовало бы передавать методам ЗНАЧЕНИЯ в качестве параметров только ЗНАЧЕНИЯ. В этом случае мы достигаем самого высокого уровня ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ. Это совсем нетрудно сделать.

```
float priority =
        businessPriority.priority(
              product.businessPriorityTotals());
```

В этом выражении мы просто просим объект класса `Product` вернуть экземпляр значения класса `BusinessPriorityTotals`. Можно предположить, что функция `priority()` должна вернуть объект другого типа, а не `float`. Это было бы особенно обоснованным, если бы выражение приоритета должно было быть формальной частью ЕДИНОГО ЯЗЫКА и для него стоило бы ввести специальный пользовательский тип значений. Такие решения возникают в результате *непрерывного* уточнения модели. Действительно, после определенного анализа команда SaaSOvation пришла к выводу, что СУЩНОСТЬ типа `Product` вообще не должна сама вычислять бизнес-приоритет. Эту задачу в конце концов поручили **СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ (7)** и, как будет показано в этой главе, это решение оказалось намного удачнее.

Если вы откажетесь от разработки специализированного ОБЪЕКТА-ЗНАЧЕНИЯ в пользу фундаментального типа ЗНАЧЕНИЙ, используемых в языке (элементарного или его оболочки), то можете испортить свою модель. Вы не сможете присваивать ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ из предметной области ОБЪЕКТАМ-ЗНАЧЕНИЯМ фундаментального типа. Любая специализированная функция будет отделена от ЗНАЧЕНИЯ. Даже если ваш язык программирования допускает оснащение фундаментального типа новой функцией, разве это позволит вам глубже вникнуть в предметную область?

Проверяйте свои предположения

Если вы считаете, что конкретный метод не свободен от побочных эффектов и должен изменять состояние своего экземпляра, проверьте свои предположения. Не следует ли применить замещение, а не изменение? В предыдущем примере описан очень простой подход к созданию нового ЗНАЧЕНИЯ с помощью повторного использования частей существующего ЗНАЧЕНИЯ и замены лишь конкретных изменяемых частей. В системах редко бывает, чтобы все объекты были ЗНАЧЕНИЯМИ. Некоторые объекты практически наверняка окажутся СУЩНОСТЯМИ. Внимательно сравнивайте квалификаторы характеристик ЗНАЧЕНИЙ с их аналогами у СУЩНОСТЕЙ. Этому следует уделить особое внимание.

Прочитав в справочнике [Эванс] советы, касающиеся ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ, и другую информацию о ЦЕЛОСТНЫХ ЗНАЧЕНИЯХ, проектировщики компании SaaSOvation поняли, что следует как можно чаще использовать ОБЪЕКТЫ-ЗНАЧЕНИЯ. Они пришли к выводу, что понимание характеристик ЗНАЧЕНИЯ действительно помогло им выявить более естественные типы ЗНАЧЕНИЙ в их предметной области.

Все ли являются ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ?

Вы можете заподозрить, что все вокруг являются ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ. Это все же лучше, чем считать, что все вокруг являются СУЩНОСТЯМИ. Вам следует проявлять немного осторожности в ситуациях, в которых существуют действительно простые атрибуты, не требующие специальной обработки. Возможно, это булевы типы или числовые значения, которые действительно являются самодостаточными, не требуют дополнительной функциональной поддержки и не связаны ни с какими другими атрибутами той же самой СУЩНОСТИ. Такие простые атрибуты представляют собой ОСМЫСЛЕННОЕ ЦЕЛОЕ (MEANINGFUL WHOLE). Впрочем, вы можете "по ошибке" завернуть отдельный атрибут в тип ЗНАЧЕНИЯ без специальной функциональности, и это лучше, чем никогда не использовать ЗНАЧЕНИЯ. Когда вы поймете, что погорячились, вы всегда сможете выполнить небольшой рефакторинг.

Интеграция в стиле минимализма

В каждом проекте DDD всегда существует несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Это значит, что нам необходимо найти правильные способы для их интеграции. При передаче объектов в КОНТЕКСТЕ от вышележащей модели в нижележащую по возможности следует использовать ОБЪЕКТЫ-ЗНАЧЕНИЯ. В этом случае можно достичь интеграции минимальными усилиями, т.е. минимизировать количество свойств, которыми придется управлять в нижележащей модели. Использование неизменяемых ЗНАЧЕНИЙ снижает ответственность.

Зачем нести ответственность?

Использование неизменяемых ЗНАЧЕНИЙ снижает ответственность.

Возвращаясь к примеру из главы, посвященной **ОГРАНИЧЕННЫМ КОНТЕКСТАМ (2)**, напомним, что два АГРЕГАТА в вышележащем *Контексте идентификации и доступа* влияют на нижележащий *Контекст сотрудничества* (рис. 6.1). В *Контексте идентификации и доступа* существуют два АГРЕГАТА — `User` и `Role`. *Контекст сотрудничества* интересуется, играет ли пользователь (`User`) конкретную роль (`Role`), а именно `Moderator`. *Контекст сотрудничества*

использует свой **ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (3)** для отправления запроса в **СЛУЖБУ С ОТКРЫТЫМ ПРОТОКОЛОМ (3)** в *Контексте идентификации и доступа*. Если запрос, использующий интеграцию, свидетельствует о том, что конкретный пользователь играет роль МОДЕРАТОРА, то *Контекст сотрудничества* создает репрезентативный объект, а именно — объект класса `Moderator`.

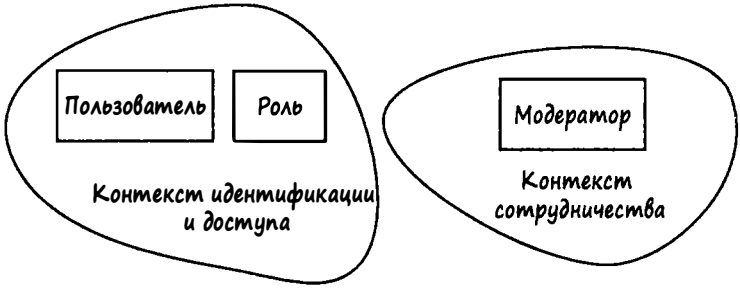


Рис. 6.1. Класс `Moderator` в своем КОНТЕКСТЕ основан на состоянии классов `User` и `Role` из другого КОНТЕКСТА. Классы `User` и `Role` являются АГРЕГАТАМИ, а класс `Moderator` — это ОБЪЕКТ-ЗНАЧЕНИЕ

Класс `Moderator` вместе с подклассами класса `Collaborator`, показанными на рис. 6.2, моделируется как ОБЪЕКТ-ЗНАЧЕНИЕ. Его экземпляры создаются статически и связываются с АГРЕГАТОМ `Forum`. Важным обстоятельством является то, что при этом минимизируется влияние многочисленных АГРЕГАТОВ, обладающих многими атрибутами в вышележащем *Контексте идентификации и доступа*, на *Контекст сотрудничества*. Имея несколько атрибутов, класс `Moderator` моделирует важную концепцию ЕДИНОГО ЯЗЫКА, относящуюся к *Контексту сотрудничества*. Более того, класс `Moderator` не получает отдельный атрибут от АГРЕГАТА `Role`. Имя класса само отражает роль МОДЕРАТОРА, которую играет пользователь. Класс `Moderator` представляет собой статически созданный экземпляр ЗНАЧЕНИЯ и не обязан сохранять синхронизацию с удаленным КОНТЕКСТОМ. Его тщательно проработанный контракт о качестве услуг позволяет разгрузить обслуживаемый КОНТЕКСТ.

Разумеется, иногда объекты в нижележащем КОНТЕКСТЕ вынуждены согласовывать часть своего состояния с одним или несколькими АГРЕГАТАМИ в удаленном КОНТЕКСТЕ. В этом случае можно спроектировать АГРЕГАТ в нижележащем обслуживаемом КОНТЕКСТЕ, потому что для поддержки потока непрерывных изменений используются СУЩНОСТИ. Однако этого следует всеми силами избегать. Если можете, для моделирования интеграции используйте ОБЪЕКТЫ-ЗНАЧЕНИЯ. Этот совет относится также ко многим ситуациям, в которых обслуживаемый контекст образуют СТАНДАРТНЫЕ ТИПЫ.

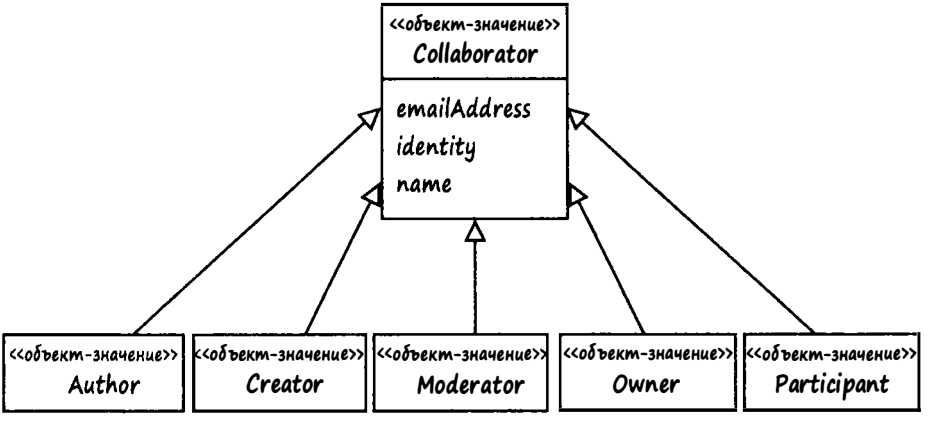


Рис. 6.2. Иерархия ОБЪЕКТОВ-ЗНАЧЕНИЙ класса `Collaborator`. Только некоторые атрибуты класса `User` поступают из вышележащего КОНТЕКСТА, а роли явно отражаются в именах классов

Стандартные типы, выраженные в виде значений

Во многих системах и приложениях возникает необходимость использования **СТАНДАРТНЫХ ТИПОВ (STANDARD TYPES)**. Стандартные типы — это описательные объекты, обозначающие типы вещей. Существуют вещь (СУЩНОСТЬ) и описание (ЗНАЧЕНИЕ), а также СТАНДАРТНЫЕ ТИПЫ, для того чтобы отличаться от других типов той же вещи. В индустрии для этой концепции нет стандартного имени, но я слышал, что их называют также *кодом типа* (type code) и *поиском* (lookup). Название *код типа* ни о чем не говорит. А если *поиск*, то поиск чего? Я выбрал название СТАНДАРТНЫЕ ТИПЫ, потому что оно является более информативным. Для разъяснения этой концепции рассмотрим несколько примеров. В некоторых случаях эти типы моделируются как **СТЕПЕННЫЕ ТИПЫ (POWER TYPES)**.

В вашем ЕДИНОМ ЯЗЫКЕ определено понятие `PhoneNumber` (ЗНАЧЕНИЕ). Оно требует уточнения типа. "Это домашний, мобильный, рабочий или какой-то другой номер телефона?", — спрашивает ваш эксперт в предметной области. Следует ли моделировать разные типы телефонных номеров в виде иерархии классов? Если каждый тип номера будет описываться отдельным классом, клиентам будет трудно их различать. В этой ситуации может помочь СТАНДАРТНЫЙ ТИП, описывающий тип телефона: `Home`, `Mobile`, `Work` или `Other`. Эти описания представляют собой СТАНДАРТНЫЕ ТИПЫ телефонов.

Как указывалось ранее, в области финансов может существовать тип ЗНАЧЕНИЯ `Currency`, ограничивающий тип `MonetaryValue` конкретной валютой.

В этом случае СТАНДАРТНЫЙ ТИП может описывать ЗНАЧЕНИЕ для каждой из валют: AUD, CAD, CNY, EUR, GBP, JPY, USD и т.д. Здесь СТАНДАРТНЫЙ ТИП позволяет избежать использования фиктивной валюты. Несмотря на то что объекту класса `MonetaryValue` можно присвоить сумму в неправильной валюте, ему невозможно присвоить сумму в несуществующей валюте. Если бы использовался атрибут в виде строки, то модель можно было бы привести в некорректное состояние. Представьте себе опечатку *doolars* и проблемы, которые она вызывает.

Теперь давайте представим, что вы работаете в фармацевтической промышленности и создаете лекарства, разработка и производство которых регулируется разными административными правилами. Конкретное лекарство (СУЩНОСТЬ) имеет продолжительный жизненный цикл и со временем изменяется, проходя путь от идеи к исследованиям, разработке, тестированию, производству, улучшению и, наконец, до снятия с производства. Вы можете управлять стадиями жизненного цикла с помощью СТАНДАРТНЫХ ТИПОВ или не делать этого. Эти стадии могут относиться к разным ОГРАНИЧЕННЫМ КОНТЕКСТАМ. С другой стороны, с точки зрения администрации больниц каждое из лекарств можно описать такими СТАНДАРТНЫМИ ТИПАМИ, как `IV`, `Oral` и `Topical`.

В зависимости от уровня стандартизации эти типы могут поддерживаться только на уровне приложения, на уровне общих корпоративных баз данных или даже национальных или международных комитетов по стандартизации. Уровень стандартизации иногда сам влияет на описание и использование СТАНДАРТНЫХ ТИПОВ в модели.

Все это можно описать в виде СУЩНОСТЕЙ, потому что каждое из них существует в своем специальном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Независимо от процесса создания и поддержки со стороны комитетов по стандартизации, следует стремиться обрабатывать их как значения в обслуживаемом КОНТЕКСТЕ. Это возможно, потому что они измеряют и описывают типы вещей, а измерения и описания лучше всего моделируются как ЗНАЧЕНИЯ. Более того, один экземпляр лекарства категории {IV}[4], например, не отличается от другого экземпляра лекарства категории {IV}. Они являются взаимозаменяемыми, что также означает, что они допускают замену и обладают свойствами ЗНАЧЕНИЯ. Таким образом, если у вас нет необходимости обеспечивать непрерывные изменения на протяжении жизненного цикла описательных типов в *вашем* ОГРАНИЧЕННОМ КОНТЕКСТЕ, то моделируйте их как ЗНАЧЕНИЯ.

Для удобства обслуживания естественно разместить СТАНДАРТНЫЕ ТИПЫ и использующие их модели в разных КОНТЕКСТАХ. В этих КОНТЕКСТАХ они являются СУЩНОСТЯМИ и обладают такими атрибутами, как `identity`, `name` и `description`. Помимо этих, они могут иметь и другие атрибуты, но

[4] IV — intravenous (внутривенное). — *Примеч. ред.*

перечисленные атрибуты являются наиболее общими в обслуживаемом КОНТЕКСТЕ. Часто используется только один из этих атрибутов. Это соответствует цели интеграции в стиле минимализма.

В качестве очень простого примера рассмотрим СТАНДАРТНЫЙ ТИП, моделирующий члена группы, который может иметь один из двух типов. Группа может состоят — из пользователей или членов, которые сами являются группами (вложенные группы). Приведенное ниже перечисление на языке Java описывает СТАНДАРТНЫЙ ТИП.

```
package com.saasovation.identityaccess.domain.model.identity;

public enum GroupMemberType {

   GROUP {
       public boolean isGroup() {
           return true;
       }
   },
   USER {
       public boolean isUser() {
           return true;
       }
   };

    public boolean isGroup() {
       return false;
    }

    public boolean isUser() {
       return false;
    }
}
```

Экземпляр ЗНАЧЕНИЯ `GroupMember` создается с конкретным типом `GroupMemberType`. Когда пользователь (`User`) или группа (`Group`) включается в группу (`Group`), присвоенный АГРЕГАТ получает команду обновить соответствующее перечисление `GroupMember`. Рассмотрим реализацию метода `toGroupMember()` в классе `User`.

```
protected GroupMember toGroupMember() {
    GroupMember groupMember =
         new GroupMember(
                 this.tenantId(),
                 this.username(),
                 GroupMemberType.USER); // стандартный тип enum
    return groupMember;
}
```

Использование перечисления из языка Java — очень простой способ поддержки СТАНДАРТНОГО ТИПА. Перечисление предоставляет точно определенное конечное количество ЗНАЧЕНИЙ (в данном случае — два), занимает очень мало места и по определению является ФУНКЦИЕЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ. А где же текстуальное описание ЗНАЧЕНИЙ? Существует два возможных ответа на этот вопрос. Часто описание типа не требуется и достаточно только имени. Почему? Текстуальные описания, как правило, являются корректными только на **УРОВНЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (USER INTERFACE LAYER) (14)** и могут обеспечиваться путем сравнения имени типа со свойством, ориентированным на представление. Во многих ситуациях свойство, ориентированное на представление, должно быть локализованным (как при многоязычном программировании), что совершенно неприемлемо для поддержки модели. Часто просто имя СТАНДАРТНОГО ТИПА является наилучшим атрибутом для использования в модели. Второй ответ заключается в том, что существует ограниченное количество описаний, встроенных в перечислениях имен GROUP и USER. Описательные имена каждого типа можно вывести на экран с помощью функции `toString()`. Однако при необходимости описательный текст каждого типа можно смоделировать.

Кроме того, это простое Java-перечисление СТАНДАРТНЫХ ТИПОВ, по существу, является элегантным и упорядоченным объектом **СОСТОЯНИЯ** [Гамма и др.]. В конце объявления этого перечисления содержатся два метода, `isGroup()` и `isUser()`, реализующих стандартное поведение всех СОСТОЯНИЙ. По умолчанию оба эти метода возвращают значение `false`, что соответствует базовому поведению. Однако в каждом из определений СОСТОЯНИЯ методы замещаются и применяются к своим конкретным СОСТОЯНИЯМ, возвращая значение `true`. Если состояние СТАНДАРТНОГО ТИПА равно GROUP, то метод `isGroup()` замещается и возвращает `true`. Если состояние СТАНДАРТНОГО ТИПА равно USER, то метод `isUser()` замещается и возвращает `true`. Состояние изменяется путем замены текущего значения перечисления другим значением.

Это перечисление демонстрирует очень простое базовое поведение. В зависимости от предметной области реализация шаблона STATE может быть намного сложнее, включая большее количество стандартных функций, которые замещаются и специализируются каждым СОСТОЯНИЕМ. Перечисление представляет собой тип ЗНАЧЕНИЯ, состояния которого ограничены точно определенным набором констант. Важным типом является перечисление `BacklogItemStatusType`, имеющее состояния PLANNED, SCHEDULED, COMMITTED, DONE и REMOVED. Я использую СТАНДАРТНЫЕ ТИПЫ во всех трех ОГРАНИЧЕННЫХ КОНТЕКСТАХ, которые приводятся как примеры, стараясь сделать их как можно более простыми.

Шаблон СОСТОЯНИЕ считается опасным?

Некоторые специалисты считают шаблон СОСТОЯНИЕ очень нежелательным. Обычно жалуются на необходимость создавать абстрактную реализацию каждой функции, поддерживаемой типом (два метода в конце перечисления `GroupMemberType`), а затем замещать эти функции, когда конкретное СОСТОЯНИЕ должно обеспечивать специализированную реализацию. В языке Java для этого обычно требуется создавать отдельный класс (как правило, в отдельном файле) для абстрактного типа и для каждого состояния. Нравится вам это или нет, но именно так устроен шаблон СОСТОЯНИЕ.

Я согласен, что обязательная разработка отдельных классов СОСТОЯНИЯ — по одному для каждого состояния и абстрактного типа — может привести к безнадежной путанице. Отдельные функции в каждом классе, смешанные с функциями из абстрактного класса, заданными по умолчанию, могут создать тесную взаимосвязь между подклассами и сделать типы менее ясными. Если СОСТОЯНИЙ много, это приводит к слишком большим потерям. Однако я полагаю, что использование Java-перечисления для описания набора СТАНДАРТНЫХ ТИПОВ не требует никаких усилий и является более эффективным, чем применение шаблона СОСТОЯНИЕ. Вы можете взять лучшее из обоих подходов. Вы получите очень простой СТАНДАРТНЫЙ ТИП и способ исследования его текущего состояния. Это обеспечит связанность функций и типа. Для практического использования функции СОСТОЯНИЯ можно ограничить.

И все же возможно, что даже такая простая реализация состояния вам не понравится.

Если вам не нравится использование Java-перечислений для поддержки СТАНДАРТНЫХ ТИПОВ, то вы всегда можете применить уникальный экземпляр ЗНАЧЕНИЯ для каждого типа. Однако если вам, в первую очередь, не нравится идея использовать шаблон СОСТОЯНИЕ, то вы можете легко и элегантно обеспечить поддержку стандартных типов с помощью перечисления, не считая его шаблоном СОСТОЯНИЕ. Помимо прочего, именно я впервые выдвинул идею использовать перечисление как СОСТОЯНИЕ. Это было альтернативой реализаций СТАНДАРТНЫХ ТИПОВ с помощью других механизмов или в виде ЗНАЧЕНИЙ.

В качестве альтернативы вы можете использовать АГРЕГАТ как СТАНДАРТНЫЙ ТИП, создав по одному АГРЕГАТУ на тип. Однако дважды подумайте, прежде чем пойти по этому пути. СТАНДАРТНЫЕ ТИПЫ обычно не должны поддерживаться внутри ОГРАНИЧЕННОГО КОНТЕКСТА, в котором они используются. Широко используемые СТАНДАРТНЫЕ ТИПЫ обычно должны поддерживаться в отдельном КОНТЕКСТЕ, а их модификации должны тщательно продумываться. Вместо этого вы можете решить открыть неизменяемые АГРЕГАТЫ СТАНДАРТНЫХ ТИПОВ в КОНТЕКСТАХ, которые их используют. При этом следует спросить

себя, соответствует ли эта неизменяемая СУЩНОСТЬ формальному определению СУЩНОСТИ. Если нет, стоит задуматься о моделировании этой сущности как общего неизменяемого ОБЪЕКТА-ЗНАЧЕНИЯ.

Общие неизменяемые ОБЪЕКТЫ-ЗНАЧЕНИЯ можно извлекать из скрытого постоянного хранилища. Это целесообразно, когда используются СТАНДАРТНЫЕ ТИПЫ **СЛУЖБА (7)** или **ФАБРИКА (11)**. В этом случае необходимо иметь поставщика отдельной СЛУЖБЫ или ФАБРИКИ для каждого набора СТАНДАРТНЫХ ТИПОВ (одну — для типов телефонных номеров, другую — для типов почтовых адресов, одну — для типов валюты), как показано на рис. 6.3. В обоих случаях конкретная реализация СЛУЖБЫ или ФАБРИКИ должна иметь доступ к постоянному хранилищу, чтобы при необходимости извлекать оттуда общие ЗНАЧЕНИЯ, но клиенты никогда не должны знать, что ЗНАЧЕНИЯ хранятся в стандартной базе данных. Использование СЛУЖБЫ или ФАБРИКИ в качестве поставщика типов открывает возможность использовать множество ценных стратегий кеширования, позволяющих работать легко и безопасно благодаря тому, что ЗНАЧЕНИЯ только считываются из базы данных и не изменяются в системе.

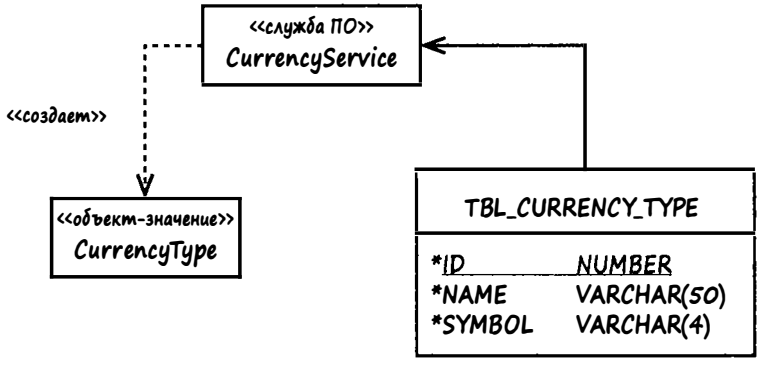


Рис. 6.3. СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ можно использовать для обеспечения СТАНДАРТНЫХ ТИПОВ. В этом случае СЛУЖБА обращается к базе данных для чтения состояния требуемого класса `CurrencyType`

В заключение, я считаю, что лучше всего отдать предпочтение реализации СТАНДАРТНЫХ ТИПОВ с помощью перечисления, независимо от того, рассматриваете ли вы его как СОСТОЯНИЕ или нет. Если в каждой категории есть много возможных экземпляров СТАНДАРТНЫХ ТИПОВ, рассмотрите возможность генерации кода для создания перечисления. Механизм генерации кода может прочитать все существующие СТАНДАРТНЫЕ ТИПЫ из соответствующего постоянного хранилища (системы записей) и создать, например, уникальный тип/состояние для каждой строки.

Если вы решите использовать в качестве СТАНДАРТНЫХ ТИПОВ классические ОБЪЕКТЫ-ЗНАЧЕНИЯ, то может оказаться полезным ввести СЛУЖБУ или ФАБРИКУ для статического создания экземпляров по мере надобности. Мотивация этого решения не отличается от предыдущей, но реализация отличается от создания общих ЗНАЧЕНИЙ. В этом случае ваша СЛУЖБА или ФАБРИКА может статически создавать неизменяемые экземпляры ЗНАЧЕНИЙ для каждого СТАНДАРТНОГО ТИПА. Любые изменения в системе записей, содержащей сущности базового СТАНДАРТНОГО ТИПА, не должны автоматически отображаться на уже существующих статически созданных экземплярах представления. Если вы хотите синхронизировать статически созданные экземпляры ЗНАЧЕНИЙ с системой записей, то должны реализовать поиск и обновление их состояния в модели. Это может негативно сказаться на эффективности такого подхода.[5] Таким образом, с самого начала проекта вы можете выяснить, что все такие статически созданные ЗНАЧЕНИЯ СТАНДАРТНОГО ТИПА никогда не будут изменяться в использующем их КОНТЕКСТЕ.

Тестирование ОБЪЕКТОВ-ЗНАЧЕНИЙ

Для того чтобы подчеркнуть важность тестирования, я сначала опишу простые тесты и только потом покажу реализацию ОБЪЕКТА-ЗНАЧЕНИЯ. Эти тесты влияют на проектирование модели предметной области, демонстрируя примеры использования клиентом каждого объекта.

Придерживаясь такого стиля, мы не интересуемся разнообразными аспектами модульного тестирования, стремясь строго доказать, что модель тщательно и всесторонне проверена. Вместо этого мы демонстрируем, как разные объекты в предметной области будут использоваться клиентами и чего клиенты могут от них ожидать. Для того чтобы правильно выяснить основные концепции модели в ходе ее проектирования, очень важно встать на точку зрения клиента. В противном случае мы будем исходить из своих представлений, а не из бизнес-логики.

Наилучший пример кода

Один из возможных стилей тестирования можно описать так: если вы разрабатываете руководство пользователя для модели, то опишите эти тесты как наиболее приемлемые образцы кода для того, чтобы клиенты понимали, как использовать конкретный объект предметной области.

[5] В такой ситуации целесообразно моделировать АГРЕГАТ в вышележащем КОНТЕКСТЕ и как АГРЕГАТ в нижележащем КОНТЕКСТЕ. При этом он не обязан быть тем же классом или содержать все те же самые атрибуты, но моделирование концепции в нижележащем КОНТЕКСТЕ в виде АГРЕГАТА обеспечит итоговую согласованность и обновление, сосредоточенное в одной точке.

Это не значит, что следует отказаться от разработки модульных тестов. Все дополнительные тесты, касающиеся принятых стандартов, создавать необходимо. Однако у каждого типа тестов есть своя мотивация. Модульные тесты и функциональные тесты занимают свое место, а тесты моделирования — свое.

ОБЪЕКТ-ЗНАЧЕНИЕ был выбран как удобное средство представления из **СМЫСЛОВОГО ЯДРА (CORE DOMAIN) (2)** *Контекста управления гибким проектированием.*

В этом ОГРАНИЧЕННОМ КОНТЕКСТЕ эксперты в предметной области говорят о "деловых приоритетах списка заданий для спринта". Для наполнения указанной части ЕДИНОГО ЯЗЫКА мы моделируем эту концепцию в виде класса `BusinessPriority`. Он обеспечивает удобные вычисления для поддержки бизнес-анализа стоимости разработки каждого из списков заданий для спринта [Wiegers]. Результатами вычисления являются процентная доля стоимости или стоимость разработки конкретного списка заданий по сравнению со стоимостью разработки всех остальных списков; общая выгода, представляющая собой общую выгоду, полученную от разработки конкретного списка заданий, и процентная выгода, равная выгоде от разработки конкретного списка заданий по сравнению с выгодой от разработки всех остальных списков, а также вычисленный приоритет, который следует присвоить этому списку по сравнению со всеми остальными.

Несмотря на то что эти тесты представлены в окончательном виде, они являются результатом выполнения многократных итераций и пошаговых уточнений в ходе рефакторинга.

```
package com.saasovation.agilepm.domain.model.product;

import com.saasovation.agilepm.domain.model.DomainTest;

import java.text.NumberFormat;

public class BusinessPriorityTest extends DomainTest {

    public BusinessPriorityTest() {
        super();
    }
    ...

    private NumberFormat oneDecimal() {
        return this.decimal(1);
    }

    private NumberFormat twoDecimals() {
        return this.decimal(2);
    }
```

```
    private NumberFormat decimal(int aNumberOfDecimals) {
        NumberFormat fmt = NumberFormat.getInstance();
        fmt.setMinimumFractionDigits(aNumberOfDecimals);
        fmt.setMaximumFractionDigits(aNumberOfDecimals);
        return fmt;
    }
}
```

Этот класс содержит несколько вспомогательных методов. Поскольку проектировщики должны проверить точность вычислений, они написали методы для создания экземпляров класса `NumberFormat` для дробных значений, которые состоят из одной или нескольких цифр после десятичной точки. Впоследствии мы еще увидим, насколько это полезно.

```
public void testCostPercentageCalculation() throws Exception {

    BusinessPriority businessPriority =
        new BusinessPriority(
               new BusinessPriorityRatings(2, 4, 1, 1));

   BusinessPriority businessPriorityCopy =
        new BusinessPriority(businessPriority);

   assertEquals(businessPriority, businessPriorityCopy);

   BusinessPriorityTotals totals =
        new BusinessPriorityTotals(53, 49, 53 + 49, 37, 33);

   float cost = businessPriority.costPercentage(totals);

   assertEquals(this.oneDecimal().format(cost), "2.7");

   assertEquals(businessPriority, businessPriorityCopy);
 }
```

Разработчикам пришла в голову хорошая идея проверить неизменяемость. Сначала каждый тест был создан как экземпляр класса `BusinessPriority`, а затем с помощью копирующего конструктора была создана его эквивалентная копия. Первое утверждение в тесте гарантировало, что этот копирующий конструктор создаст точную копию оригинала.

Затем они разработали тест для создания объекта класса `BusinessPriorityTotals` и присвоили его аргументу метода `totals`. С помощью переменной `totals` они могли использовать метод запроса `costPercentage()` и присваивать результат переменной `cost`. Затем они проверили утверждение, что возвращаемое значение равно `2.7`. Это правильное число было вычислено вручную. В заключение они проверили утверждение, что метод `costPercentage()` действительно не имеет побочных эффектов, которые могли бы возникнуть, если бы объект `businessPriority` был равен ЗНАЧЕНИЮ `businessPriorityCopy`. Благодаря этому тесту разработчики поняли, как вычислить и представить процентную стоимость.

Далее возникла необходимость протестировать вычисления приоритета, общей выгоды и процентной выгоды с помощью того же бизнес-плана.

```
public void testPriorityCalculation() throws Exception {

   BusinessPriority businessPriority =
       new BusinessPriority(
               new BusinessPriorityRatings(2, 4, 1, 1));

   BusinessPriority businessPriorityCopy =
       new BusinessPriority(businessPriority);

  assertEquals(businessPriorityCopy, businessPriority);

  BusinessPriorityTotals totals =
       new BusinessPriorityTotals(53, 49, 53 + 49, 37, 33);

  float calculatedPriority = businessPriority.priority(totals);

  assertEquals("1.03",
                this.twoDecimals().format(calculatedPriority));

   assertEquals(businessPriority, businessPriorityCopy);
}

public void testTotalValueCalculation() throws Exception {

   BusinessPriority businessPriority =
       new BusinessPriority(
               new BusinessPriorityRatings(2, 4, 1, 1));

   BusinessPriority businessPriorityCopy =
       new BusinessPriority(businessPriority);

   assertEquals(businessPriority, businessPriorityCopy);

   float totalValue = businessPriority.totalValue();

   assertEquals("6.0", this.oneDecimal().format(totalValue));

   assertEquals(businessPriority, businessPriorityCopy);
}

public void testValuePercentageCalculation() throws Exception {

   BusinessPriority businessPriority =
      new BusinessPriority(
             new BusinessPriorityRatings(2, 4, 1, 1));

   BusinessPriority businessPriorityCopy =
      new BusinessPriority(businessPriority);

   assertEquals(businessPriority, businessPriorityCopy);

   BusinessPriorityTotals totals =
      new BusinessPriorityTotals(53, 49, 53 + 49, 37, 33);
```

```
    float valuePercentage =
           businessPriority.valuePercentage(totals);

    assertEquals("5.9", this.oneDecimal().format(valuePercentage));

    assertEquals(businessPriorityCopy, businessPriority);
}
```

Тесты должны быть осмысленными с точки зрения предметной области

Тесты вашей модели должны быть понятными экспертам в предметной области.

Читая эти тесты, эксперты в предметной области, не являющиеся техническими специалистами, должны иметь возможность при небольшой помощи разработчиков понимать, как используется класс `BusinessPriority`, какие виды результатов он создает, что его функции не имеют побочных эффектов и полностью согласованы с концепциями и предназначением ЕДИНОГО ЯЗЫКА.

Следует подчеркнуть, что состояние ОБЪЕКТА-ЗНАЧЕНИЯ при любом использовании гарантированно остается неизменным. При необходимости клиенты могут вычислять приоритет для любого количества списков заданий для спринта, упорядочивать и сравнивать их, а также настраивать класс `BusinessPriorityRatings` для каждого списка.

Реализация

Мне нравится пример, основанный на классе `BusinessPriority`, потому что он демонстрирует все характеристики ЗНАЧЕНИЯ и не только. Он не только показывает, как обеспечить неизменяемость, концептуальную целостность, заменяемость, проверку равенства ЗНАЧЕНИЙ и ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ, но и демонстрирует, как использовать тип значения в качестве **СТРАТЕГИИ** [Гамма и др.].

Разработав все тестовые методы, проектировщики лучше поняли, как клиент может использовать класс `BusinessPriority`, что позволило им реализовывать его в полном соответствии с тестами. Рассмотрим определение базового класса и его конструкторов.

```
public final class BusinessPriority implements Serializable {

    private static final long serialVersionUID = 1L;

    private BusinessPriorityRatings ratings;

    public BusinessPriority(BusinessPriorityRatings aRatings) {
        super();
        this.setRatings(aRatings);

    }

    public BusinessPriority(BusinessPriority aBusinessPriority) {
         this(aBusinessPriority.ratings());
    }
```

Команда решила объявить свои типы ЗНАЧЕНИЙ как `Serializable`. Иногда экземпляр ЗНАЧЕНИЯ необходимо сериализировать, например, когда они взаимодействуют с удаленной системой и могут оказаться полезными в рамках некоторых стратегий постоянного хранения данных.

Класс `BusinessPriority` был разработан для хранения свойства ЗНАЧЕНИЯ с именем `ratings` типа `BusinessPriorityRatings` (в примере не показан). Свойство `ratings` описывает бизнес-ценность и баланс издержек, связанных с реализацией списка заданий для спринта или отказом от его реализации. Тип `BusinessPriorityRatings` обеспечивает тип `BusinessPriority` с рейтингами `benefit`, `cost`, `penalty` и `risk`, позволяющими выполнять широкий круг вычислений.

Обычно в каждом из своих ОБЪЕКТОВ-ЗНАЧЕНИЙ я предусматриваю как минимум два конструктора. Первый конструктор получает полный набор параметров, необходимых для наследования и/или задания атрибутов состояния. Этот главный конструктор инициализирует состояние по умолчанию. Инициализация основных атрибутов выполняется путем вызова закрытых методов-установщиков. Я рекомендую использовать *самоделегирование*, которое в данном примере демонстрируется с помощью закрытых методов-установщиков.

Обеспечение неизменяемости значений

Самоделегирование для задания свойств/атрибутов использует только главный конструктор. Никакие другие методы не должны выполнять самоделегирование к методам-установщикам. Поскольку все методы-установщики в ОБЪЕКТЕ-ЗНАЧЕНИИ всегда являются закрытыми, атрибуты оказываются недоступными для изменений со стороны их пользователей. Эти два фактора обеспечивают неизменяемость ЗНАЧЕНИЙ.

Второй конструктор предназначен для копирования существующего ЗНАЧЕНИЯ при создании нового экземпляра и называется *копирующим*. Этот конструктор выполняет *поверхностное копирование*, поскольку он выполняет самоделегирование к главному конструктору, передавая как параметры все соответствующие атрибуты копируемого ЗНАЧЕНИЯ. Мы могли бы выполнить *глубокое копирование*, или *клонирование*, при котором для создания нового объекта, эквивалентного существующему, копируются все атрибуты и свойства, содержащиеся в оригинале. Однако во многих ситуациях при работе со ЗНАЧЕНИЯМИ это слишком сложно и излишне. Если глубокая копия все же нужна, ее можно добавить. Впрочем, при работе с неизменяемыми ЗНАЧЕНИЯМИ всегда можно обеспечить совместное использование атрибутов/свойств несколькими экземплярами.

Второй, копирующий, конструктор играет важную роль в модульном тестировании. При тестировании ОБЪЕКТА-ЗНАЧЕНИЯ мы хотим включить верификацию его неизменяемости. Как показано ранее, в начале модульного тестирования создается новый тестовый экземпляр ОБЪЕКТА-ЗНАЧЕНИЯ и конструктор копирования создает его копию. Согласно проверочному утверждению эти экземпляры являются одинаковыми. Далее выполняется проверка, обладает ли экземпляр ЗНАЧЕНИЯ ПОВЕДЕНИЕМ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ. Если все проверочные утверждения выполняются, то последнее утверждение означает, что проверяемые и копируемые экземпляры являются одинаковыми.

Далее мы реализуем СТРАТЕГИЮ типа ЗНАЧЕНИЯ.

```
public float costPercentage(BusinessPriorityTotals aTotals) {
    return (float) 100 * this.ratings().cost() /
        aTotals.totalCost();
}

public float priority(BusinessPriorityTotals aTotals) {
    return
        this.valuePercentage(aTotals) /
            (this.costPercentage(aTotals) +
                  this.riskPercentage(aTotals));
}

public float riskPercentage(BusinessPriorityTotals aTotals) {
    return (float) 100 * this.ratings().risk() /
        aTotals.totalRisk();
}

public float totalValue() {
    return this.ratings().benefit() + this.ratings().penalty();
}

public float valuePercentage(BusinessPriorityTotals aTotals) {
    return (float) 100 * this.totalValue() / aTotals.totalValue();
}
```

```
public BusinessPriorityRatings ratings() {
    return this.ratings;
}
```

Для некоторых вычислительных функций требуется указать параметр типа `BusinessPriorityTotals`. Это ЗНАЧЕНИЕ описывает риск затрат по всем спискам заданий для спринта. Эти числа необходимы для вычисления относительного и абсолютного бизнес-приоритетов всех списков. Ни одна из этих функций не изменяет состояние своего экземпляра. Этот факт проверяется в тестах путем сравнения копируемого значения с текущим значением в ходе выполнения каждой функции.

Пока что это не **ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС (SEPARATED INTERFACE)** [Fowler, P of EAA] СТРАТЕГИИ, потому что существует только одна реализация. Несомненно, со временем он изменится, и клиенты системы управления гибким проектированием SaaS получат другие возможности для вычисления бизнес-приоритетов в соответствии со своими реализациями СТРАТЕГИИ.

Имена ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ играют важную роль. Все они возвращают ЗНАЧЕНИЯ (потому что они являются методами CQS-запроса), поэтому их имена преднамеренно не содержат префикса `get`, предусмотренного правилами именования в пакете JavaBean. Это простой, но эффективный способ проектирования объектов, обеспечивающий соответствие ОБЪЕКТОВ-ЗНАЧЕНИЙ правилам ЕДИНОГО ЯЗЫКА. Вызов функции `getValuePercentage()` — техническая компьютерная инструкция, а вызов функции `valuePercentage()` — обычное выражение, понятное человеку.

Куда делся мой привычный язык Java?

Я думаю, что спецификация JavaBean оказала негативное влияние на проектирование объектов, поскольку она не соответствует принципам предметно-ориентированного проектирования и вообще правилам правильного проектирования объектов. Рассмотрим интерфейс прикладного программирования Java API, существовавший до появления спецификации JavaBean. Возьмем, например, класс `java.lang.String`. В классе `String` есть лишь несколько методов запроса, имеющих префикс `get`. Большинство же методов запроса называются менее формально, например `charAt()`, `compareTo()`, `concat()`, `contains()`, `endsWith()`, `indexOf()`, `length()`, `replace()`, `startsWith()`, `substring()` и т.д. Спецификацией JavaBean здесь и не пахнет! Конечно, сам по себе этот пример не доказывает мою точку зрения. Тем не менее спецификация JavaBean оказала сильное влияние на интерфейсы Java API и снизила их выразительность. Удобочитаемость и выразительность языкового выражения — очень важное свойство стиля.

Если вы планируете разработать инструментарий, зависящий от спецификации JavaBean, то обратите внимание на существующие решения этой проблемы. Например, библиотека Hibernate обеспечивает доступ к полям (атрибутам объекта). Следовательно, используя библиотеку Hibernate, вы можете называть методы, как захотите, без отрицательных последствий.

Впрочем, использование других инструментов для проектирования выразительных интерфейсов может дать побочный эффект. Если вы решите использовать стандартные языки Java EL и OGNL, например, то не сможете представлять такие типы непосредственно. Вам потребуются другие средства, такие как проектный шаблон **ОБЪЕКТ ПЕРЕДАЧИ ДАННЫХ (DATA TRANSFER OBJECT)** [Fowler, P of EAA] с методами-получателями, чтобы преобразовать свойства ОБЪЕКТА-ЗНАЧЕНИЯ в пользовательский интерфейс. Поскольку шаблон DTO используется довольно широко, часто без достаточных технических оснований, могут возникнуть определенные последствия. Если шаблон DTO вам не подходит, обратите внимание на другие шаблоны. Рассмотрите возможность использования шаблона **МОДЕЛЬ ПРЕЗЕНТАЦИИ (PRESENTATION MODEL)**, которая обсуждается в главе, посвященной шаблону **ПРИЛОЖЕНИЕ (APPLICATION) (14)**. Поскольку ваша МОДЕЛЬ ПРЕЗЕНТАЦИИ может служить **АДАПТЕРОМ (ADAPTER)** [Гамма и др.], она может, например, предоставлять методы-получатели для представлений, основанных на языке EL. Если же и это вас не устроит, то вам придется скрепя сердце проектировать объекты предметной области с методами-получателями.

Если вы придете к такому заключению, то вам не следует разрабатывать ОБЪЕКТЫ-ЗНАЧЕНИЯ с полноценными возможностями пакета JavaBean, допускающими инициализацию с помощью открытых методов-установщиков. Это нарушило бы важное свойство неизменяемости ЗНАЧЕНИЯ.

Следующий набор методов включает стандартное замещение функций `equals()`, `hashCode()` и `toString()`.

```
@Override
public boolean equals(Object anObject) {
    boolean equalObjects = false;
    if (anObject != null &&
            this.getClass() == anObject.getClass()) {
        BusinessPriority typedObject = (BusinessPriority) anObject;
        equalObjects =
            this.ratings().equals(typedObject.ratings());
    }
    return equalObjects;
}

@Override
public int hashCode() {
    int hashCodeValue =
        + (169065 * 179)
        + this.ratings().hashCode();
```

```
    return hashCodeValue;
}

@Override
public String toString() {
    return
       "BusinessPriority"
       + " ratings = " + this.ratings(); }
```

Метод `equals()` реализует одну из пяти характеристик ЗНАЧЕНИЯ — требование проверять равенство ЗНАЧЕНИЙ. Здесь проверка равенства никогда не включает сравнение со значением `null`. Класс параметра должен совпадать с классом ЗНАЧЕНИЯ. Если они совпадают, то выполняется сравнение всех свойств/атрибутов обоих ЗНАЧЕНИЙ. Если все свойства/атрибуты совпадают с соответствующими свойствами/атрибутами, ЦЕЛОСТНЫЕ ЗНАЧЕНИЯ считаются одинаковыми.

В соответствии со стандартом Java функция `hashCode()` во всех равных ЗНАЧЕНИЯХ имеет такой же контракт, как и функция `equals()`, и возвращает хешированные значения кода.

В функции `toString()` нет ничего особенного. Она создает удобочитаемое представление состояния экземпляра ЗНАЧЕНИЯ. При необходимости можно разработать формат этого представления.

Для обзора осталось несколько методов.

```
protected BusinessPriority() {
   super();
}

private void setRatings(BusinessPriorityRatings aRatings) {
    if (aRatings == null) {
            throw new IllegalArgumentException(
                   "The ratings are required.");
       }
       this.ratings = aRatings;
    }
}
```

Конструктор без параметров предназначен для работы с каркасом, которому он нужен, например с библиотекой Hibernate. Поскольку конструктор без параметров всегда является скрытым, нет никакой опасности, что клиенты модели создадут некорректные экземпляры. Функции библиотеки Hibernate прекрасно работают со скрытыми конструкторами и методами доступа. Этот конструктор позволяет библиотеке Hibernate и другим инструментам создавать экземпляры требуемого типа, например хранилища данных. Инструменты используют конструкторы без параметров для создания изначально пустого экземпляра, а затем вызывают методы-установщики каждого свойства/атрибута для наполнения

этого объекта. При желании можно попросить библиотеку Hibernate проигнорировать методы-установщики и непосредственно задавать атрибуты, как это сделано в описываемой модели, которая не полностью поддерживает интерфейс JavaBean. Напоминаем еще раз, что клиенты модели всегда используют открытые конструкторы и никогда не используют закрытые.

Определение класса заканчивается методом-установщиком свойства `ratings`. Этот метод демонстрирует одно из преимуществ самоинкапсуляции/самоделегирования. Метод доступа — получатель или установщик — не обязан ограничиваться заданием поля экземпляра. Он также проверяет важные **УТВЕРЖДЕНИЯ (ASSERTIONS)** [Эванс] — ключевой элемент успешной разработки программного обеспечения в принципе и DDD-моделей в частности.

УТВЕРЖДЕНИЕ, относящееся к корректным параметрам, называется *предохранителем* (guard), потому что он защищает метод от неправильных данных. Предохранители можно и нужно использовать в любом методе, в котором неправильные параметры могут привести к серьезным последствиям, если их корректность не гарантирована другим способом. В данном случае метод-установщик проверяет, что параметр `aRatings` не равен нулю, а если он равен нулю, то метод генерирует исключение `IllegalArgumentException`. Действительно, метод-установщик логично использовать только один раз на протяжении всего жизненного цикла ЗНАЧЕНИЯ. Впрочем, УТВЕРЖДЕНИЕ — отличный предохранитель, расположенный в правильном месте. Кроме того, в любом коде мы видим преимущества самоделегирования. В частности, методы проверки значений как часть общей стратегии проверки корректности описываются в главе, посвященной **СУЩНОСТЯМ (ENTITIES) (5)**.

Хранение ОБЪЕКТОВ-ЗНАЧЕНИЙ

Существует множество способов записи экземпляров ОБЪЕКТА-ЗНАЧЕНИЯ в постоянное хранилище. В широком смысле они включают в себя сериализацию объекта в текстовом или двоичном формате и сохранение его на диске. Однако, поскольку нас не интересует сам механизм постоянного хранения экземпляров ЗНАЧЕНИЯ, мы не станем останавливаться на универсальном постоянном хранении. Вместо этого нас интересует постоянное хранение ЗНАЧЕНИЙ вместе с состоянием экземпляров содержащего их АГРЕГАТА. Рассматриваемые ниже подходы основаны на предположении, что родительская СУЩНОСТЬ хранит ссылки на экземпляры хранящихся ЗНАЧЕНИЙ. Кроме того, все следующие примеры основаны на предположении, что агрегат добавляется или считывается из своего **ХРАНИЛИЩА (REPOSITORY) (12)**, а содержащиеся в нем ЗНАЧЕНИЯ хранятся и скрыто воспроизводятся вместе с СУЩНОСТЬЮ, например с КОРНЕМ АГРЕГАТА, который их содержит.

Широко распространенным способом постоянного хранения ОБЪЕКТОВ-ЗНАЧЕНИЙ является объектно-реляционное отображение (Object-relational mapping — ORM, например Hibernate). Однако использование механизма ORM для отображения каждого класса в таблицу и каждого атрибута в столбец повышает сложность, что может быть нежелательным. В настоящее время растет популярность использования баз данных NoSQL и хранилищ типа "ключ-значение" благодаря их способности обеспечивать высокопроизводительные, масштабируемые, устойчивые к ошибкам и легко доступные промышленные хранилища. Кроме того, хранилище типа "ключ-значение" может значительно упростить постоянное хранение АГРЕГАТА. В этой главе рассматривается механизм постоянного хранения, основанный на принципах ORM. Поскольку хранилища типа "ключ-значение" относятся к технологиям NoSQL и поэтому исключительно хорошо хранят АГРЕГАТЫ, я рассматриваю этот стиль в главе, посвященной **ХРАНИЛИЩАМ (12)**.

Однако, прежде чем перейти к примерам хранения ЗНАЧЕНИЙ с помощью механизма ORM, необходимо хорошо понять очень важное условие моделирования и строго его выполнять. Исходя из этого, для начала мы посмотрим, что произойдет, если моделирование данных (в противоположность моделированию предметной области) оказывает недопустимое влияние на вашу модель предметной области, и что можно сделать для того, чтобы предотвратить это вредное и опасное влияние.

Предотвращение чрезмерного влияния утечки информации из модели данных

В подавляющем большинстве случаев, когда ОБЪЕКТ-ЗНАЧЕНИЕ помещается в хранилище (например, с помощью механизма ORM и реляционной базы данных), он сохраняется в денормализованном виде; иначе говоря, его атрибуты хранятся в той же строке таблицы базы данных, в которой записан его родительский объект СУЩНОСТИ. Это упрощает и оптимизирует запись и извлечение ЗНАЧЕНИЙ, а также предотвращает утечку информации из постоянного хранилища в модель. Это удобный и простой способ хранения ЗНАЧЕНИЙ.

Однако иногда ОБЪЕКТ-ЗНАЧЕНИЕ в модели обязан храниться в постоянном реляционном хранилище как СУЩНОСТЬ. Иначе говоря, экземпляр конкретного ОБЪЕКТА-ЗНАЧЕНИЯ будет занимать отдельную строку в реляционной базе данных, предназначенной именно для этого типа данных, причем в таблице будет предусмотрен отдельный столбец для его первичного ключа. Например, это происходит при поддержке коллекции экземпляров ОБЪЕКТОВ-ЗНАЧЕНИЙ с помощью механизма ORM. В этих ситуациях данные типа ЗНАЧЕНИЯ моделируются как сущности базы данных.

Свидетельствует ли это о том, что объект предметной области должен отражать модель данных и быть СУЩНОСТЬЮ, а не ЗНАЧЕНИЕМ? Нет. Когда происходит такая потеря соответствия, важно придерживаться точки зрения модели предметной области, а не механизма постоянного хранения. Для того чтобы сохранить ориентацию на модель предметной области, задайте себе несколько вопросов.

1. Является ли моделируемая концепция вещью из предметной области или она измеряет, подсчитывает или описывает вещь как его свойство?
2. Если концепция модели правильно описывает элемент предметной области, должна ли она обладать всеми или большинством характеристик значения?
3. Рассматриваете ли вы использование СУЩНОСТИ только потому, что базовая модель данных должна хранить объект модели предметной области как сущность?
4. Используете ли вы СУЩНОСТЬ, потому что модель предметной области требует наличия уникального идентификатора, интересуют ли вас индивидуальные экземпляры и должны ли вы управлять непрерывными изменениями этих экземпляров на протяжении их жизненного цикла?

Если вы ответили "Описывает, да, да и нет", то должны использовать ОБЪЕКТ-ЗНАЧЕНИЕ. Моделируя постоянное хранилище, предназначенное для хранения объекта, не позволяйте ему влиять на концептуализацию свойства ЗНАЧЕНИЯ в модели предметной области.

Модель данных должна быть подчиненной

Разрабатывайте модель данных с учетом модели предметной области, а не модель предметной области с учетом модели данных.

По возможности всегда проектируйте модель данных с учетом модели предметной области, а не модель предметной области с учетом модели данных. В первом случае вы придерживаетесь точки зрения модели предметной области. Во втором случае вы становитесь на точку зрения модели данных, а ваша модель предметной области становится простой проекцией модели данных. Когда вы привыкнете рассуждать в терминах модели предметной области, а не модели данных, т.е. станете заниматься действительно предметно-ориентированным проектированием, то сможете избежать негативных последствий, связанных с утечкой информации из модели данных. Образ мышления в рамках предметно-ориентированного моделирования описывается в главе, посвященной **СУЩНОСТЯМ (5)**.

Разумеется, в некоторых ситуациях важна ссылочная целостность базы данных (например, для внешних ключей). Конечно, столбцы ключей должны быть правильно индексированы. Естественно, нужна поддержка механизмов,

генерирующих бизнес-отчеты и обрабатывающих бизнес-данные. Важность всех этих факторов можно оценить. Большинство специалистов считают, что механизмы генерации отчетов и анализа данных не должны оперировать производственными данными, а вместо этого необходимо использовать специализированную и тщательно спроектированную модель данных. Такой стратегический подход освобождает модель предметной области от влияния модели данных и лучше соответствует принципам DDD.

Какие бы технические факторы не влияли на вашу модель данных, ее сущности, первичные ключи, ссылочная целостность и индексы никак не влияли бы на объекты предметной области. Подход DDD не касается структуризации данных в нормализованном виде. В центре его внимания находится ЕДИНЫЙ ЯЗЫК в соответствующем ОГРАНИЧЕННОМ КОНТЕКСТЕ. Я призываю читателей придерживаться принципов DDD, а не структуры данных. В этом случае следует внимательно продумывать каждый шаг, чтобы предотвратить утечку информации из модели данных (которая происходит при использовании механизма ORM, хотя, возможно, в минимальной степени) в вашу модель предметной области и клиентам. Этот вопрос я рассматриваю в следующем разделе.

Механизм ORM и отдельные объекты-значения

Хранение отдельного экземпляра ОБЪЕКТА-ЗНАЧЕНИЯ в базе данных обычно реализуется очень просто. Здесь мы сосредоточимся на использовании библиотеки Hibernate и реляционной базы данных MySQL. Основная идея заключается в том, чтобы хранить все атрибуты ЗНАЧЕНИЯ в отдельных столбцах строки, в которой хранится родительская СУЩНОСТЬ. Иначе говоря, отдельный ОБЪЕКТ-ЗНАЧЕНИЕ денормализуется в строку своей родительской СУЩНОСТИ. Целесообразно принять соглашение об именах столбцов и стандартизировать способ именования сериализированных объектов. В этом разделе я опишу правила выбора имен для хранящихся ОБЪЕКТОВ-ЗНАЧЕНИЙ.

Используя библиотеку Hibernate для хранения отдельного экземпляра объекта-значения, применяйте элемент отображения `component`. Этот элемент позволяет непосредственно отобразить ЗНАЧЕНИЕ в строку таблицы родительской сущности в денормализованном виде. Это оптимальный метод сериализации, позволяющий включать ЗНАЧЕНИЯ в запросы SQL. Рассмотрим раздел документа из библиотеки Hibernate, который описывает отображение ОБЪЕКТА-ЗНАЧЕНИЯ `BusinessPriority`, который находится в его родительской СУЩНОСТИ — классе `BacklogItem`.

```
<component name="businessPriority"
    class="com.saasovation.agilepm.domain.model.product. ↵
        BusinessPriority">
```

```
    <component name="ratings"
           class="com.saasovation.agilepm.domain.model.product. ↵
               BusinessPriorityRatings">
        <property
            name="benefit"
            column="business_priority_ratings_benefit"
            type="int"
            update="true"
            insert="true"
            lazy="false"
            />
        <property
            name="cost"
            column="business_priority_ratings_cost"
            type="int"
            update="true"
            insert="true"
            lazy="false"
            />
        <property
            name="penalty"
            column="business_priority_ratings_penalty"
            type="int"
            update="true"
            insert="true"
            lazy="false"
            />
        <property
            name="risk"
            column="business_priority_ratings_risk"
            type="int"
            update="true"
            insert="true"
            lazy="false"
            />
    </component>
</component>
```

Это хороший пример, потому что он демонстрирует простое отображение ОБЪЕКТА-ЗНАЧЕНИЯ, который содержит дочерний экземпляр. Напомним, что класс `BusinessPriority` имеет одно свойство ЗНАЧЕНИЯ `ratings` и не имеет дополнительных атрибутов. Таким образом, в описании отображения внешний элемент `component` имеет вложенный элемент `component`. Он используется для денормализации свойства вложенного ЗНАЧЕНИЯ `ratings` типа `BusinessPriorityRatings`. Поскольку класс `BusinessPriority` не имеет своих атрибутов, во внешний элемент `component` ничего не отображается. Вместо этого мы прямо выполняем вложение отображения свойства ЗНАЧЕНИЯ `ratings`. В заключение мы реально сохраняем только четыре целочисленных атрибута экземпляра `BusinessPriorityRatings` в четыре отдельных столбца таблицы `tbl_backlog_item`. Итак, мы отображаем два ОБЪЕКТА-ЗНАЧЕНИЯ

component, один из которых не имеет своих атрибутов, и внутреннее ЗНАЧЕНИЕ, имеющее четыре атрибута.

Отметим, что мы использовали стандартные правила именования столбцов для каждого из элементов property из библиотеки Hibernate. Правила именования используют путь навигации от родительского ЗНАЧЕНИЯ вниз к отдельным атрибутам. Например, рассмотрим путь навигации из класса BusinessPriority к атрибуту benefit экземпляра ValueCostRiskRatings. Вполне логично, что он называется

```
businessPriority.ratings.benefit
```

Для того чтобы представить путь навигации в виде имени отдельного реляционного столбца, я использовал следующее имя.

```
business_priority_ratings_benefit
```

Разумеется, при желании вы можете использовать другое репрезентативное имя. Возможно, вы предпочтете смешанный "верблюжий" стиль и знаки подчеркивания.

```
businessPriority_ratings_benefit
```

Это простое обозначение лучше выражает навигацию. Я стандартизировал все знаки подчеркивания, поскольку они напоминают традиционные имена столбцов SQL, а не имена объектов. Соответствующее определение таблицы в базе данных MySQL содержит следующие столбцы.

```
CREATE TABLE 'tbl_backlog_item' (
    ...
    'business_priority_ratings_benefit' int NOT NULL,
    'business_priority_ratings_cost' int NOT NULL,
    'business_priority_ratings_penalty' int NOT NULL,
    'business_priority_ratings_risk' int NOT NULL,
    ...
) ENGINE=InnoDB;
```

Отображение Hibernate и определение таблицы реляционной базы данных образуют оптимальный сохраняемый объект, допускающий запросы. Поскольку атрибуты ЗНАЧЕНИЯ денормализуются в строку таблицы родительской СУЩНОСТИ, нет необходимости использовать объединения в базе данных даже для извлечения глубоко вложенного экземпляра ЗНАЧЕНИЯ. При формулировке запроса HQL каркас Hibernate допускает легкое отображение из объектного выражения атрибута объекта в оптимальный SQL-запрос с помощью столбца, где имя

```
businessPriority.ratings.benefit
```

принимает вид

```
business_priority_ratings_benefit
```

Следовательно, хотя существует явная несогласованность между объектами и реляционными базами данных, мы реализовали более чем функциональное и оптимальное отображение.

Механизм ORM и многочисленные значения, сериализованные в одном столбце

Отображение коллекции многочисленных ОБЪЕКТОВ-ЗНАЧЕНИЙ в реляционную базу данных с помощью механизма ORM порождает много уникальных проблем. Под коллекцией я имею в виду контейнеры `List` и `Set`, содержащие СУЩНОСТЬ, а также один или несколько экземпляров ЗНАЧЕНИЙ или не содержащие ни одного ЗНАЧЕНИЯ. Эти проблемы не являются неразрешимыми, но объектно-реляционная несогласованность в этих ситуациях проявляется особенно ярко.

Среди прочих возможностей объектно-реляционного отображения библиотека Hibernate позволяет осуществлять сериализацию всей коллекции объектов в текстуальное представление и сохранять это представление в отдельном столбце. У этого подхода есть недостатки. Впрочем, в некоторых случаях недостатки незаметны и могут быть проигнорированы в пользу усиления преимуществ этой возможности. В таких случаях можно использовать возможность хранения набора ЗНАЧЕНИЙ. Рассмотрим потенциальные недостатки этого подхода.

- *Максимальный размер столбца.* Иногда невозможно определить максимальное количество ЗНАЧЕНИЙ в коллекции или максимальный размер каждого сериализованного ЗНАЧЕНИЯ. Например, некоторые коллекции объектов должны содержать произвольное количество элементов, т.е. их верхний предел не определен. Кроме того, все ЗНАЧЕНИЯ в коллекции могут иметь неопределенную длину символьного представления. Это может произойти, если один или несколько атрибутов типа ЗНАЧЕНИЯ имеют тип `String`, а количество символов, из которых он состоит, велико или не определено. В обоих случаях сериализованное представление или вся коллекция может превысить допустимый размер символьного столбца. Ситуация может усложниться, если символьные столбцы имеют относительно небольшой максимальный размер или для хранения всей строки данных доступно недостаточно большое максимальное количество байтов.

Например, в механизме базы данных MySQL InnoDB максимальный размер типа VARCHAR составляет 65 535 символов, тогда как для хранения отдельной строки выделяется только 65 535 байт. Для хранения всей СУЩНОСТИ необходимо выделить достаточное количество столбцов. В базе данных Oracle Database максимальный размер типа VARCHAR2/NVARCHAR2 равен 4000. Если невозможно заранее определить максимальное количество байтов для хранения сериализованного представления ЗНАЧЕНИЯ и/или существует опасность превысить максимальный размер столбца, то этой возможности следует избегать.

- *Обязательный запрос.* Поскольку в рассматриваемом нами виде коллекций ЗНАЧЕНИЯ сериализуются в обычное текстовое представление, атрибуты ЗНАЧЕНИЙ нельзя использовать в выражениях SQL-запросов. Если все атрибуты значений должны допускать запросы, эта возможность исчезает. Впрочем, необходимость запрашивать один или несколько атрибутов возникает редко.
- *Требуется пользовательский тип.* Для использования этого подхода необходимо разрабатывать специальный пользовательский тип Hibernate, управляющий сериализацией и десериализацией каждой коллекции. Лично я считаю, что это требование не слишком обременительное, потому что отдельная, продуманная, специализированная реализация пользовательского типа может поддерживать коллекции ОБЪЕКТОВ-ЗНАЧЕНИЙ любого типа (один размер подходит для всех).

Я не привожу здесь специализированный пользовательский тип Hibernate для управления сериализацией коллекции в отдельный столбец, но в среде пользователей каркаса Hibernate существует огромное количество описаний таких классов.

Механизм ORM и многочисленные значения на основе сущности из базы данных

Существует очень простой подход к реализации постоянного хранения коллекций экземпляров ЗНАЧЕНИЙ с помощью библиотеки Hibernate (или другого механизма ORM) и реляционной базы данных, основанный на представлении типа ЗНАЧЕНИЯ в виде СУЩНОСТИ из модели данных. Как было сказано в разделе "Предотвращение чрезмерного влияния утечки информации из модели данных", этот подход *не должен* приводить к неправильному моделированию концепции в виде СУЩНОСТИ внутри модели предметной области, потому что для обеспечения постоянного хранения ее лучше всего представлять в виде сущности базы данных. В некоторых ситуациях объектно-ориентированная несогласованность

требует применения именно этого подхода, а не принципов DDD. Если бы существовал идеально согласованный стиль постоянного хранения, то можно было бы моделировать эту концепцию как тип ЗНАЧЕНИЯ и никогда не считать ее СУЩНОСТЬЮ базы данных. Это точнее соответствовало бы предметно-ориентированному стилю моделирования.

Для реализации этого подхода можно применить шаблон **СУПЕРТИП УРОВНЯ (LAYER SUPERTYPE)** [Fowler, P of EAA]. Лично я предпочитаю избегать суррогатных сущностей (первичных ключей). Однако, поскольку каждый экземпляр класса `Object` в языке Java (и других языках) уже имеет внутренний уникальный идентификатор, который используется только виртуальной машиной, целесообразно включить специальный идентификатор непосредственно в ЗНАЧЕНИЕ. Какой бы выбор вы ни сделали, объектно-реляционная несогласованность заставит вас обосновать свое техническое решение. Ниже я описываю свои предпочтения по этому вопросу.

Рассмотрим пример, касающийся суррогатных ключей, в котором используются классы СУПЕРТИПА УРОВНЯ.

```
public abstract class IdentifiedDomainObject
        implements Serializable {

   private long id = -1;

   public IdentifiedDomainObject() {
        super();
   }

   protected long id() {
        return this.id;
   }

    protected void setId(long anId) {
        this.id = anId;
    }
}
```

Первым СУПЕРТИПОМ УРОВНЯ является класс `IdentifiedDomainObject`. Этот абстрактный базовый класс обеспечивает суррогатный первичный ключ, скрытый от клиентского представления. Поскольку методы доступа объявлены как `protected`, клиенты никогда не узнают об их существовании. Разумеется, их можно скрыть еще больше, использовав ключевое слово `private`. В библиотеке Hibernate нет проблем с использованием отражения метода или поля, имеющего любую другую область видимости, помимо `public`.

Далее, я написал еще один СУПЕРТИП УРОВНЯ, специфичный для ОБЪЕКТОВ-ЗНАЧЕНИЙ.

```
public abstract class IdentifiedValueObject
       extends IdentifiedDomainObject {

   public IdentifiedValueObject() {
          super();
   }
}
```

Класс `IdentifiedValueObject` можно считать просто классом-маркером, подклассом класса `IdentifiedDomainObject` без методов. Он предназначен для документирования исходного кода, позволяя более четко сформулировать задачи моделирования. Кроме того, класс `IdentifiedDomainObject` имеет второй непосредственный производный абстрактный подкласс `Entity`, который обсуждается в главе, посвященной **СУЩНОСТЯМ (5)**. Мне нравится этот подход, но вы можете исключить эти вспомогательные классы.

Теперь, имея удобный и скрытый механизм для создания суррогатного идентификатора для любого типа значения, приведем простой класс, демонстрирующий его использование.

```
public final class GroupMember extends IdentifiedValueObject {
    private String name;
    private TenantId tenantId;
    private GroupMemberType type;

    public GroupMember(
            TenantId aTenantId,
            String aName,
            GroupMemberType aType) {

    this();
    this.setName(aName);
    this.setTenantId(aTenantId);
    this.setType(aType);
    this.initialize();

    }
    ...
}
```

Класс `GroupMember` — это тип ЗНАЧЕНИЯ, которое входит в коллекцию КОРНЕВОЙ СУЩНОСТИ агрегатного класса `Group`. КОРНЕВАЯ СУЩНОСТЬ содержит произвольное количество экземпляров класса `GroupMember`. Теперь каждый экземпляр класса `GroupMember` можно однозначно идентифицировать в рамках модели данных с помощью его суррогатного первичного ключа и отобразить его в сущность базы данных, сохранив значение в МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ. Ниже приведен соответствующий фрагмент класса `Group`.

```
public class Group extends Entity {
    private String description;
    private Set<GroupMember> groupMembers;
    private String name;
    private TenantId tenantId;

    public Group(
            TenantId aTenantId,
            String aName,
            String aDescription) {
       this();
       this.setDescription(aDescription);
       this.setName(aName);
       this.setTenantId(aTenantId);
       this.initialize();
    }
    ...
    protected Group() {
       super();
       this.setGroupMembers(new HashSet<GroupMember>(0));
    }
    ...
}
```

Класс `Group` включает произвольное количество экземпляров класса `GroupMember` в контейнер `Set` с именем `groupMembers`. Учтите, что если вы даже полностью замените всю коллекцию, перед заменой всегда используйте метод `clear()` из класса `Collection`. Это гарантирует, что базовая реализация класса `Collection` из каркаса Hibernate удалит устаревшие элементы из базы данных. Ниже приведен не настоящий метод класса `Group`, а пример, демонстрирующий, как избежать потери ЗНАЧЕНИЙ при замене всей коллекции.

```
public void replaceMembers(Set<GroupMember> aReplacementMembers) {
    this.groupMembers().clear();
    this.setGroupMembers(aReplacementMembers);
}
```

Я полагаю, что утечка информации из механизма ORM в модель является незаметной, потому что она использует общие возможности класса `Collection`, и, кроме того, клиент его не видит. Синхронизация содержания коллекции с базой данных не всегда требует осторожности. Простое удаление хранилища ЗНАЧЕНИЙ автоматически осуществляется методом `remove()` класса `Collection`, и в этом случае утечка информации из механизма ORM не возникает вообще.

Далее, рассмотрим раздел класса `Group`, который осуществляет отображение коллекции.

```
<hibernate-mapping>
    <class name="com.saasovation.identityaccess.domain.model.
                identity.Group"
    table="tbl_group" lazy="true">
       ...
       <set name="groupMembers" cascade="all,delete-orphan"
          inverse="false" lazy="true">
            <key column="group_id" not-null="true" />
              <one-to-many class="com.saasovation.[ccc]
                identityaccess.domain.model.identity.GroupMember" />
       </set>
       ...
    </class>
</hibernate-mapping>
```

Контейнер `Set`, содержащий элементы `groupMembers`, точно отображается как сущность базы данных. Кроме того, мы видим полное описание отображения `GroupMember`.

```
<hibernate-mapping>
   <class name="com.saasovation.identityaccess.domain.model.
          identity.GroupMember"
          table="tbl_group_member" lazy="true">
      <id
          name="id"
          type="long"
          column="id"
          unsaved-value="-1">
          <generator class="native"/>
     </id>
     <property
          name="name"
          column="name"
          type="java.lang.String"
          update="true"
          insert="true"
          lazy="false"
     />
     <component name="tenantId"
          class="com.saasovation.identityaccess.domain.model.
              identity.TenantId">
          <property
              name="id"
              column="tenant_id_id"
              type="java.lang.String"
              update="true"
              insert="true"
              lazy="false"
          />
      </component>
      <property
           name="type"
```

```
            column="type"
            type="com.saasovation.identityaccess.infrastructure.
                    persistence.GroupMemberTypeUserType"
            update="true"
            insert="true"
            not-null="true"
        />
    </class>
</hibernate-mapping>
```

Обратите внимание на элемент <id>, определяющий суррогатный первичный ключ для постоянного хранения. И наконец рассмотрим описание таблицы tbl_group_member базы данных MySQL.

```
CREATE TABLE 'tbl_group_member' (
     'id' int(11) NOT NULL auto_increment,
     'name' varchar(100) NOT NULL,
     'tenant_id_id' varchar(36) NOT NULL,
     'type' varchar(5) NOT NULL,
     'group_id' int(11) NOT NULL,
     KEY 'k_group_id' ('group_id'),
     KEY 'k_tenant_id_id' ('tenant_id_id'),
     CONSTRAINT 'fk_1_tbl_group_member_tbl_group'
     FOREIGN KEY ('group_id`) REFERENCES 'tbl_group' ('id'),
     PRIMARY KEY ('id')
) ENGINE=InnoDB;
```

Глядя на отображение GroupMember и описание таблицы базы данных, можно получить ясное впечатление, что мы работаем с сущностью. Существует первичный ключ с именем id. Существует отдельная таблица, которую необходимо объединить с таблицей tbl_group. Существует внешний ключ, осуществляющий связь с таблицей tbl_group. Любое другое имя относится к сущности, *но только с точки зрения модели данных*. В модели предметной области отображение GroupMember, очевидно, является ОБЪЕКТОМ-ЗНАЧЕНИЕМ. В модели предметной области были сделаны правильные шаги, для того чтобы тщательно скрыть все, что относится к механизму постоянного хранения. У клиентов нет никаких шансов обнаружить утечку информации о механизме постоянного хранения из модели предметной области. И что важнее всего, даже разработчики модели должны приложить немалые усилия, чтобы обнаружить эту утечку.

Механизм ORM и многочисленные значения на основе объединенной таблицы

Библиотека Hibernate предоставляет средства для постоянного хранения многозначных коллекций в объединенной таблице, не требуя от типа ЗНАЧЕНИЯ

наличия характеристик сущности из модели данных. Этот тип отображения просто сохраняет ЗНАЧЕНИЯ из коллекции в предназначенную для этого таблицу с внешним ключом, который является идентификатором СУЩНОСТИ объекта предметной области в базе данных. Таким образом, все ЗНАЧЕНИЯ из коллекции можно запросить по их родительскому внешнему ключу и вернуть в коллекцию ЗНАЧЕНИЙ. Преимущество этого подхода заключается в том, что тип ЗНАЧЕНИЯ не обязан иметь скрытый суррогатный идентификатор, предназначенный для объединения. Для использования этого отображения коллекции ЗНАЧЕНИЯ используется дескриптор `<composite-element>` из библиотеки Hibernate.

Это кажется большой победой и может оказаться полезным. Однако следует знать и о слабостях этого подхода. Один из недостатков заключается в том, что объединение необходимо, даже если тип ЗНАЧЕНИЯ не требует суррогатного ключа, потому что он использует нормализацию двух таблиц. На самом деле подход "механизм ORM и многочисленные значения, основанные на сущности базы данных" также требует объединения. В то же время этот подход имеет и другие недостатки.

Если в качестве коллекции используется контейнер `Set`, ни один из атрибутов типа ЗНАЧЕНИЯ не может быть равным `null`. Это объясняется тем, что для удаления заданного элемента контейнера `Set` (при сборе мусора в модели данных) необходимо использовать все атрибуты, однозначно определяющие это ЗНАЧЕНИЕ. Значение `null` нельзя использовать как часть требуемого сложного ключа. Конечно, если известно, что заданный тип ЗНАЧЕНИЯ никогда не имеет атрибутов `null`, этот подход становится целесообразным, т.е. никаких дополнительных конфликтов не возникает.

Третий недостаток использования отображения заключается в том, что отображаемый тип значения сам не может содержать коллекцию. Для элементов, которые сами являются коллекцией, в компоненте `<composite-element>` не предусмотрено никаких механизмов. Если же тип ЗНАЧЕНИЙ не содержит коллекций любого вида и выполняет все остальные требования, предъявляемые стилем, основанным на отображении, его можно безопасно использовать.

В заключение, я считаю, что подход, основанный на отображении, является настолько ограниченным, что в принципе его следует избегать. Мне кажется, что намного проще поместить скрытый суррогатный идентификатор в тип ЗНАЧЕНИЙ, входящий в коллекцию с помощью связи "один-ко-многим", и не беспокоиться об ограничениях компонента `<composite-element>`. Вы можете со мной не согласиться и воспользоваться своими преимуществами, обладая всей информацией.

ORM и ОБЪЕКТЫ, в которых состояние представлено перечислением

Если в качестве эффективного средства моделирования объектов СТАНДАРТНЫХ ТИПОВ и/или СОСТОЯНИЯ вы хотите использовать перечисления, то вам потребуются инструменты для их хранения. В сочетании с библиотекой Hibernate перечисления в языке Java требуют специального способа постоянного хранения. К сожалению, библиотека Hibernate не имеет готового инструмента для поддержки перечислений как стандартного типа свойства. Следовательно, для постоянного хранения перечислений в модели необходимо разработать специальный пользовательский тип Hibernate.

Напомним, что каждый экземпляр класса `GroupMember` содержит экземпляр типа `GroupMemberType`.

```
public final class GroupMember extends IdentifiedValueObject {
    private String name;
    private TenantId tenantId;
    private GroupMemberType type;

    public GroupMember(
            TenantId aTenantId,
            String aName,
            GroupMemberType aType) {
       this();
       this.setName(aName);
       this.setTenantId(aTenantId);
       this.setType(aType);
       this.initialize();
    }
    ...
}
```

Перечисление СТАНДАРТНЫХ ТИПОВ `GroupMemberType` включает типы `GROUP` и `USER`. Рассмотрим их определения.

```
package com.saas ovation.identityaccess.domain.model.identity;

public enum GroupMemberType {

    GROUP {
        public boolean isGroup() {
             return true;
        }
    },
    USER {
        public boolean isUser() {
             return true;
        }
    };
```

```
    public boolean isGroup() {
        return false;
    }

   public boolean isUser() {
        return false;
   }
}
```

Простой ответ заключается в том, чтобы хранить перечисление ЗНАЧЕНИЙ в виде их текстового представления. Однако этот простой ответ ведет к развертыванию немного более сложного механизма для создания пользовательского типа Hibernate. Чтобы не описывать здесь использование разных подходов к классу `EnumUserType`, существующих в среде Hibernate, я написал статью и разместил ее по адресу `http://community.jboss.org/wiki/Java5EnumUserType`.

В этой статье описано несколько подходов. В ней приведены примеры реализации специального пользовательского типа для каждого типа перечислений; способ использования параметризованных типов Hibernate 3, позволяющих избежать реализации пользователя для каждого типа перечислений (очень желательное свойство); поддержка не только текстовой строки, но и числовых представлений значений перечисления, а также усовершенствованная реализация Гэвина Кинга (Gavin King). Эта реализация позволяет использовать перечисление как дискриминатор типа или как идентификатор таблицы данных (`id`).

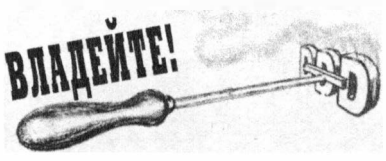
Выбрав одну из этих возможностей, рассмотрим пример отображения перечисления `GroupMemberType`.

```
<hibernate-mapping>
   <class name="com.saasovation.identityaccess.domain.model.
          identity.GroupMember" table="tbl_group_member" lazy="true">
       ...
       <property
          name="type"
          column="type"
          type="com.saasovation.identityaccess.infrastructure.
                  persistence.GroupMemberTypeUserType"
          update="true"
          insert="true"
          not-null="true"
        />
    </class>
</hibernate-mapping>
```

Отметим, что атрибут типа элемента `<property>` представляет собой полный путь класса `GroupMemberTypeUserType`. Это просто одна из потенциальных возможностей. Вы можете выбрать любую из них. Напомним, что описание таблицы MySQL содержит столбец для хранения перечисления.

```
CREATE TABLE 'tbl_group_member' (
    ...
    'type' varchar(5) NOT NULL,
    ...
) ENGINE=InnoDB;
```

Столбец имеет тип VARCHAR, максимальный размер которого равен пяти символам, и этого достаточно для хранения самого широкого текстового значения: GROUP или USER.



Резюме

В главе продемонстрирована важность максимально широкого использования ОБЪЕКТОВ-ЗНАЧЕНИЙ, потому что их проще разрабатывать, тестировать и поддерживать.

- Вы изучили характеристики ОБЪЕКТОВ-ЗНАЧЕНИЙ и научились их использовать.
- Вы увидели, как с помощью ОБЪЕКТОВ-ЗНАЧЕНИЙ можно минимизировать сложность интеграции.
- Вы научились использовать СТАНДАРТНЫЕ ТИПЫ предметной области, выраженные как ЗНАЧЕНИЯ, и узнали пять стратегий их реализации.
- Вы увидели, как компания SaaSOvation перешла на максимально широкое использование ЗНАЧЕНИЙ.
- Вы научились тестировать, реализовывать и хранить типы ЗНАЧЕНИЙ в рамках проектов SaaSOvation.

Далее мы рассмотрим СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, представляющие собой операции без состояния, которые являются частью модели.

Глава 7

Службы

Не все на свете сводится к вещам и предметам.
Эрик Эванс

СЛУЖБА (SERVICE) в предметной области — это операция, не имеющая собственного состояния и выполняющая специфическое задание из предметной области. Часто верным признаком необходимости создания СЛУЖБЫ в предметной области является наличие операции, которую неудобно выполнять как метод **АГРЕГАТА (10)** или **ОБЪЕКТА-ЗНАЧЕНИЯ (6)**. Для того чтобы устранить этот дискомфорт, естественно было бы создать статический метод в классе КОРНЯ АГРЕГАТА. Однако в рамках подхода DDD это является признаком плохого кода, свидетельствующим о том, что вам необходима СЛУЖБА.

Назначение главы

- Показать, как уточнения модели предметной области могут привести к осознанию необходимости СЛУЖБЫ.
- Описать, чем является СЛУЖБА в предметной области и чем она не является.
- Сформулировать необходимые предостережения, которые необходимо учитывать при решении создавать или не создавать службу.
- Продемонстрировать СЛУЖБЫ в рамках модели предметной области на двух примерах из практики компании SaaSOvation.

Плохой код? Именно с этим явлением столкнулись разработчики из компании SaaSOvation в ходе рефакторинга АГРЕГАТА. Рассмотрим их тактические исправления. Вот как это было...

Ранее команда моделировала коллекцию объектов класса `BacklogItem` как составную часть АГРЕГАТА в классе `Product`. Это позволяло вычислять бизнес-приоритеты всех списокв заданий для спринта с помощью простого метода экземпляра класса `Product`.

```
public class Product extends ConcurrencySafeEntity {
    ...
    private Set<BacklogItem> backlogItems;
    ...
    public BusinessPriorityTotals businessPriorityTotals() {
    ...
   }
    ...
}
```

На этот раз проект оказался идеальным, потому что метод `businessPriorityTotals()` мог просто перебирать экземпляры класса `BacklogItem` и возвращать запрошенный общий бизнес-приоритет. Проект правильно давал ответ на запрос к ОБЪЕКТУ-ЗНАЧЕНИЮ, а именно — к классу `BusinessPriorityTotals`.

Однако такая ситуация не могла сохраняться вечно. Как показал анализ **АГРЕГАТОВ (10)**, большой кластер `Product` следовало разбить на части, а класс `BacklogItem` необходимо было перепроектировать, чтобы он сам стал АГРЕГАТОМ. Таким образом, предыдущий проект, использовавший метод экземпляра, следовало переделать.

Поскольку класс `Product` больше не содержал коллекцию экземпляров класса `BacklogItem`, первой реакцией команды было выполнить рефакторинг существующего метода экземпляра, чтобы он использовал новый класс `BacklogItemRepository` для выполнения всех вычислений, относящихся к экземплярам класса `BacklogItem`. Правильно ли это?

На самом деле руководитель убедил команду не делать этого. Как показывает опыт, по возможности следует избегать обращения к **ХРАНИЛИЩАМ (12)** изнутри АГРЕГАТОВ. Может быть, лучше просто создать такой же статический метод класса `Product` и передать ему коллекцию экземпляров класса `BacklogItem`, необходимую для выполнения вычислений? В этом случае метод остался бы практически неизменным, за исключением нового параметра.

```
    public class Product extends ConcurrencySafeEntity {
        ...
        public static BusinessPriorityTotals businessPriorityTotals(
                  Set<BacklogItem> aBacklogItems) {
                  ...
        }
        ...
}
```

Действительно ли класс `Product` представляет собой самое лучшее место для создания статического метода? На первый взгляд, довольно трудно понять, где должен находиться этот метод. Поскольку операция используется только для вычисления бизнес-приоритета каждого экземпляра класса `BacklogItem`, может быть, статический метод следует включить в этот класс. Впрочем, бизнес-приоритет скорее относится к продукции, а не к списку заданий для спринта. Непонятно...

В этот момент руководитель вдруг вскрикнул. Он понял, что весь этот дискомфорт можно устранить с помощью одного-единственного инструмента моделирования — СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICE). Как же это сделать?

Сначала изложим основы. Затем мы вернемся к ситуации, которая сложилась в ходе моделирования, и увидим, к какому выводу пришла команда.

Чем является служба предметной области (но сначала о том, чем она не является)

Когда мы слышим термин *служба* в контексте программного обеспечения, то, естественно, представляем себе крупномодульный компонент, позволяющий удаленному клиенту взаимодействовать со сложной информационной системой. Так в общих чертах определяется служба в **СЕРВИС-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ (SERVICE-ORIENTED ARCHITECTURE — SOA) (4)**. Существуют разные технологии и подходы к разработке служб SOA. Все виды таких служб в конце концов сводятся к подчеркиванию роли *вызова удаленных процедур на системном уровне* (remote procedure calls — RPCs) или *промежуточного программного обеспечения, ориентированного на обмен сообщениями* (message-oriented middleware — MoM), при этом другие системы в рамках информационного центра или в глобальном масштабе могут взаимодействовать со службой при выполнении своих бизнес-операций.

Ни одна из этих служб не является СЛУЖБОЙ ПРЕДМЕТНОЙ ОБЛАСТИ.

Кроме того, не следует путать СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ с **ПРИКЛАДНОЙ СЛУЖБОЙ (APPLICATION SERVICE)**. Мы хотим поместить бизнес-логику не в ПРИКЛАДНУЮ СЛУЖБУ, а в СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ. Если вы

не понимаете разницы между ними, сравните их с **ПРИЛОЖЕНИЕМ (14)**. Кратко говоря, для того чтобы различать эти службы, надо помнить, что ПРИКЛАДНАЯ СЛУЖБА — это естественный клиент модели предметной области, который, как правило, может быть также клиентом СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Мы покажем это чуть позже.

То, что в термин СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ входит слово *служба*, не значит, что она представляет собой крупномодульную масштабную транзакционную операцию, обеспечивающую удаленный доступ.[1]

Ковбойская логика

LB: Всегда внимательно смотри на то, что ты собираешься съесть. Не столь важно, чем оно является сейчас, но очень важно, чем оно было раньше.

Службы, которые принадлежат сугубо предметной области, представляют собой идеальное средство моделирования ситуаций, в которых ваши потребности пересекаются с их возможностями. Итак, теперь, когда вы знаете, чем *не* являются СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, пора разобраться в том, чем они *являются*.

> Не все на свете сводится к вещам и предметам... Если существенно важный процесс или преобразование в модели не относится к естественным обязанностям объекта СУЩНОСТИ или ЗНАЧЕНИЯ, добавьте в модель эту операцию с отдельным интерфейсом и назовите ее СЛУЖБОЙ. Определите интерфейс на языке модели и сделайте имя операции элементом ЕДИНОГО ЯЗЫКА. У СЛУЖБЫ не должно быть собственного состояния [Эванс, с. 106, 108].

Поскольку модель предметной области обычно оперирует мелкомодульными функциями, ориентированными на конкретные аспекты бизнеса, СЛУЖБЫ в предметной области часто основаны на тех же принципах. Поскольку они могут применять к многочисленным объектам предметной области отдельные атомарные операции, сложность системы может увеличиться.

При каких условиях операция может не принадлежать ни одной из существующих **СУЩНОСТЕЙ (5)** или ни одному ОБЪЕКТУ-ЗНАЧЕНИЮ? Перечислить исчерпывающий список таких ситуаций довольно трудно, поэтому мы приведем лишь

---

[1] Иногда СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ связана с удаленными вызовами внешнего ОГРАНИЧЕННОГО КОНТЕКСТА (2). Дело здесь в том, что СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ не предоставляет интерфейс для вызова удаленной процедуры, а сама является клиентом этого интерфейса.

некоторые из них. Службу предметной области следует применять в следующих ситуациях.

- Для выполнения важного бизнес-процесса.
- Для преобразования объекта предметной области путем изменения его состава.
- Для вычисления входной информации, требуемой ЗНАЧЕНИЕМ от нескольких объектов предметной области.

Вероятно, вычисление можно считать "важным процессом", но я указал его только для ясности. Эта операция является очень широко распространенной и требует на входе двух или, возможно, многих разных АГРЕГАТОВ или их составных частей. Если же она является слишком громоздкой для того, чтобы включить ее в СУЩНОСТЬ или ЗНАЧЕНИЕ, то лучше всего определить СЛУЖБУ. Убедитесь, что эта СЛУЖБА не имеет собственного *состояния* и имеет интерфейс, который четко выражает **ЕДИНЫЙ ЯЗЫК (1)** в своем ОГРАНИЧЕННОМ КОНТЕКСТЕ.

Убедитесь, что вам необходима СЛУЖБА

Не торопитесь моделировать концепцию предметной области в виде СЛУЖБЫ. Так можно делать только под давлением обстоятельств. Будьте осторожны, потому что вам может показаться, что моделирование службы — это универсальная волшебная палочка. Чрезмерное увлечение СЛУЖБАМИ обычно приводит к негативным последствиям и созданию **АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ** [Fowler, Anemic], в которой вся логика предметной области заключена в службах, а не распределена по СУЩНОСТЯМ и ОБЪЕКТАМ-ЗНАЧЕНИЯМ. Следующий анализ показывает важность тщательного обдумывания тактики в ходе анализа каждой моделируемой ситуации. Следуя этим советам, вы будете принимать правильные решения о том, следует ли моделировать СЛУЖБУ.

Рассмотрим пример, позволяющий распознать потребность в моделировании СЛУЖБЫ. Представьте себе, что вы пытаетесь авторизовать экземпляр класса `User` в нашем *Контексте идентификации и доступа*.

Напомним, что в главе, посвященной **СУЩНОСТЯМ (5)**, мы рассматривали сценарий, в котором команда отложила отказ в идентификации на более поздний момент. Ну вот, этот момент наступил.

- Пользователи системы должны быть авторизованы, но могут авторизоваться, только если арендатор является активным.

Посмотрим, почему в этой ситуации необходима СЛУЖБА. Нельзя ли просто включить эту функцию в СУЩНОСТЬ? С точки зрения клиента мы могли бы смоделировать авторизацию следующим образом.

```
// Клиент находит экземпляр класса User и просит его авторизоваться

boolean authentic = false;

User user =
     DomainRegistry
     .userRepository()
     .userWithUsername(aTenantId, aUsername);

if (user != null) {
    authentic = user.isAuthentic(aPassword);
}

return authentic;
```

Я думаю, что в этой схеме возникает сразу несколько проблем. Мы требуем от клиентов, чтобы они понимали, в чем заключается авторизация. Они должны найти экземпляр класса `User`, а затем спросить его, совпадает ли введенный пароль с тем, что хранится у него. Кроме того, здесь нет явного моделирования ЕДИНОГО ЯЗЫКА. Мы спрашиваем экземпляр класса `User`, "является ли он авторизованным", вместо того чтобы приказать модели "выполнить авторизацию". По возможности следует стремиться выражать модель с помощью естественных выражений, которые используются командами, а не заставлять команду подстраивать свою точку зрения под неестественные правила лишь потому, что нам не удалось найти лучшую модель концепции. Впрочем, здесь есть проблемы и посерьезнее.

Именно эту неправильную модель создали члены команды, пытаясь описать процедуру авторизации пользователя. Бросается в глаза, что в этой модели нет проверки активности арендатора. В соответствии с условиями, если арендатор, к которому относится пользователь, не является активным, то пользователь не авторизуется. Возможно, проблему можно было бы решить следующим образом.

```
// Может быть, лучше так...
boolean authentic = false;

Tenant tenant =
    DomainRegistry
    .tenantRepository()
    .tenantOfId(aTenantId);

    if (tenant != null && tenant.isActive()) {
        User user =
```

```
            DomainRegistry
            .userRepository()
            .userWithUsername(aTenantId, aUsername);

        if (user != null) {
            authentic = tenant.authenticate(user, aPassword)
        }
    }

return authentic;
```

Перед авторизацией этот тест правильно распознает, что экземпляр класса `Tenant` активен. Кроме того, можно было освободить класс `User` от метода `isAuthentic()`, поместив в класс `Tenant` метод `authenticate()`.

Но и это решение не устранило все проблемы. Теперь на клиента возложены дополнительные обязанности. Он должен глубже понимать процесс авторизации. Это требование можно было бы смягчить, проверяя активность экземпляра класса `Tenant` внутри метода `authenticate()`, вызывая метод `isActive()`, но в этом случае мы не создали бы явной модели. Кроме того, это породило бы новую проблему. Теперь арендатор должен понимать, что делать с паролем. Напомним, что команда сформулировала, но не включила в сценарий авторизации еще одно требование.

- Пароли должны храниться в зашифрованном, а не открытом виде.

Предложенные выше решения усложняют модель. Приняв последнее предложение, мы должны выбрать один из четырех нежелательных подходов.

1. Класс `Tenant` шифрует пароль и передает его в класс `User`. Это нарушает принцип **ЕДИНСТВЕННОЙ ОБЯЗАННОСТИ (SINGLE RESPONSIBILITY)** [Martin, SRP], используемый только для моделирования арендаторов.
2. Возможно, в класс `User` следует включить немного информации о шифровании, поскольку он обязан гарантировать, что все пароли хранятся в зашифрованном виде. В этом случае необходимо включить в класс `User` метод, знающий, как авторизовать пользователя по заданному открытому паролю. Правда, в этом случае авторизация становится фасадом класса `Tenant` и полностью реализуется только классом `User`. Более того, класс `User` должен иметь защищенный интерфейс авторизации, чтобы предотвратить его непосредственное использование внешними клиентами модели.
3. Класс `Tenant` просит класс `User` зашифровать открытый пароль, а затем сравнивает его с одним из паролей, хранящихся в классе `User`. Эти действия кажутся лишними в запутанной процедуре взаимодействия между этими классами. Класс `Tenant` по-прежнему должен знать детали авторизации, даже несмотря на то, что он ее не осуществляет.

4. Клиент шифрует пароль и передает его в класс Tenant. Это возлагает на клиента дополнительную обязанность, в то время как клиент не должен вообще ничего знать о необходимости шифровать пароли.

Ни одно из этих предложений не подходит, и клиент остается чрезмерно сложным. Обязанность, которую мы возложили на клиента, необходимо удалить из модели. Знание, относящееся исключительно к предметной области, никогда не должно уходить к клиентам. Даже если клиентом является ПРИКЛАДНАЯ СЛУЖБА, этот компонент не обязан управлять процессом идентификации и доступа.

Ковбойская логика

AJ: Если ты оказался в яме, прежде всего прекрати копать.

Действительно, единственной бизнес-обязанностью, которую должен иметь клиент, является координация использования отдельной предметно-ориентированной операции с другими деталями бизнес-проблемы.

```
// Клиент внутри ПРИКЛАДНОЙ СЛУЖБЫ
// с единственной обязанностью координации

UserDescriptor userDescriptor =
    DomainRegistry
       .authenticationService()
       .authenticate(aTenantId, aUsername, aPassword);
```

В этом простом и элегантном решении клиент обязан лишь получить ссылку на экземпляр класса AuthenticationService, не имеющего собственного состояния, а затем вызвать из него метод authenticate(). В этом случае все детали, связанные с авторизацией, переносятся из клиента ПРИКЛАДНОЙ СЛУЖБЫ в СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ. При необходимости СЛУЖБА может использовать любое количество объектов предметной области. При этом гарантируется, что шифрование пароля выполняется правильно. Клиент не обязан знать все детали шифрования. Кроме того, выполняются все правила ЕДИНОГО ЯЗЫКА в КОНТЕКСТЕ, потому что его термины полностью выражаются в программном обеспечении, моделирующем предметную область управления идентификацией, а не частично моделью и частично клиентом.

Метод СЛУЖБЫ возвращает ОБЪЕКТ-ЗНАЧЕНИЕ класса UserDescriptor. Это небольшой и безопасный объект. В отличие от полного класса User, он включает в себя лишь несколько атрибутов, относящихся к ссылкам на класс User.

```
public class UserDescriptor implements Serializable {
    private String emailAddress;
    private TenantId tenantId;
    private String username;

    public UserDescriptor(
            TenantId aTenantId,
            String aUsername,
            String anEmailAddress) {
        ...
    }
    ...
}
```

Этот класс подходит для организации пользовательского веб-сеанса. Клиент прикладной службы сам может возвращать этот объект в вызывающий объект или создавать более удобный объект.

Моделирование службы в предметной области

В зависимости от предназначения СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ иногда ее довольно просто моделировать. Сначала необходимо решить, должна ли СЛУЖБА иметь **ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС (SEPARATED INTERFACE)** [Fowler, P of EAA]. В этом случае определение интерфейса может выглядеть следующим образом.

```
package com.saasovation.identityaccess.domain.model.identity;

public interface AuthenticationService {

public UserDescriptor authenticate(
        TenantId aTenantId,
        String aUsername,
        String aPassword);
}
```

Интерфейс объявлен в том же самом **МОДУЛЕ (9)**, что и связанные с ним АГРЕГАТЫ, такие как `Tenant`, `User` и `Group`. Это сделано потому, что класс `AuthenticationService` описывает концепцию авторизации и мы в настоящий момент разместили все концепции, связанные с идентификацией, в МОДУЛЕ `identity`. Определение интерфейса выглядит довольно просто. В нем необходима только одна операция — `authenticate()`.

Необходимо выбрать место для класса реализации. Если мы используем **ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТИ (4)** или **ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ (4)**, то можно разместить класс технической реализации за пределами модели

предметной области. Например, техническую реализацию можно включить в МОДУЛЬ на УРОВНЕ ИНФРАСТРУКТУРЫ.

Приведем описание класса.

```
package com.saasovation.identityaccess.infrastructure.services;

import com.saasovation.identityaccess.domain.model.DomainRegistry;
import com.saasovation.identityaccess.domain.model.identity..
AuthenticationService;
import com.saasovation.identityaccess.domain.model.identity.Tenant;
import com.saasovation.identityaccess.domain.model.identity.TenantId;
import com.saasovation.identityaccess.domain.model..
identity.User;
import com.saasovation.identityaccess.domain.model..
identity.UserDescriptor;

public class DefaultEncryptionAuthenticationService
       implements AuthenticationService {

    public DefaultEncryptionAuthenticationService() {
        super();
    }

    @Override
    public UserDescriptor authenticate(
           TenantId aTenantId,
           String aUsername,
           String aPassword) {

       if (aTenantId == null) {
           throw new IllegalArgumentException(
                   "TenantId must not be null.");
       }
       if (aUsername == null) {
           throw new IllegalArgumentException(
                   "Username must not be null.");
       }
       if (aPassword == null) {
           throw new IllegalArgumentException(
                   "Password must not be null.");
       }

       UserDescriptor userDescriptor = null;

       Tenant tenant =
           DomainRegistry
           .tenantRepository()
           .tenantOfId(aTenantId);

       if (tenant != null && tenant.isActive()) {
           String encryptedPassword =
               DomainRegistry
               .encryptionService()
               .encryptedValue(aPassword);
```

```
            User user =
                DomainRegistry
                .userRepository()
                .userFromAuthenticCredentials(
                     aTenantId,
                     aUsername,
                     encryptedPassword);

            if (user != null && user.isEnabled()) {
                userDescriptor = user.userDescriptor();
            }
        }

        return userDescriptor;
    }
}
```

Этот метод защищает от нулевых параметров. В противном случае, если процесс авторизации в нормальных условиях завершается сбоем, возвращаемый объект `UserDescriptor` будет равен `null`.

Для авторизации мы начинаем попытку извлечь экземпляр класса `Tenant` из ХРАНИЛИЩА с помощью его идентификатора. Если экземпляр класса `Tenant` существует и активен, мы шифруем открытый пароль. Мы делаем это сейчас, потому что будем использовать зашифрованный пароль для извлечения экземпляра класса `User`. Вместо запроса экземпляра класса `User` только по идентификатору `TenantId` и сравнения с именем `username` мы сравниваем зашифрованный пароль. (Результаты шифрования двух одинаковых открытых паролей всегда совпадают.) ХРАНИЛИЩЕ предназначено для фильтрации всех трех атрибутов.

Если пользователь-человек указал правильный идентификатор арендатора, имя пользователя и открытый пароль, результатом станет извлеченный экземпляр класса `User` с данными атрибутами. Однако этого недостаточно для авторизации пользователя, потому что все еще не выполнено последнее требование.

- Пользователи могут быть авторизованы, только если они являются допустимыми.

Даже если ХРАНИЛИЩЕ нашло отфильтрованный экземпляр класса `User`, он может оказаться недопустимым. Возможность блокировки экземпляра класса `User` позволяет арендатору контролировать авторизацию на разных уровнях. Таким образом, на последнем этапе экземпляр класса `User` одновременно не должен быть равным `null` и должен быть допустимым, поэтому класс `UserDescriptor` является производным от класса `User`.

Необходим ли ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС

Поскольку класс AuthenticationService не имеет технической реализации, не следует ли создать ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС и класс реализации в выделенных УРОВНЯХ и МОДУЛЯХ? Нет, на самом деле это совершенно необязательно. Мы могли бы создать эту конкретную СЛУЖБУ только с одним классом реализации, имя которого совпадает с именем СЛУЖБЫ.

```
package com.saasovation.identityaccess.domain.model.identity;

public class AuthenticationService {

public AuthenticationService() {
    super();
}

public UserDescriptor authenticate(
        TenantId aTenantId,
        String aUsername,
        String aPassword) {
        ...
    }
}
```

В этом нет ничего плохого. Вы даже можете прийти к выводу, что этот подход более удобен, потому что для данной конкретной СЛУЖБЫ может никогда не понадобиться несколько реализаций. Однако если разные арендаторы примут разные стандарты безопасности, им потребуются разные реализации. Впрочем, пока команда решила отказаться от ВЫДЕЛЕННОГО ИНТЕРФЕЙСА и остановиться на приведенном выше классе.

Как назвать класс реализации

В среде программистов на языке Java принято перед именем класса реализации ставить в качестве префикса имя его интерфейса, а в качестве окончания указывать слово Impl. Если бы мы последовали этому правилу, то в нашем примере класс имел бы имя AuthenticationServiceImpl. Кроме того, интерфейс и класс реализации часто находятся в одном и том же пакете. Хорошо ли это?

Действительно, если ваш класс реализации называется именно так, то такое имя, вероятно, является верным признаком того, что вам не нужен выделенный интерфейс или что вам следует внимательно подумать об имени класса реализации. Итак, имя AuthenticationServiceImpl на самом деле выбрано неудачно. Впрочем, имя DefaultEncryptionAuthenticationService тоже выглядит плохо. По этой причине разработчики компании SaaSOvation решили пока исключить ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС и использовать простой класс AuthenticationService.

Если ваш класс реализации предназначен для разукрупнения, потому что вам нужно много разных реализаций, называйте его в соответствии с его специализацией. Необходимость тщательно выбирать имя для каждой специализированной реализации является свидетельством того, что эти специализации существуют в вашей предметной области.

Одни читатели могут прийти к выводу, что классы интерфейса и классы реализации с похожими именами упрощают просмотр и обход больших пакетов, составленных из таких пар. Однако другие читатели могут назвать такие крупные пакеты плохо спроектированными и не соответствующими целям шаблона МОДУЛЬ. Кроме того, специалисты, стремящиеся к модульности, предпочитают размещать классы интерфейса и разные классы реализации в разных пакетах, как это принято в соответствии с **ПРИНЦИПОМ ИНВЕРСИИ ЗАВИСИМОСТИ (4)**. Например, интерфейс `EncryptionService` находится в модели предметной области, а класс `MD5EncryptionService` входит в инфраструктуру.

Исключение ВЫДЕЛЕННОГО ИНТЕРФЕЙСА из СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ, не имеющих технического предназначения, не уменьшает возможности тестирования, поскольку все интерфейсы, от которых зависит СЛУЖБА, можно при необходимости внедрить с помощью ФАБРИКИ СЛУЖБ, настроенной на тестирование, и передать в качестве параметров входящих и исходящих зависимостей. Напомним также, что корректность служб, не имеющих технического характера и относящихся к предметной области, например предназначенных для вычислений, должна обязательно проверяться.

Очевидно, что это противоречивая тема, и я понимаю, что существует много специалистов, которые регулярно называют реализации интерфейса с помощью окончания `Impl`. Просто надо понимать, что всегда найдутся хорошо информированные оппоненты, которые найдут причины отвергнуть этот подход. Как всегда, выбор за вами!

Использование ВЫДЕЛЕННОГО ИНТЕРФЕЙСА может оказаться в большей степени делом вкуса в ситуациях, в которых СЛУЖБА всегда относится к предметной области и никогда не имеет технической реализации или несколько реализаций. Как утверждает Фаулер [Fowler, P of EAA], ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС полезен, если он используется для разукрупнения: "Клиент, которому необходима зависимость от интерфейса, может ничего не знать о реализации". Однако, если мы используем **ВНЕДРЕНИЕ ЗАВИСИМОСТИ (DEPENDENCY INJECTION)** или **ФАБРИКУ (FACTORY)** [Гамма и др.] СЛУЖБ, даже если интерфейс службы и класс объединены, клиент всегда можно изолировать от реализации.

Иначе говоря, при следующем использовании класса `DomainRegistry` как ФАБРИКИ СЛУЖБ клиент отделяется от реализации.

```
// Этот реестр позволяет отделить клиента от реализации

UserDescriptor userDescriptor =
    DomainRegistry
        .authenticationService()
        .authenticate(aTenantId, aUsername, aPassword);
```

Аналогичных результатов можно достичь при использовании шаблона ВНЕДРЕНИЕ ЗАВИСИМОСТИ.

```
public class SomeApplicationService ... {
    @Autowired
    private AuthenticationService authenticationService;
    ...
}
```

Контейнер, используемый для инверсии зависимости (например, Spring), внедряет экземпляр СЛУЖБЫ. Поскольку клиент никогда не создает экземпляр СЛУЖБЫ, он не знает, объединены или разъединены интерфейс и реализация.

Очевидно, что некоторые специалисты могут пренебречь ФАБРИКОЙ СЛУЖБ и ВНЕДРЕНИЕМ ЗАВИСИМОСТИ и отдать предпочтение установлению исходящих зависимостей с помощью конструктора или передать их в качестве параметров метода. Это более ясный способ установления зависимостей, который упрощает тестирование, и более простой, чем ВНЕДРЕНИЕ ЗАВИСИМОСТИ. Некоторые специалисты видят выгоду в использовании комбинации всех трех шаблонов в зависимости от ситуации, причем зависимости при этом устанавливаются с помощью конструктора. В некоторых примерах из этой главы для ясности используется класс `DomainRegistry`, *хотя это ни о чем не говорит*. Многие примеры исходного кода из книги, размещенные в сети, демонстрируют установление зависимостей с помощью конструкторов или путем непосредственной передачи параметров соответствующим методам.

Процесс вычисления

Рассмотрим другой пример, на этот раз из текущего **СМЫСЛОВОГО ЯДРА (2)**, т.е. *Контекста управления гибким проектированием*. Эта СЛУЖБА вычисляет результат по ЗНАЧЕНИЯМ, полученным из любого количества АГРЕГАТОВ конкретного типа. Мне кажется, что использовать здесь ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС нецелесообразно, по крайней мере пока. Вычисления всегда выполняются одинаково. Если ситуация не изменится, мы не должны беспокоиться об отделении интерфейса от реализации.

Ковбойская логика

LB: Мой жеребец приносит мне пять тысяч долларов за каждую случку[2], и кобылы выстраиваются к нему в очередь.

AJ: Ну, хоть *этот* конь занимается своим делом.[3]

Напомним, что разработчики компании SaaSOvation разработали мелкомодульные статические методы в классе `Product` для выполнения требуемых вычислений. Вот что произошло потом...

Руководитель указал на желательность использования СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ вместо статического метода. Идея, лежащая в основе СЛУЖБЫ, очень похожа на то, что разработчики уже придумали: вычислить и вернуть экземпляр ЗНАЧЕНИЯ класса `BusinessPriorityTotals`. Однако СЛУЖБА должна делать немного больше. Она должна найти все *отложенные* задачи для данного продукта Scrum, а затем суммировать все их индивидуальные ЗНАЧЕНИЯ класса `BusinessPriority`. Вот как выглядит реализация этой СЛУЖБЫ.

```
package com.saasovation.agilepm.domain.model.product;

import com.saasovation.agilepm.domain.model.DomainRegistry;
import com.saasovation.agilepm.domain.model.tenant.Tenant;

public class BusinessPriorityCalculator {

    public BusinessPriorityCalculator() {
       super();
    }

    public BusinessPriorityTotals businessPriorityTotals(
            Tenant aTenant,
            ProductId aProductId) {
       int totalBenefit = 0;
       int totalPenalty = 0;
       int totalCost = 0;
       int totalRisk = 0;

       java.util.Collection<BacklogItem> outstandingBacklogItems =
           DomainRegistry
               .backlogItemRepository()
               .allOutstandingProductBacklogItems(
```

[2] Игра слов: *service* — это и "служба", и "случка". — *Примеч. ред.*

[3] Игра слов: *be in own domain* — "заниматься своим делом". — *Примеч. ред.*

```
                aTenant,
                aProductId);

        for (BacklogItem backlogItem : outstandingBacklogItems) {
             if (backlogItem.hasBusinessPriority()) {
                 BusinessPriorityRatings ratings =
                    backlogItem.businessPriority().ratings();

                 totalBenefit += ratings.benefit();
                 totalPenalty += ratings.penalty();
                 totalCost += ratings.cost();
                 totalRisk += ratings.risk();
             }
        }

        BusinessPriorityTotals businessPriorityTotals =
            new BusinessPriorityTotals(
                     totalBenefit,
                     totalPenalty,
                     totalBenefit + totalPenalty,
                     totalCost,
                     totalRisk);

        return businessPriorityTotals;
    }
}
```

Класс `BacklogItemRepository` используется для получения всех *отложенных* экземпляров класса `BacklogItem`. Отложенный экземпляр класса `BacklogItem` может иметь состояние *Запланированный*, *Включенный в расписание* или *Назначенный*, но не *Сделанный* или *Удаленный*. СЛУЖБА в предметной области при необходимости может использовать ХРАНИЛИЩЕ, но обращение к ХРАНИЛИЩАМ из экземпляра АГРЕГАТА, как правило, не рекомендуется.

Мы перебираем все отложенные элементы для заданного продукта и суммируем их поля `ratings` из класса `BusinessPriority`. Вычисленная сумма используется для создания нового экземпляра класса `BusinessPriorityTotals`, который возвращается клиенту. Нет никакой необходимости усложнять процесс вычисления, выполняемый СЛУЖБОЙ, хотя иногда это необходимо. На этот раз все выглядит довольно просто.

Отметим, что в этом примере мы *совершенно не хотели* включать эту логику в ПРИКЛАДНУЮ СЛУЖБУ. Даже если бы мы считали суммирование рейтингов с помощью цикла `for` тривиальной задачей, этот процесс относится к бизнес-логике. Приведем еще одну причину.

```
BusinessPriorityTotals businessPriorityTotals =
    new BusinessPriorityTotals(
            totalBenefit,
            totalPenalty,
```

```
            totalBenefit + totalPenalty,
            totalCost,
            totalRisk);
```

После создания экземпляра BusinessPriorityTotals его атрибут totalValue выводится путем суммирования атрибутов totalBenefit и totalPenalty. Эта логика определяется предметной областью и не должна проникать на ПРИКЛАДНОЙ УРОВЕНЬ. Можно было бы предложить, чтобы конструктор класса BusinessPriorityTotals сам решал эту задачу, суммируя два переданных ему параметра. Несмотря на то что это могло бы улучшить модель, все же это слабое обоснование, чтобы переносить остальные вычисления в ПРИКЛАДНУЮ СЛУЖБУ.

Хотя мы не включаем эту бизнес-логику в ПРИКЛАДНУЮ СЛУЖБУ, она сама является клиентом СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ.

```
public class ProductService ... {
    ...
    private BusinessPriorityTotals productBusinessPriority(
            String aTenantId,
            String aProductId) {
        BusinessPriorityTotals productBusinessPriority =
            DomainRegistry
                .businessPriorityCalculator()
                .businessPriorityTotals(
                      new TenantId(aTenantId),
                      new ProductId(aProductId));

        return productBusinessPriority;
    }
}
```

В этом случае закрытый метод в ПРИКЛАДНОЙ СЛУЖБЕ запрашивает общий бизнес-приоритет продукта. Этот метод может возвращать только часть полезных данных, возвращаемых клиенту класса ProductService, например пользовательскому интерфейсу.

Службы преобразования

Реализации СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ, имеющие более технический характер и поэтому относящиеся к ИНФРАСТРУКТУРЕ, часто используются для интеграции. По этой причине я включил соответствующие примеры в главу, посвященную **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (INTEGRATING BOUNDED CONTEXTS) (13)**. Там можно найти интерфейсы СЛУЖБЫ, классы реализации, а также **АДАПТЕРЫ (ADAPTERS)** [Гамма и др.] и трансляторы, используемые реализациями.

Использование микроуровней служб предметной области

Иногда желательно создать "микроуровень" СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ над остальными СУЩНОСТЯМИ И ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ, входящими в модель предметной области. Как указывалось ранее, часто это приводит к ненадежной АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ, которая считается антишаблоном.

Впрочем, существуют системы, в которых проектирование микроуровней СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ имеет больше смысла, чем в других, и при этом не приводит к появлению АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ. Это зависит от характеристик модели предметной области. В данном случае *Контекст идентификации и доступа* оказывается действительно полезным.

Если у вас есть опыт работы в таких предметных областях и вы решили разработать микроуровень СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ, помните, что эти службы отличаются от ПРИКЛАДНЫХ СЛУЖБ на ПРИКЛАДНОМ УРОВНЕ. Вопросы выполнения транзакций и проблемы безопасности должны решаться ПРИКЛАДНЫМИ СЛУЖБАМИ, а не СЛУЖБАМИ ПРЕДМЕТНОЙ ОБЛАСТИ.

Службы тестирования

Мы хотим тестировать наши СЛУЖБЫ, чтобы убедиться, что мы правильно отразили точку зрения клиента, которую должны были смоделировать. Мы хотим, чтобы предметно-ориентированные тесты отражали способ использования модели, а не проверяли корректность программного обеспечения.

> **Не слишком ли поздно для тестирования?**
>
> Обычно я провожу тестирование до реализации. Я пока не приводил фрагменты тестов в процессе анализа СЛУЖБЫ. Это объясняется тем, что я считал более естественным обсудить вопросы реализации чуть раньше, вот и все. Однако это не значит, что я считаю раннее тестирование совершенно излишним, хотя оно и может немного смещать фокус внимания в процессе моделирования.

Эти тесты демонстрируют, как правильно использовать класс `AuthenticationService`. Сначала мы тестируем сценарий авторизации.

```
public class AuthenticationServiceTest
       extends IdentityTest {

    public void testAuthenticationSuccess() throws Exception {

       User user = this.getUserFixture();

       DomainRegistry
            .userRepository()
            .add(user);
```

```
        UserDescriptor userDescriptor =
             DomainRegistry
                 .authenticationService()
                 .authenticate(
                     user.tenantId(),
                     user.username(),
                     FIXTURE_PASSWORD);

        assertNotNull(userDescriptor);
        assertEquals(user.tenantId(), userDescriptor.tenantId());
        assertEquals(user.username(), userDescriptor.username());
        assertEquals(user.person().emailAddress(),
                     userDescriptor.emailAddress());
    }
 ...
}
```

Этот пример показывает, как клиент ПРИКЛАДНОЙ СЛУЖБЫ должен использовать класс `AuthenticationService`. Он демонстрирует успешную авторизацию пользователя с помощью передачи ожидаемых параметров.

Отметим, что ХРАНИЛИЩЕ может быть полностью реализовано, находиться в оперативной памяти или имитироваться. Если полная реализация является достаточно быстрой, тестирование не вызывает никаких проблем, поскольку тест заканчивается откатом транзакции, который предотвращает передачу излишних экземпляров между тестами. Выбор вида ХРАНИЛИЩА для тестирования — дело вашего вкуса.

Теперь продемонстрируем сценарий, в котором авторизация завершается провалом.

```
public void testAuthenticationTenantFailure() throws Exception {

    User user = this.getUserFixture();

    DomainRegistry
        .userRepository()
        .add(user);

    TenantId bogusTenantId =
         DomainRegistry.tenantRepository().nextIdentity();

    UserDescriptor userDescriptor =
         DomainRegistry
             .authenticationService()
             .authenticate(
             bogusTenantId, // bogus
             user.username(),
             FIXTURE_PASSWORD);

    assertNull(userDescriptor);
}
```

Этот тест авторизации провален, потому что мы преднамеренно передали экземпляр класса `TenantId`, который отличается от созданного в классе `User`. Теперь покажем, как проверяется ввод неправильного имени пользователя.

```
public void testAuthenticationUsernameFailure() throws Exception {

    User user = this.getUserFixture();

    DomainRegistry
         .userRepository()
         .add(user);

    UserDescriptor userDescriptor =
          DomainRegistry
               .authenticationService()
               .authenticate(
                     user.tenantId(),
                     "bogususername",
                     user.password());

    assertNull(userDescriptor);
}
```

Этот сценарий авторизации заканчивается провалом, потому что мы передаем неправильное имя пользователя. Покажем последний сценарий, который также проваливает тесты.

```
public void testAuthenticationPasswordFailure() throws Exception {

User user = this.getUserFixture();

     DomainRegistry
          .userRepository()
          .add(user);

     UserDescriptor userDescriptor =
          DomainRegistry
               .authenticationService()
               .authenticate(
               user.tenantId(),
               user.username(),
               "passw0rd");

      assertNull(userDescriptor);
    }
}
```

В этом сценарии передается неправильный пароль, который вызывает отказ. Во всех провальных сценариях экземпляр класса `UserDescriptor` возвращался как значение `null`. Клиенты должны знать об этой детали, которая

свидетельствует об отказе в авторизации пользователя. Она также свидетельствует о том, что безуспешная авторизация не является ошибкой, порождающей исключение, а представляет собой обычное явление в данной предметной области. Если бы это было иначе, то СЛУЖБА должна была бы создавать исключение `AuthenticationFailedException`.

Мы не привели всего несколько тестов. Я оставил читателям в качестве домашнего задания сценарии тестирования, в которых экземпляр класса `Tenant` не активен, а экземпляр класса `User` является заблокированным. Помимо прочего, вы можете сами создать тесты для проверки класса `BusinessPriorityCalculator`.

Резюме

В этой главе мы рассмотрели, что собой представляет СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ и чего она собой не представляет, а также проанализировали, когда следует использовать СЛУЖБУ, а когда операцию над СУЩНОСТЬЮ или ОБЪЕКТОМ-ЗНАЧЕНИЕМ. Кроме того, мы достигли следующих результатов.

- Вы научились распознавать условия, в которых СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ действительно нужна, чтобы избегать злоупотребления СЛУЖБАМИ.
- Вы поняли, что злоупотребление СЛУЖБАМИ ПРЕДМЕТНОЙ ОБЛАСТИ приводит к появлению АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ, т.е. к антишаблону.
- Вы увидели конкретные этапы реализации СЛУЖБЫ.
- Вы узнали преимущества и недостатки использования ВЫДЕЛЕННОГО ИНТЕРФЕЙСА.
- Вы рассмотрели процесс вычислений из *Контекста управления гибким проектированием*.
- В заключение вы рассмотрели тесты, демонстрирующие использование служб в нашей модели.

Далее мы перейдем к современным средствам тактического моделирования на основе принципов DDD, в частности — к мощному стандартному проектному шаблону СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN EVENT).

Глава 8

События предметной области

История — это лишь версия случившихся событий
в нашей интерпретации.

Наполеон Бонапарт

Для фиксации того, что случилось в предметной области, используется шаблон **СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ**. Это чрезвычайно мощный инструмент моделирования. Научившись использовать СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, вы станете их приверженцем и будете удивляться, как могли обходиться без них раньше. Для начала необходимо понять, что представляют собой СОБЫТИЯ.

Назначение главы

- Узнать, что такое СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, а также когда и почему их следует использовать.
- Научиться моделировать СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ в виде объектов и распознавать ситуации, в которых они должны иметь уникальную идентичность.
- Изучить шаблон **ИЗДАТЕЛЬ-ПОДПИСЧИК (PUBLISH-SUBSCRIBE)** [Гамма и др.] и понять, как он позволяет уведомлять клиентов о событиях.
- Узнать, какие компоненты публикуют СОБЫТИЯ, а какие являются подписчиками.
- Изучить ситуации, в которых могут понадобиться ХРАНИЛИЩА СОБЫТИЙ, узнать, как они устроены и используются.
- На примере компании SaaSOvation изучить разные способы публикации событий для автономных систем.

Когда и почему происходят СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ

В книге [Эванс] вы не найдете формального определения СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ. Этот шаблон появился уже после опубликования книги. Прежде чем начать обсуждение реализации СОБЫТИЙ, происходящих в **ПРЕДМЕТНОЙ ОБЛАСТИ (2)**, рассмотрим современное определение.

> Факт, имеющий значение для экспертов предметной области.
>
> Моделируйте информацию о деятельности в предметной области как последовательность отдельных событий. Представляйте каждое событие как объект предметной области... Событие предметной области — это полноценная часть модели предметной области, представление факта, имеющего место в этой предметной области [Evans, с. 20].

Как определить, произошло ли нечто важное для экспертов в предметной области? Обсуждая с ними проект, следует внимательно следить за ходом их мысли. Рассмотрим несколько ключевых фраз, которые можно услышать от экспертов в предметной области.

- "Когда..."
- "Если это случится..."
- "Сообщите мне, если..." или "Уведомьте меня, если..."
- "В случае..."

Конечно, выражения "Сообщите мне, если..." и "Уведомьте меня, если..." не являются извещениями, образующими СОБЫТИЕ. Это просто высказывание о факте в предметной области, о котором *надо сообщить*, потому что он важен, и который, *возможно, свидетельствует* о необходимости моделировать явное событие. Кроме того, эксперты в предметной области могут произносить такие фразы: "Если *это* произойдет, то *это* не важно, а вот если *то* произойдет, то *это* важно". (Можете заменить частицы *это* и *то* на любое осмысленное событие в вашей предметной области.) В зависимости от уровня культуры организации фразы-триггеры могут изменяться.

Ковбойская логика

AJ: Если я хочу позвать своего коня, я просто кричу "Триггер, ко мне!", и он скачет ко мне. Конечно, при этом никогда не помешает показать ему кусочек сахара.

Иногда разговорный язык экспертов не позволяет понять, следует ли моделировать СОБЫТИЕ, хотя бизнес-ситуация может требовать этого. Эксперты в предметной области могут знать, а могут и не знать об этих требованиях. Это выясняется только в совместных обсуждениях. Часто возникают ситуации, в которых СОБЫТИЕ передается во внешние службы, где системы вашего предприятия отделены друг от друга, и сведения о событии должны пересекать границы **ОГРАНИЧЕННЫХ КОНТЕКСТОВ (2)**. В этом случае говорят о публикации событий и уведомлении подписчиков. Поскольку СОБЫТИЯ обрабатываются подписчиками, они могут оказывать сильное влияние как на локальные, так и на удаленные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ.

Эксперты в предметной области и события

Несмотря на то что эксперты в предметной области изначально могут не знать о потребности во всех типах СОБЫТИЙ, они должны понимать причины, по которым они принимают участие в обсуждении конкретных СОБЫТИЙ. Если достигнут консенсус, новые СОБЫТИЯ становятся формальной частью **ЕДИНОГО ЯЗЫКА (1)**.

Когда события достигают заинтересованных сторон в локальных или внешних системах, они обычно используются для достижения итоговой согласованности. Это их цель и предназначение. Они позволяют избежать двухфазной системы фиксации глобальных транзакций и соответствуют правилам работы с **АГРЕГАТАМИ (10)**. Одно из правил работы с АГРЕГАТАМИ утверждает, что за одну транзакцию должен изменяться только один экземпляр, а все остальные связанные с этим изменения должны выполняться в других транзакциях. Этот подход позволяет синхронизировать другие экземпляры АГРЕГАТА в локальном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Кроме того, это позволяет согласовать периоды ожидания удаленных зависимостей. Декомпозиция позволяет обеспечить высокую масштабируемость и наибольшую производительность взаимодействующих служб. Кроме того, она помогает достичь слабой связанности между системами.

На рис. 8.1 показано, как могут происходить СОБЫТИЯ, как их можно хранить и пересылать, а также где их можно использовать. События могут обрабатываться в локальных или внешних ОГРАНИЧЕННЫХ КОНТЕКСТАХ.

Кроме того, следует подумать о ситуациях, в которых ваши системы обычно выполняют пакетную обработку. Возможно, что в моменты минимальной нагрузки (вероятно, ночью) ваши системы выполняют определенную вспомогательную работу, удаляя устаревшие объекты, создавая новые, согласовывая разные объекты и даже уведомляя некоторых пользователей о важных событиях. Часто такая пакетная обработка требует последовательного выполнения сложных запросов, чтобы выделить бизнес-ситуации, требующие особого внимания. Вычисления и процедуры пакетной обработки требуют больших затрат, а синхронизация всех изменений требует выполнения крупных транзакций. Как бы избавиться от этих утомительных процедур?

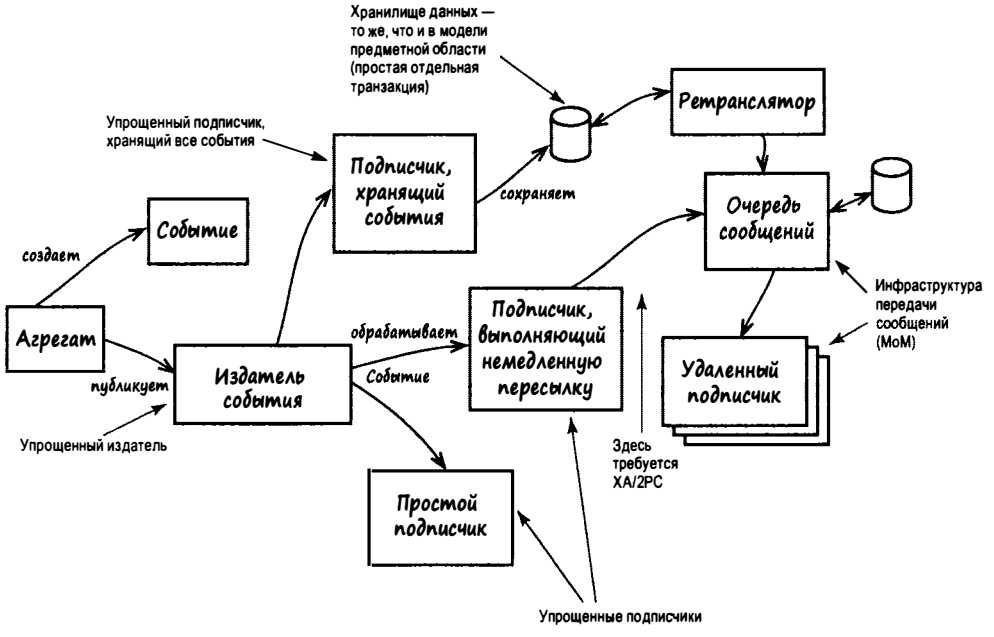


Рис. 8.1. АГРЕГАТЫ создают и публикуют СОБЫТИЯ. Подписчики могут сохранять СОБЫТИЯ, а затем пересылать их удаленным подписчикам или просто пересылать их, не сохраняя. Если промежуточное программное обеспечение, выполняющее пересылку, не использует хранилище данных модели, то для непосредственной пересылки требуется XA (двухфазовая фиксация транзакций)

Подумайте о том, что такого произошло в течение прошлого дня, что может иметь последствия. Можно ли упростить ситуацию, если эти факты описать в виде отдельных СОБЫТИЙ и опубликовать их в вашей системе? Действительно, это позволило бы избавиться от сложных запросов, потому что вы точно знаете, что именно и когда произошло, образуя контекст *действий, которые следует выполнить*. Вы можете выполнить их, получив уведомление о каждом СОБЫТИИ. Обработку, связанную в вводом-выводом и пакетными процессами, в течение дня можно разделить на короткие отрезки. В результате бизнес-ситуацию можно гармонизировать намного быстрее, позволяя пользователям выполнять дальнейшие действия.

Является ли СОБЫТИЕ результатом каждой команды АГРЕГАТА? Важно не только распознавать *необходимость* СОБЫТИЯ, но и знать, *когда* их можно *игнорировать*. В зависимости от аспектов технической реализации модели или целей сотрудничающих систем, СОБЫТИЯ могут возникать чаще, чем предполагают эксперты. Например, это происходит при **ПОРОЖДЕНИИ СОБЫТИЙ (4 и А)**.

Эти вопросы мы обсудим в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**, а здесь рассмотрим основные средства моделирования.

Моделирование событий

Рассмотрим ограничение из *Контекста управления гибким проектированием.* Эксперты в предметной области отражают потребность в этом СОБЫТИИ следующим образом (для выделения используется курсив).

Допустим, что каждый список заданий зарегистрирован для спринта. Он может быть зарегистрирован, только если его выпуск уже запланирован. Если же он зарегистрирован для другого спринта, первую регистрацию следует отменить. *Когда список регистрируется, следует уведомить об этом спринт и другие заинтересованные стороны.*

При моделировании СОБЫТИЙ им следует присваивать имена и свойства в соответствии с ЕДИНЫМ ЯЗЫКОМ в ОГРАНИЧЕННОМ КОНТЕКСТЕ, в котором они происходят. Если СОБЫТИЕ является результатом выполнения командной операции над АГРЕГАТОМ, то его имя обычно наследуется из выполняемой команды. Причиной СОБЫТИЯ является команда, значит, имя СОБЫТИЯ следует выбирать в соответствии с выполненной командой. Для примера рассмотрим сценарий, в котором мы регистрируем список заданий для спринта и публикуем СОБЫТИЕ, которое явно моделирует происходящее в предметной области.

Командная операция: `BacklogItem#commitTo(Sprint aSprint)`
Результирующее событие: `BacklogItemCommitted`

Имя СОБЫТИЯ несет информацию о том, что произошло (в прошлом) в АГРЕГАТЕ после завершения требуемой операции: "Список заданий для спринта зарегистрирован". Команда могла бы назвать СОБЫТИЕ более многословно, например `BacklogItemCommittedToSprint`, и это тоже было бы правильно. Однако в ЕДИНОМ ЯЗЫКЕ системы Scrum список заданий никогда не регистрируется ни для чего другого, кроме спринта. Иначе говоря, списки заданий планируются (schedules) к выпуску, а не регистрируются (committed) для выпуска. Нет никаких сомнений в том, что данное СОБЫТИЕ было опубликовано в результате выполнения операции `commitTo()`. Таким образом, имя СОБЫТИЯ уже несет в себе достаточно информации и является достаточно удобным для чтения. Впрочем, если ваша команда предпочитает более многословные имена, вы можете использовать их без ограничений.

При публикации СОБЫТИЙ из АГРЕГАТОВ важно, чтобы имя СОБЫТИЯ отражало прошлое время. Событие происходит не сейчас. Оно уже произошло. Имя должно содержать информацию об этом факте.

Выбрав правильное имя, необходимо определить свойства СОБЫТИЯ. Для этого необходима временная метка, указывающая, когда именно произошло СОБЫТИЕ. В языке Java мы могли бы использовать пакет `java.util.Date`.

```
package com.saasovation.agilepm.domain.model.product;

public class BacklogItemCommitted implements DomainEvent {
    private Date occurredOn;
    ...

}
```

Минимальный интерфейс `DomainEvent`, реализованный во всех событиях, гарантирует поддержку метода доступа `occurredOn()`. Он задает базовый контракт для всех СОБЫТИЙ.

```
package com.saasovation.agilepm.domain.model;

import java.util.Date;

public interface DomainEvent {
    public Date occurredOn();
}
```

Кроме того, команда должна определить остальные важные свойства, несущие информацию о том, что произошло. Посмотрим, как можно включить триггер, который запустит СОБЫТИЕ снова. Обычно он содержит информацию, позволяющую идентифицировать экземпляр АГРЕГАТА, в котором произошло событие, или информацию об экземплярах АГРЕГАТА, имеющих к этому отношение. Используя эту информацию, можно создать свойства с любыми параметрами, определяемыми СОБЫТИЕМ, если в ходе обсуждения признать это целесообразным. Возможно, для подписчиков окажутся полезными некоторые параметры перехода АГРЕГАТА из одного состояния в другое.

Анализ события `BacklogItemCommitted` привел к следующему коду.

```
package com.saasovation.agilepm.domain.model.product;

public class BacklogItemCommitted implements DomainEvent {
    private Date occurredOn;
    private BacklogItemId backlogItemId;
    private SprintId committedToSprintId;
```

```
    private TenantId tenantId;
    ...
}
```

Команда решила, что идентификатор объекта класса BacklogItem и его связь с объектом класса Sprint имеют значение. Происходящее СОБЫТИЕ относится к объекту класса BacklogItem и объекту класса Sprint. В то же время в это решение вовлечены и другие объекты. Согласно регламенту необходимо уведомить объект класса Sprint о том, что для него зарегистрирован объект класса BacklogItem. Таким образом, подписчик СОБЫТИЯ, находящийся в том же ОГРАНИЧЕННОМ КОНТЕКСТЕ, должен информировать объект класса Sprint, а это возможно, только если событие BacklogItemCommitted имеет свойство SprintId.

Кроме того, в многоарендной среде всегда необходима запись идентификатора TenantId, даже если он не передается как параметр команды. Это необходимо как для локальных, так и для внешних ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Локально команда должна использовать идентификатор TenantId для запроса объектов классов BacklogItem и Sprint из соответствующих ХРАНИЛИЩ (12). Аналогично любые внешние удаленные системы, ожидающие данного СОБЫТИЯ, должны знать, какой идентификатор TenantId к нему относится.

Как моделировать функции, поддерживаемые СОБЫТИЯМИ? Это очень просто, потому что СОБЫТИЕ обычно проектируется неизменяемым. Главное заключается в том, что интерфейс СОБЫТИЯ должен выражать его предназначение, а свойства должны отражать его причину. Большинство СОБЫТИЙ должны иметь конструктор, позволяющий только полную инициализацию состояния и методы чтения каждого из его свойств.

На основе вышесказанного команда ProjectOvation создала следующий код.

```
package com.saasovation.agilepm.domain.model.product;

public class BacklogItemCommitted implements DomainEvent {
    ...
    public BacklogItemCommitted(
            TenantId aTenantId,
            BacklogItemId aBacklogItemId,
            SprintId aCommittedToSprintId) {
        super();
        this.setOccurredOn(new Date());
        this.setBacklogItemId(aBacklogItemId);
        this.setCommittedToSprintId(aCommittedToSprintId);
```

```
        this.setTenantId(aTenantId);
    }

    @Override
    public Date occurredOn() {
        return this.occurredOn;
    }

    public BacklogItemId backlogItemId() {
        return this.backlogItemId;
    }

    public SprintId committedToSprintId() {
        return this.committedToSprintId;
    }

    public TenantId tenantId() {
        return this.tenant;
    }
    ...
}
```

После публикации СОБЫТИЯ **подписчик в локальном** ОГРАНИЧЕННОМ КОНТЕКСТЕ **может использовать его для уведомления объекта класса** `Sprint` **о том, что для него недавно был зарегистрирован объект класса** `BacklogItem`.

```
MessageConsumer.instance(messageSource, false)
    .receiveOnly(
            new String[] { "BacklogItemCommitted" },
            new MessageListener(Type.TEXT) {
        @Override
        public void handleMessage(
            String aType,
            String aMessageId,
            Date aTimestamp,
            String aTextMessage,
            long aDeliveryTag,
            boolean isRedelivery)

        throws Exception {
            // сначала выполняем дедубликацию сообщения aMessageId
            ...
            // получаем tenantId, sprintId и backlogItemId из JSON
            ...

            Sprint sprint =
                    sprintRepository.sprintOfId(tenantId, sprintId);

            BacklogItem backlogItem =
                    backlogItemRepository.backlogItemOfId(
                        tenantId,
                        backlogItemId);
```

```
            sprint.commit(backlogItem);
        }
    });
```

В соответствии с системными требованиями после обработки конкретного сообщения "BacklogItemCommitted" объект класса Sprint становится согласованным с объектом класса BacklogItem, который был для него зарегистрирован. Как подписчик получает это СОБЫТИЕ, мы расскажем позднее в этой главе.

Команда поняла, что возникла небольшая проблема. Как управлять транзакцией обновления объекта класса Sprint? Для этого можно было бы предусмотреть обработчик сообщения, но в любом случае код такого обработчика придется переделать. Было бы лучше всего делегировать эту задачу **ПРИКЛАДНОЙ СЛУЖБЕ (14)**, чтобы не нарушать принципы **ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ (4)**. Это позволило бы ПРИКЛАДНОЙ СЛУЖБЕ управлять транзакцией, что вполне естественно. В этом случае код обработчика мог бы выглядеть следующим образом.

```
MessageConsumer.instance(messageSource, false)
    .receiveOnly(
            new String[] { "BacklogItemCommitted" },
            new MessageListener(Type.TEXT) {
        @Override
        public void handleMessage(
            String aType,
            String aMessageId,
            Date aTimestamp,
            String aTextMessage,
            long aDeliveryTag,
            boolean isRedelivery)
        throws Exception {
            // получаем tenantId, sprintId и backlogItemId из JSON
            String tenantId = ...
            String sprintId = ...
            String backlogItemId = ...

            ApplicationServiceRegistry
                        .sprintService()
                        .commitBacklogItem(
                                tenantId, sprintId, backlogItemId);
        }
    });
```

В этом примере выполнять дедубликацию СОБЫТИЯ не обязательно, потому что регистрация объекта класса `BacklogItem` для объекта класса `Sprint` является идемпотентной операцией. Если конкретный объект класса `BacklogItem` уже зарегистрирован для объекта класса `Sprint`, новый запрос на регистрацию будет проигнорирован.

Иногда, если подписчик требует не только указания причины СОБЫТИЯ, возникает необходимость обеспечить дополнительное состояние и функцию. Эту ситуацию можно описать с помощью более сложного состояния (с большим количеством свойств) или с помощью операций, которые создают более сложное состояние. Тем самым подписчики избегают обратного запроса к АГРЕГАТУ, являющегося источником СОБЫТИЯ. Такой запрос может быть трудным или дорогим. СОБЫТИЯ могут усложняться при использовании шаблона ПОРОЖДЕНИЕ СОБЫТИЙ, потому что для СОБЫТИЯ, используемого для хранения и отправляемого за пределы ОГРАНИЧЕННОГО КОНТЕКСТА, может потребоваться дополнительное состояние. Примеры таких событий приведены в приложении А.

Заметки на доске

- Перечислите виды СОБЫТИЙ, которые уже возникли в вашей предметной области, но еще не зарегистрированы.
- Обратите внимание на то, как явное выделение СОБЫТИЙ в вашей модели улучшает проект.

Если требуется обеспечить итоговую согласованность, проще всего было бы идентифицировать АГРЕГАТЫ, зависящие от состояния других АГРЕГАТОВ.

Для вывода более сложных состояний с помощью операций убедитесь, что все дополнительные функции СОБЫТИЯ являются **ФУНКЦИЯМИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ**, которые обсуждались в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**. Это гарантирует неизменяемость объектов.

Характеристики АГРЕГАТОВ

Иногда СОБЫТИЯ проектируются так, чтобы они создавались по прямым запросам клиентов. Это происходит вследствие факта, который не является результатом выполнения какой-то функции экземпляра АГРЕГАТА. Возможно, пользователь системы инициировал какое-то действие, которое влечет за собой появление СОБЫТИЯ. Когда оно происходит, его можно моделировать как АГРЕГАТ и хранить в своем ХРАНИЛИЩЕ. Поскольку СОБЫТИЕ является следствием факта, имевшего место в прошлом, ХРАНИЛИЩЕ не должно допускать удаления СОБЫТИЯ.

При таком моделировании СОБЫТИЯ АГРЕГАТЫ становятся частью структуры модели. Таким образом, они являются не просто записью о прошлом факте. СОБЫТИЕ по-прежнему проектируется неизменяемым, но теперь ему можно присвоить генерируемый уникальный идентификатор. Однако иногда идентификатор можно обеспечить с помощью многочисленных свойств СОБЫТИЯ. Даже если уникальный идентификатор можно определить набором свойств, лучше все же присвоить СОБЫТИЮ сгенерированный уникальный идентификатор, как указано в главе, посвященной **СУЩНОСТЯМ (5)**. Это позволило бы варьировать схему СОБЫТИЯ в ходе проектирования, не подвергая опасности его уникальность.

При таком моделировании СОБЫТИЕ можно публиковать с помощью инфраструктуры обмена сообщениями одновременно с добавлением в свое ХРАНИЛИЩЕ. Для создания СОБЫТИЯ клиент может вызвать **СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ (7)**, добавить его в соответствующее хранилище, а затем опубликовать с помощью инфраструктуры обмена сообщениями. В рамках такого подхода для гарантии успеха ХРАНИЛИЩЕ и инфраструктура обмена сообщениями должны использовать одну и ту же базу данных или глобальную транзакцию (т.е. XA и двухфазную систему фиксации транзакций).

После того как инфраструктура обмена сообщениями успешно сохранит новое сообщение о СОБЫТИИ в соответствующем постоянном хранилище, она может асинхронно рассылать его в любую очередь слушателей, подписчиков и акторов, используя модель МОДЕЛЬ АКТОРОВ (Actor Model).[1] Если инфраструктура обмена сообщениями использует постоянное хранилище, отличающееся от хранилища, используемого моделью, и если оно не поддерживает глобальные транзакции, ваша СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ сначала будет сохранять СОБЫТИЕ в ХРАНИЛИЩЕ СОБЫТИЙ, которое в этом случае служит очередью сообщений, посылаемых за пределы сети. Каждое СОБЫТИЕ в этом случае обрабатывается компонентом пересылки, который рассылает его в пределах инфраструктуры обмена сообщениями. Этот способ будет рассмотрен в главе немного позднее.

Идентичность

Рассмотрим причины, по которым СОБЫТИЯМ приходится присваивать уникальный идентификатор. Иногда возникает необходимость отличать одно СОБЫТИЕ от другого, но это происходит редко. В ОГРАНИЧЕННОМ КОНТЕКСТЕ, в котором СОБЫТИЕ было вызвано, создано и опубликовано, редко возникает необходимость сравнивать СОБЫТИЯ. Что же делать, если СОБЫТИЯ по каким-то причинам все же надо сравнивать? И что делать, если СОБЫТИЕ было спроектировано как АГРЕГАТ?

[1] См. Модель параллельных акторов в языках Erlang и Scala. В частности, при использовании языков Scala и Java стоит изучить каркас Akka.

Иногда достаточно представить идентификатор СОБЫТИЯ с помощью его свойств аналогично ОБЪЕКТАМ-ЗНАЧЕНИЯМ. В этом случае используются имя и тип СОБЫТИЯ, идентификатор АГРЕГАТА (или идентификаторы АГРЕГАТОВ), а также временная метка о моменте возникновения СОБЫТИЯ.

Если СОБЫТИЕ моделируется как АГРЕГАТ, а также в других ситуациях, в которых СОБЫТИЯ должны сравниваться, но комбинации их свойств не позволяют отличать их друг от друга, можно присвоить СОБЫТИЮ формальный уникальный идентификатор. Впрочем, для присваивания уникального идентификатора могут существовать и другие причины.

Уникальный идентификатор может оказаться необходимым, если СОБЫТИЕ было опубликовано за пределами локального ОГРАНИЧЕННОГО КОНТЕКСТА, в котором оно возникло, и было переслано инфраструктурой обмена сообщениями. В некоторых ситуациях отдельные сообщения могут доставляться несколько раз. Это может произойти, если отправитель сообщения терпит крах до того, как инфраструктура обмена сообщениями подтвердит отправление сообщения.

Какой бы ни была причина повторной доставки сообщения, достаточно дать удаленным подписчикам возможность выполнять дедубликацию сообщений, если сообщение уже было получено. Для того чтобы облегчить решение этой задачи, некоторые инфраструктуры обмена сообщениями используют уникальный идентификатор в качестве части заголовка или конверта, содержащего сообщение. В этом случае моделировать генерацию не обязательно. Даже если система отправки сообщений не генерирует уникальный идентификатор автоматически для всех сообщений, издатели могут самостоятельно присваивать его СОБЫТИЮ или сообщению. В любом случае удаленные подписчики могут использовать уникальный идентификатор для дедубликации многократно присланных сообщений.

Нужна ли реализация функций `equals()` и `hashCode()`? Чаще всего такая необходимость возникает, только если они используются в локальном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Иногда СОБЫТИЯ, отосланные инфраструктурой и полученные подписчиками, не воспроизводятся в их естественном виде, а используются в формате XML, JSON или отображений "ключ-значение". С другой стороны, если СОБЫТИЕ спроектировано как АГРЕГАТ и сохранено в своем ХРАНИЛИЩЕ, тип СОБЫТИЯ должен сохраняться обоими стандартными методами.

Публикация событий за пределами модели предметной области

Следует избегать включения в модель предметной области информации о промежуточном программном обеспечении, образующем инфраструктуру обмена сообщениями. Эти компоненты должны существовать только в рамках

инфраструктуры. Хотя модель предметной области иногда может неявно использовать эту инфраструктуру, их нельзя связывать явным образом. Мы будем придерживаться подхода, в котором вообще не используется инфраструктура.

Одними из простейших и наиболее эффективных способов публикации СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ без привязки к компонентам, находящимся за пределами модели предметной области, является создание упрощенного **НАБЛЮДАТЕЛЯ** [Гамма и др.]. Впрочем, я использую более привычное имя этого шаблона — ИЗДАТЕЛЬ–ПОДПИСЧИК, которое авторы книги [Гамма и др.] тоже признали синонимом. Примеры использования этого шаблона являются упрощенными, поскольку в них нет сети подписчиков и издателей СОБЫТИЙ. Все зарегистрированные подписчики и издатель функционируют в одном и том же пространстве процессов и одном и том же потоке. После публикации СОБЫТИЯ каждый подписчик получает синхронное уведомление один за другим. Отсюда также следует, что все подписчики функционируют в рамках одной и той же транзакции, возможно, под управлением ПРИКЛАДНОЙ СЛУЖБЫ, которая является прямым клиентом модели предметной области.

Рассмотрим две половины шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК по отдельности, чтобы лучше понять контекст DDD.

Издатель

Вероятно, чаще всего СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ используются в ситуациях, в которых АГРЕГАТ создает СОБЫТИЕ и публикует его. Издатель находится внутри **МОДУЛЯ (9)** в рамках модели, но не моделирует никаких аспектов предметной области. Он лишь предоставляет услугу АГРЕГАТУ, который хочет уведомить подписчиков о СОБЫТИИ. Ниже приведен класс `DomainEventPublisher`, который соответствует этому определению. Абстрактное представление функционирования класса `DomainEventPublisher` показано на рис. 8.2.

```
package com.saasovation.agilepm.domain.model;

import java.util.ArrayList;
import java.util.List;

public class DomainEventPublisher {

    @SuppressWarnings("unchecked")
    private static final ThreadLocal<List> subscribers =
            new ThreadLocal<List>();

    private static final ThreadLocal<Boolean> publishing =
            new ThreadLocal<Boolean>() {
       protected Boolean initialValue() {
            return Boolean.FALSE;
      }
    };
```

```
    public static DomainEventPublisher instance() {
            return new DomainEventPublisher();
    }

    public DomainEventPublisher() {
       super();
    }

    @SuppressWarnings("unchecked")
    public <T> void publish(final T aDomainEvent) {
        if (publishing.get()) {
            return;
        }
        try {
              publishing.set(Boolean.TRUE);
              List<DomainEventSubscriber<T>> registeredSubscribers =
                      subscribers.get();
              if (registeredSubscribers != null) {
                  Class<?> eventType = aDomainEvent.getClass();
                  for (DomainEventSubscriber<T> subscriber :
                       registeredSubscribers) {
                       Class<?> subscribedTo =
                               subscriber.subscribedToEventType();
                       if (subscribedTo == eventType ||
                           subscribedTo == DomainEvent.class) {
                           subscriber.handleEvent(aDomainEvent);
                       }
                  }
              }
        } finally {
              publishing.set(Boolean.FALSE);
       }
    }

    public DomainEventPublisher reset() {
       if (!publishing.get()) {
            subscribers.set(null);
       }
       return this;
    }

    @SuppressWarnings("unchecked")
    public <T> void subscribe(DomainEventSubscriber<T> aSubscriber) {
       if (publishing.get()) {
           return;
       }
       List<DomainEventSubscriber<T>> registeredSubscribers =
            subscribers.get();
       if (registeredSubscribers == null) {
           registeredSubscribers =
                new ArrayList<DomainEventSubscriber<T>>();
           subscribers.set(registeredSubscribers);
       }
       registeredSubscribers.add(aSubscriber);

    }
}
```

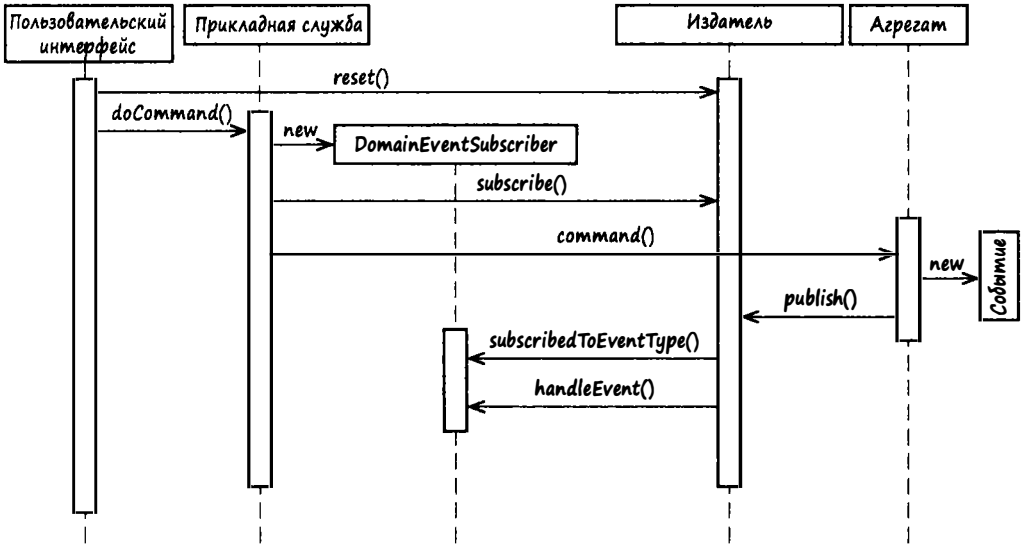


Рис. 8.2. Абстрактное представление последовательности взаимодействий между упрощенными НАБЛЮДАТЕЛЕМ, **ПОЛЬЗОВАТЕЛЬСКИМ ИНТЕРФЕЙСОМ (14)**, ПРИКЛАДНЫМИ СЛУЖБАМИ и **МОДЕЛЬЮ ПРЕДМЕТНОЙ ОБЛАСТИ (1)**

Поскольку каждый запрос, поступающий от пользователя системы, обрабатывается отдельным, предназначенным для этого потоком, мы разделяем подписчиков по потокам. Итак, каждому потоку выделены две переменные класса `ThreadLocal` — `subscribers` и `publishing`. Когда заинтересованные стороны используют операцию `subscribe()` для регистрации, ссылка на объект подписчика добавляется в объект `List`, соответствующий потоку. В потоке может зарегистрироваться любое количество подписчиков.

В зависимости от сервера приложения потоки могут объединяться в пулы и повторно использоваться последовательными запросами. Мы не хотим, чтобы подписчик, зарегистрированный потоком для предыдущего запроса, оставался зарегистрированным для следующего запроса, который повторно использует поток. Когда система получает новый запрос пользователя, необходимо выполнить операцию `reset()` для удаления всех предыдущих подписчиков. Это гарантирует, что список подписчиков будет содержать только тех, кто зарегистрировался в данный момент. Например, на уровне представления (`"Пользовательский интерфейс"` на рис. 8.2) мы можем перехватывать все запросы с помощью фильтра. Перехватывающий компонент может выполнять операцию `reset()`.

```
// в фильтрующем веб-компоненте после получения запроса пользователя
DomainEventPublisher.instance().reset();
```

```
// позднее в ПРИКЛАДНОЙ СЛУЖБЕ в ходе выполнения того же самого запроса
DomainEventPublisher.instance().subscribe(subscriber);
```

Прослеживая выполнение этого кода двумя разными компонентами, показанными на рис. 8.2, мы видим, что потоку соответствует только один зарегистрированный подписчик. Реализация метода `subscribe()` демонстрирует нам, что подписчики могут регистрироваться, только когда издатель не находится в процессе публикации. Это позволяет избежать таких проблем, как исключения, связанные с параллельными изменениями объекта класса `List`. Эта проблема возникает, если подписчики выполняют обратные вызовы к издателю, чтобы добавить новых подписчиков на обрабатываемое СОБЫТИЕ.

Далее обратите внимание на то, как АГРЕГАТ публикует СОБЫТИЕ. Продолжая прослеживать пример, мы видим, что после успешного выполнения метода `commitTo()` класса `BacklogItem` публикуется событие `BacklogItemCommitted`.

```
public class BacklogItem extends ConcurrencySafeEntity {
    ...
    public void commitTo(Sprint aSprint) {
        ...
        DomainEventPublisher
            .instance()
            .publish(new BacklogItemCommitted(
                this.tenantId(),
                this.backlogItemId(),
                this.sprintId()));
    }
    ...
}
```

В ходе выполнения метод `publish()` из класса `DomainEventPublisher` выполняет перебор всех зарегистрированных подписчиков. Вызов метода `subscribedToEventType()` из каждого подписчика позволяет отфильтровать всех подписчиков, не подписанных на СОБЫТИЕ конкретного типа. Подписчики, возвращающие на запрос фильтра объект типа `DomainEvent.class`, будут получать все СОБЫТИЯ. Все подписчики, прошедшие проверку, получают опубликованное СОБЫТИЕ с помощью своего метода `handleEvent()`. После того как все подписчики будут либо отфильтрованы, либо уведомлены, издатель завершает работу.

Как и метод `subscribe()`, метод `publish()` не позволяет вложенным запросам публиковать СОБЫТИЯ. Для этого выполняется проверка булевой переменной `publishing`, соответствующей потоку, которая должна быть равной `false` и не позволять методу `publish()` выполнять перебор и диспетчеризацию.

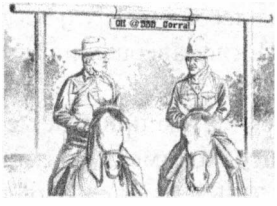
Как распространить публикацию СОБЫТИЯ на удаленные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ, поддерживающие автономные службы? Вскоре мы рассмотрим этот вопрос, а пока поближе познакомимся с локальными подписчиками.

Подписчики

Какие компоненты регистрируют подписчиков на СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ? Вообще говоря, это делают **ПРИКЛАДНЫЕ СЛУЖБЫ (14)**, а иногда СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Подписчиком может быть любой компонент, функционирующий в том же потоке, что и АГРЕГАТ, публикующий СООБЩЕНИЕ, и способный оформить подписку до того, как СОБЫТИЕ будет опубликовано. Это значит, что подписчик регистрируется в ходе выполнения метода, который используется моделью предметной области.

Ковбойская логика

LB: Я хочу подписаться на *The Fence Post*, чтобы найти там еще более тупые шутки для этой книги.



Поскольку ПРИКЛАДНЫЕ СЛУЖБЫ являются прямыми клиентами модели предметной области в рамках ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ, они идеально подходят для регистрации подписчика издателем, прежде чем он выполнит функцию АГРЕГАТА, генерирующую СОБЫТИЕ. Рассмотрим пример ПРИКЛАДНОЙ СЛУЖБЫ, выполняющей подписку.

```
public class BacklogItemApplicationService ... {
    public void commitBacklogItem(
            Tenant aTenant,
            BacklogItemId aBacklogItemId,
            SprintId aSprintId) {

        DomainEventSubscriber subscriber =
                new DomainEventSubscriber<BacklogItemCommitted>() {
            @Override
            public void handleEvent(BacklogItemCommitted aDomainEvent) {
                // здесь выполняется обработка события ...
            }
            @Override
            public Class<BacklogItemCommitted> subscribedToEventType() {
                return BacklogItemCommitted.class;
            }
        }

        DomainEventPublisher.instance().subscribe(subscriber);
```

```
        BacklogItem backlogItem =
              backlogItemRepository
                  .backlogItemOfId(aTenant, aBacklogItemId);

        Sprint sprint = sprintRepository.sprintOfId(aTenant, aSprintId);

        backlogItem.commitTo(sprint);
    }
}
```

В этом искусственном примере класс BacklogItemApplicationService представляет собой ПРИКЛАДНУЮ СЛУЖБУ со служебным методом commitBacklogItem(). Этот метод создает экземпляр анонимного объекта класса DomainEventSubscriber. Затем координатор задач ПРИКЛАДНОЙ СЛУЖБЫ регистрирует подписчика с помощью объекта класса DomainEventPublisher. В заключение служебный метод использует ХРАНИЛИЩЕ для получения экземпляров классов BacklogItem и Sprint и выполняет функцию commitTo() для задачи. После завершения метод commitTo() публикует СОБЫТИЕ типа BacklogItemCommitted.

В этом примере не показано, что именно подписчик делает с СОБЫТИЕМ. При необходимости он может отправить электронное письмо о том, что произошло СОБЫТИЕ BacklogItemCommitted. Он может сохранить СОБЫТИЕ в ХРАНИЛИЩЕ СОБЫТИЙ. Он может переслать событие с помощью инфраструктуры обмена сообщениями. Обычно в двух последних случаях — при сохранении СОБЫТИЯ в ХРАНИЛИЩЕ СОБЫТИЙ и при пересылке события с помощью инфраструктуры обмена сообщениями — не обязательно создавать специальную ПРИКЛАДНУЮ СЛУЖБУ для обработки события именно таким способом. Для этого можно разработать отдельный компонент подписчика. Пример отдельного специального компонента, сохраняющего СОБЫТИЯ в ХРАНИЛИЩЕ СОБЫТИЙ, описан ниже в разделе "Хранилище событий".

Внимательно следите за тем, что делает обработчик событий

Напомним, что ПРИКЛАДНАЯ СЛУЖБА управляет транзакцией. Не используйте уведомление о событии для модификации второго экземпляра АГРЕГАТА. Это нарушает правило, запрещающее модифицировать отдельный экземпляр АГРЕГАТА во время выполнения транзакции.

Подписчик *не должен* получать другой экземпляр АГРЕГАТА и выполнять над ним модифицирующие операции. Это может нарушить правило, запрещающее модифицировать отдельный АГРЕГАТ в ходе отдельной транзакции, как указано в главе, посвященной **АГРЕГАТАМ (10)**. Как сказано в книге [Эванс], согласованность всех экземпляров АГРЕГАТА, кроме одного экземпляра, используемого в отдельной транзакции, должна обеспечиваться средствами асинхронной работы.

Пересылка СОБЫТИЯ через инфраструктуру обмена сообщениями обеспечила бы асинхронную доставку внешним подписчикам. Каждый из этих асинхронных подписчиков может модифицировать дополнительный экземпляр АГРЕГАТА в рамках одной или нескольких отдельных транзакций. Дополнительные экземпляры АГРЕГАТА могут находиться как в одном и том же ОГРАНИЧЕННОМ КОНТЕКСТЕ, так и в разных. Публикация СОБЫТИЯ в любом количестве внешних ОГРАНИЧЕННЫХ КОНТЕКСТОВ других **ПОДОБЛАСТЕЙ (2)** подчеркивает важность слов ПРЕДМЕТНАЯ ОБЛАСТЬ в термине СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. Иначе говоря, события представляют собой концепцию масштаба предметной области, а не отдельного ОГРАНИЧЕННОГО КОНТЕКСТА. Контракт публикации СОБЫТИЯ должен распространяться как минимум на все предприятие и даже шире. Впрочем, это не запрещает доставлять СОБЫТИЯ потребителям в одном и том же ограниченном контексте. Еще раз проанализируйте рис. 8.1.

Иногда необходимо, чтобы подписчиков регистрировали СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Это объясняется теми же причинами, по которым подписчиков регистрируют ПРИКЛАДНЫЕ СЛУЖБЫ, но в данном случае существуют дополнительные резоны прослушивания СОБЫТИЙ, связанные со спецификой предметной области.

Распространение новостей в удаленных ОГРАНИЧЕННЫХ КОНТЕКСТАХ

Существует несколько способов сообщить удаленным ОГРАНИЧЕННЫМ КОНТЕКСТАМ о событии, произошедшем в вашем ОГРАНИЧЕННОМ КОНТЕКСТЕ. Основная идея заключается в том, чтобы организовать какую-то форму передачи сообщений и создать механизм передачи сообщений в масштабе предприятия. Точнее говоря, механизм, о котором идет речь, выходит далеко за рамки простых облегченных компонентов шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК. Ниже мы обсудим, что произойдет, если отказаться от этого упрощенного механизма.

Существует много компонентов для передачи сообщений, которые обычно относятся к классу промежуточного программного обеспечения. Эти компоненты — от открытых компонентов ActiveMQ, RabbitMQ, Akka, NServiceBus и MassTransit до различных лицензионных продуктов — предоставляют массу возможностей. Можно даже использовать доморощенную форму передачи сообщений на основе ресурсов REST, в которой автономные системы представляют собой заинтересованные стороны, обращающиеся к издательской системе, требуя уведомлений о событиях, которые еще не были обработаны. Все это можно описать шаблоном **ИЗДАТЕЛЬ–ПОДПИСЧИК** [Гамма и др.] с учетом разнообразных

преимуществ и недостатков. Многое зависит от объема финансирования, вкуса, функциональных требований и нефункциональных качеств, которые должна учитывать команда.

Использование любого механизма обмена сообщениями между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ требует наличия регистрации для гарантии итоговой согласованности. Против этого нечего возразить. Изменения в одной модели, которые могут повлечь изменения в других моделях, с течением времени окажутся не полностью согласованными. И что еще важнее, в зависимости от трафика отдельных систем и их влияния друг на друга, в один прекрасный момент может оказаться, что системы как целое больше не являются полностью согласованными друг с другом.

Согласованность инфраструктуры обмена сообщениями

Во всех этих разговорах об итоговой согласованности вас может удивить, что по крайней мере два механизма в системе передачи сообщений всегда должны быть согласованными друг с другом: постоянное хранилище, используемое в модели предметной области, и постоянное хранилище, на основе которого функционирует инфраструктура обмена сообщениями, используемая для пересылки СОБЫТИЙ, публикуемых моделью. Это необходимо для того, чтобы при сохранении изменений модели одновременно гарантировалась доставка СОБЫТИЯ, и если СОБЫТИЕ доставляется путем передачи сообщений, оно отражало бы истинную ситуацию, в которой модель его опубликовала. Если эти хранилища оказываются несогласованными, в одной или нескольких взаимозависимых моделях могут возникнуть некорректные состояния.

Как достичь согласованности между моделью и механизмом постоянного хранения событий? Существуют три основных способа.

1. Ваша модель предметной области и инфраструктура обмена сообщениями совместно используют одно и то же постоянное хранилище (например, хранилище данных). Это позволяет регистрировать изменения модели и вставку нового сообщения в рамках одной и той же локальной транзакции. Преимуществом такого подхода является относительно высокая производительность. Относительным недостатком этого способа является то, что области хранения данных в системе передачи сообщений (например, в виде баз данных) могут находиться в той же базе данных (или схеме), что и области хранения данных модели. Это дело вкуса. Разумеется, это не слишком важно, если вы решили, что хранилище модели и хранилище системы передачи сообщений не могут использоваться совместно.
2. Постоянное хранилище вашей модели предметной области и постоянное хранилище системы передачи сообщений управляются глобальной

ХА-транзакцией (механизмом двухфазовой фиксации транзакций). Преимуществом этого подхода является то, что вы можете разделить хранилища модели и системы передачи сообщений. Недостатком этого способа является тот факт, что глобальные транзакции требуют специальной поддержки, которая может быть не у всех постоянных хранилищ и систем передачи сообщений. Глобальные транзакции слишком дороги и имеют низкую производительность. Кроме того, возможно, что либо хранилище модели, либо хранилище механизма передачи сообщений, или оба хранилища несовместимы с ХА-транзакциями.

3. Можно создать специальную область для хранения СОБЫТИЙ (например, таблицу базы данных) в том же самом постоянном хранилище, которое используется вашей моделью предметной области. Это ХРАНИЛИЩЕ СОБЫТИЙ, которое обсуждалось выше. Такой способ похож на первый вариант; однако эта область хранения контролируется не механизмом передачи сообщений, а ОГРАНИЧЕННЫМ КОНТЕКСТОМ. Внешний компонент, созданный вами, использует ХРАНИЛИЩЕ СОБЫТИЙ для публикации всех сохраненных и неопубликованных СОБЫТИЙ с помощью механизма передачи сообщений. Преимуществом этого подхода является то, что ваша модель и ваши СОБЫТИЯ гарантированно будут согласованными в рамках отдельной локальной транзакции. Кроме того, это позволяет использовать преимущества ХРАНИЛИЩА СОБЫТИЙ, включая способность создавать каналы уведомления на основе технологии REST. Этот подход позволяет использовать инфраструктуру обмена сообщениями, в которых хранилище сообщений является полностью закрытым от внешнего мира. Недостатком этого способа является тот факт, что если вместе с ХРАНИЛИЩЕМ СОБЫТИЙ используется промежуточный механизм передачи сообщений, то механизм пересылки СОБЫТИЙ должен быть специально разработан пользователем, чтобы иметь возможность посылать СОБЫТИЯ через механизм передачи сообщений, а клиенты должны выполнять дедубликацию входящих сообщений (см. раздел "Хранилище событий").

В своих примерах я использую третий подход. Несмотря на его недостатки, он имеет ряд преимуществ, которые заслуживают описания. Это не значит, что остальные подходы плохи. Вы и ваша команда можете выбирать любой из них.

Автономные службы и системы

Использование СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ позволяет проектировать любое количество систем предприятия как *автономные службы и системы*. Термин *автономная служба* в данном случае означает крупномодульную бизнес-службу,

которая может рассматриваться как система или приложение, действующую в основном независимо от других аналогичных служб предприятия. Автономная служба может иметь много интерфейсных терминалов службы, предлагая удаленным клиентам множество интерфейсов технических служб. Высокая степень независимости от других систем достигается благодаря отсутствию внутренних вызовов удаленных процедур (RPCs), в которых запрос пользователя удовлетворяется только после успешного выполнения запроса API к удаленной системе.

Поскольку иногда удаленная система бывает недоступной или перегруженной, процедура RPC, обращающаяся к зависимой системе, может снизить производительность. При увеличении количества систем с интерфейсами RPC API этот риск увеличивается. Таким образом, избегая внутренних вызовов удаленных процедур, мы уменьшаем зависимость и снижаем риск полного отказа и/или снижения производительности из-за отказа или снижения производительности удаленных систем.

Вместо того чтобы обращаться к другим системам, используйте асинхронную передачу сообщений, стремясь достигнуть большей степени независимости между системами, т.е. автономности. После получения сообщений, несущих СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ из ОГРАНИЧЕННЫХ КОНТЕКСТОВ вовне предприятия, выполните функцию вашей модели, которая отражает смысл этих СОБЫТИЙ в пределах вашего ОГРАНИЧЕННОГО КОНТЕКСТА. Это не означает простую репликацию или точное копирование объектов из других бизнес-служб в вашу бизнес-службу. Правда, некоторые данные могут копироваться из одной системы в другую. Как минимум скопированные данные будут содержать уникальные идентификаторы некоторых внешних АГРЕГАТОВ. Однако объекты в одной системе редко бывают, если вообще бывают, точными копиями объектов из окружающих систем. Если существует вероятная ошибка моделирования, посмотрите **ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (2)** и **КАРТЫ КОНТЕКСТА (3)**, чтобы понять, в чем причина проблемы и как ее избежать. Фактически, если СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ разработаны правильно, то они будут крайне редко включать в свое состояние целые объекты.

СОБЫТИЕ должно содержать некоторый ограниченный набор параметров команды и/или состояния АГРЕГАТА, которое передает достаточно информации, чтобы обеспечить правильную реакцию подписанных ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Конечно, если какое-либо СОБЫТИЕ не содержит достаточно информации ни для одного подписчика, контракт СОБЫТИЯ в масштабе предметной области должен быть изменен, чтобы обеспечить поставку правильной информации. Это, вероятно, означает проектирование совершенно новой версии СОБЫТИЯ или совершенно другого СОБЫТИЯ.

Также верно, что в некоторых случаях использования процедур RPC избежать нелегко. Некоторые устаревшие системы могут поддерживать только RPC. Кроме того, когда трансляцию концепции или набора концепций из внешнего

ОГРАНИЧЕННОГО КОНТЕКСТА в локальный ОГРАНИЧЕННЫЙ КОНТЕКСТ очень трудно выполнить, экстраполяция достаточной информации из многократных СОБЫТИЙ может увеличивать сложность проекта. Если вы вынуждены реплицировать понятия, объекты и их связи из внешней модели в вашу собственную модель, то, возможно, придется смириться с использованием RPC. Этот вопрос нужно решать в каждом конкретном случае, и я предлагаю не признавать RPC слишком легко. Если этого нельзя избежать, то либо согласитесь на использование RPC, либо попытайтесь повлиять на команду, которая разрабатывает внешнюю модель, чтобы найти способ упростить их проект. По общему признанию, последнее может быть очень трудной задачей, если не невозможной.

Терпимость к задержкам

Не создадут ли проблемы потенциально длинные периоды ожидания (когда задержка превышает несколько миллисекунд), прежде чем сообщение будет получено? Конечно, этот вопрос заслуживает тщательного анализа, учитывая, что асинхронные данные могут оказывать вредное и даже разрушительное действие. Мы должны спросить, какой промежуток времени между последовательными состояниями является допустимым и какие задержки следует считать слишком большими. Эксперты в предметной области, вероятно, очень хорошо понимают, какие задержки считаются приемлемыми и неприемлемыми. Разработчики могут удивиться, узнав, что в большинстве ситуаций между последовательными состояниями допускаются периоды длительностью несколько секунд, минут, часов или даже дней. Нельзя сказать, что это всегда верно. Но мы не должны предполагать, что в любой предметной области всегда обязательно нужно обеспечивать почти мгновенную согласованность.

Иногда к информативному ответу будет приводить следующий вопрос: как бизнес работал до внедрения компьютеров или как он будет работать без них теперь? Даже самая простая из систем документооборота никогда не обеспечивает мгновенной согласованности. Мгновенная согласованность успешно реализуется только в автоматизированных компьютерных системах. Мы могли бы прийти к заключению, что конечная последовательность приносит выгоду для бизнеса.

Представьте себе, что ПОДОБЛАСТЬ использовалась для планирования будущих действий команды. После одобрения любого из отдельных действий публикуется СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, которое отражает это одобрение: `TeamActivityApproved`. Оно следует за любым количеством других СОБЫТИЙ, которые были уже опубликованы и сообщают о происхождении и определении всех ранее одобренных действий. Другой ОГРАНИЧЕННЫЙ КОНТЕКСТ реагирует на одобрение, намечая последнее подготовленное действие, чтобы выполнить его когда-то в связи со всеми другими одобренными действиями.

Допустим, что любое действие определяется и одобряется по крайней мере за неделю до того, как начнется его выполнение. Тогда не следует ли сделать так, чтобы СОБЫТИЕ сопровождалось включением одобренного действия в календарный план, откуда его можно было бы извлечь через минуты, часы или, возможно, даже дни после одобрения? Возможно, дни не были бы приемлемы. Однако, если бы отключение системы заставило отсрочить СОБЫТИЕ на много часов (вероятно, это маловероятная ситуация), то не были бы эти часы простоя совершенно недопустимой задержкой? Нет, потому что отключение системы — редкое явление, которое обязательно будет устранено, а действие все равно будет отложено на недели. Поскольку это так, то, конечно, типичная задержка, возможно, на нескольких секунд — как верхний предел — для того же самого СОБЫТИЯ, поступающего в обычных обстоятельствах, была бы не только терпимой, но и приемлемой. На самом деле любые фактические задержки даже могут быть незаметными.

Ковбойская логика

AJ: Это "скоро" по-кентуккски?

LB: Наверное, это нью-йоркская "минута".

Несмотря на правдоподобность приведенного примера, другие бизнес-службы могут потребовать более высокую пропускную способность. Максимальная терпимость к задержкам должна быть хорошо проанализирована, а у систем должны быть архитектурные качества, позволяющие удовлетворить выдвигаемые требования и даже перевыполнить их. Автономные службы и сопутствующая инфраструктура обмена сообщениями должны обладать высокой доступностью и быть масштабируемыми, чтобы полностью выполнить строгие нефункциональные требования, принятые на предприятии.

Хранилище событий

Поддержание хранилища всех СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ для единственного ОГРАНИЧЕННОГО КОНТЕКСТА обладает несколькими потенциальными преимуществами. Рассмотрим то, что можно сделать, если бы вам было необходимо сохранить отдельное СОБЫТИЕ для каждой командной функции модели, которая может быть выполнена.

1. Можно использовать ХРАНИЛИЩЕ СОБЫТИЙ как очередь для публикации всех СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ посредством инфраструктуры

обмена сообщениями. Именно этот вариант в основном рассматривается в книге. Это обеспечивает интеграцию между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, в рамках которой удаленные подписчики реагируют на события в соответствии со своими контекстуальными потребностями. (См. предыдущий раздел, "Распространение новостей в удаленных ограниченных контекстах".)

2. Можно использовать то же самое ХРАНИЛИЩЕ СОБЫТИЙ для передачи клиентам уведомлений о СОБЫТИИ с помощью технологии REST. (С теоретической точки зрения это эквивалентно первому пункту, но на практике отличается от него.)
3. Можно исследовать хронологическую запись результатов выполнения каждой команды, которая когда-либо выполнялась на модели. Это могло бы помочь проследить ошибки не только в модели, но и у клиентов. Важно понимать, что ХРАНИЛИЩЕ СОБЫТИЙ — это не просто контрольный журнал. Контрольные журналы могут быть полезными для отладки, но они редко содержат полные результаты выполнения каждой команды АГРЕГАТА.
4. Можно использовать данные для определения тенденций, прогнозирования и другой деловой аналитики. Часто компании понятия не имеют, как можно использовать такие исторические данные, пока не поймут, что нуждаются в них. Если ХРАНИЛИЩЕ СОБЫТИЙ не будет работать с самого начала, то исторические данные окажутся недоступными в тот момент, когда они будут необходимы.
5. Можно использовать СОБЫТИЯ для воссоздания каждого экземпляра АГРЕГАТА при извлечении его из ХРАНИЛИЩА. Это необходимая часть шаблона ПОРОЖДЕНИЕ СОБЫТИЙ. Для этого к экземпляру АГРЕГАТА в хронологическом порядке применяются все ранее сохраненные СОБЫТИЯ. Для того чтобы оптимизировать процесс воссоздания экземпляра, можно создавать снимки любого числа сохраненных СОБЫТИЙ (например, группы из 100 событий).
6. Применяя предыдущий пункт, можно отменять блоки изменений АГРЕГАТА. При этом использование некоторых СОБЫТИЙ для воссоздания заданного экземпляра АГРЕГАТА можно предотвратить (например, удалив его или отметив как устаревшее). Можно также исправлять СОБЫТИЯ или вставлять дополнительные СОБЫТИЯ, чтобы исправить ошибки в потоке СОБЫТИЙ.

В зависимости от своего предназначения ХРАНИЛИЩЕ СОБЫТИЙ может иметь определенные особенности. Поскольку примеры, представленные здесь, прежде всего мотивированы преимуществами первого и второго вариантов, наше

ХРАНИЛИЩЕ СОБЫТИЙ в основном предназначено для хранения сериализованных событий в хронологическом порядке. Это не означает, что невозможно использовать СОБЫТИЯ для получения всех преимуществ четырех первых вариантов, потому что преимущества третьего и четвертого вариантов основаны на том факте, что мы создаем отчет обо всех значительных СОБЫТИЯХ в предметной области. Следовательно, достижение преимуществ третьего и четвертого вариантов обеспечивается первыми двумя. Однако мы не будем пытаться улучшить наше ХРАНИЛИЩЕ СОБЫТИЙ для достижения преимуществ пятого и шестого вариантов.

Для того чтобы достичь преимуществ первого и второго вариантов, необходимо выполнить несколько шагов. Эти шаги представлены на рис. 8.3. Сначала мы обсудим этапы, показанные на диаграмме последовательностей, а также задействованные при этом компоненты. Мы сделаем это на примере разработки проекта в компании SaaSOvation.

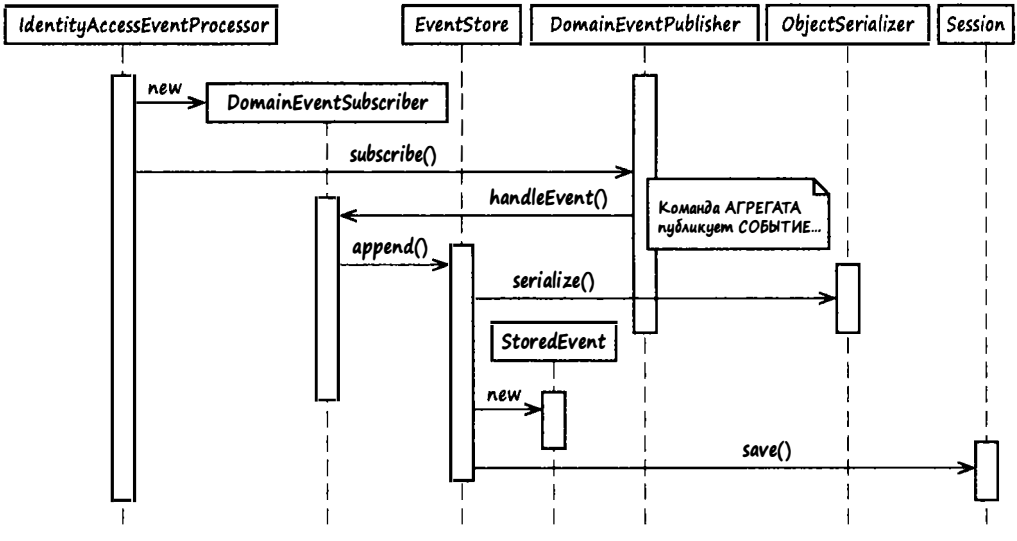


Рис. 8.3. Класс `IdentityAcessEventProcessor` анонимно подписан на все СОБЫТИЯ модели. Он делегирует их классу `EventStore`, который сериализует их в объекты класса `StoredEvent` и сохраняет их

По каким бы причинам мы ни использовали ХРАНИЛИЩЕ СОБЫТИЙ, в первую очередь необходимо создать подписчика, который будет получать каждое СОБЫТИЕ, опубликованное моделью. Команда решила сделать это, используя аспектно-ориентированную привязку, которая может вставлять себя в путь выполнения каждой ПРИКЛАДНОЙ СЛУЖБЫ в системе.

Вот какой компонент команда SaaSOvation сделала для *Контекста идентификации и доступа*. Он имеет единственную обязанность — следить за тем, чтобы все СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ были сохранены.

```
@Aspect
public class IdentityAccessEventProcessor {
    ...
    @Before(
    "execution(* com.saasovation.identityaccess.application.*.*(..))")
    public void listen() {
         DomainEventPublisher
              .instance()
              .subscribe(new DomainEventSubscriber<DomainEvent>() {

                 public void handleEvent(DomainEvent aDomainEvent) {
                     store(aDomainEvent);
                 }

                 public Class<DomainEvent> subscribedToEventType() {
                     return DomainEvent.class; // все события предметной области
                 }
         });
     }

     private void store(DomainEvent aDomainEvent) {
         EventStore.instance().append(aDomainEvent);
    }
}
```

Это простой процессор СОБЫТИЙ, который мог бы использоваться любым другим ОГРАНИЧЕННЫМ КОНТЕКСТОМ с той же самой целью. Он разработан как аспект (с помощью аспектно-ориентированного каркаса Spring), который перехватывает все вызовы метода ПРИКЛАДНОЙ СЛУЖБЫ. После выполнения метода ПРИКЛАДНОЙ СЛУЖБЫ этот процессор прослушивает все СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, опубликованные в ходе взаимодействия ПРИКЛАДНОЙ СЛУЖБЫ с моделью. Процессор регистрирует подписчика в экземпляре класса `DomainEventPublisher`, связанном с потоком выполнения. Подписчик имеет широкий фильтр, о чем свидетельствует обращение к классу `DomainEvent.class` из метода `subscribedToEventType()`. Возврат в этот класс указывает на то, что подписчик хочет получать все СОБЫТИЯ. При вызове метод `handleEvent()` делегирует событие методу `store()`, который, в свою очередь, делегирует СОБЫТИЕ классу `EventStore`, чтобы добавить событие в конец текущего ХРАНИЛИЩА СОБЫТИЙ.

Рассмотрим метод append() компонента EventStore.

```
package com.saasovation.identityaccess.application.eventStore;
...
public class EventStore ... {
    ...
    public void append(DomainEvent aDomainEvent) {

        String eventSerialization =
           EventStore.objectSerializer().serialize(aDomainEvent);

        StoredEvent storedEvent =
           new StoredEvent(
                    aDomainEvent.getClass().getName(),
                    aDomainEvent.occurredOn(),
                    eventSerialization);

        this.session().save(storedEvent);

        this.setStoredEvent(storedEvent);
    }
}
```

Метод store() сериализует экземпляр класса DomainEvent, помещая его в новый экземпляр класса StoredEvent, а затем записывает этот новый объект в ХРАНИЛИЩЕ СОБЫТИЙ. Рассмотрим часть класса StoredEvent, в которой хранится сериализованный объект класса DomainEvent.

```
package com.saasovation.identityaccess.application.eventStore;
...
public class StoredEvent {
    private String eventBody;
    private long eventId;
    private Date occurredOn;
    private String typeName;

    public StoredEvent(
           String aTypeName,
           Date anOccurredOn,
           String anEventBody) {
       this();
       this.setEventBody(anEventBody);
       this.setOccurredOn(anOccurredOn);
       this.setTypeName(aTypeName);
    }
    ...
}
```

Каждый экземпляр класса StoredEvent получает уникальное последовательное значение, автоматически генерируемое базой данных, и присваивает его переменной eventId. Его переменная eventBody содержит сериализованную форму объекта класса DomainEvent. В данном случае используется сериализация в формат JSON с помощью библиотеки [Gson], хотя мы могли бы использовать другую форму. Переменная typeName содержит название конкретного класса соответствующего СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, а переменная occurredOn — это копия переменной occurredOn из класса DomainEvent.

Все объекты класса StoredEvent сохраняются в таблице MySQL. Для сериализаций события резервируется много места, хотя 65 тысяч символов несомненно более чем достаточно для хранения одного экземпляра.

```
CREATE TABLE 'tbl_stored_event' (
    'event_id' int(11) NOT NULL auto_increment,
    'event_body' varchar(65000) NOT NULL,
    'occurred_on' datetime NOT NULL,
    'type_name' varchar(100) NOT NULL,
    PRIMARY KEY ('event_id')
) ENGINE=InnoDB;
```

Итак, для того чтобы создать ХРАНИЛИЩЕ СОБЫТИЙ для всех экземпляров СОБЫТИЙ, изданных АГРЕГАТАМИ в модели предметной области, необходимо задействовать несколько компонентов. Позднее мы рассмотрим их более подробно. Теперь давайте посмотрим, как сохраненные записи о том, что произошло в нашей модели, могут использоваться другими системами.

Архитектурные стили для пересылки сохраняемых событий

Как только ХРАНИЛИЩЕ СОБЫТИЙ заполнено, СОБЫТИЯ можно пересылать в виде уведомлений заинтересованным сторонам. Мы рассмотрим два стиля предоставления доступа к этим СОБЫТИЯМ. Один стиль основан на ресурсах RESTful, которые запрашиваются клиентами, а второй стиль предусматривает отправление сообщений с помощью механизма передачи сообщений, основанного на промежуточном программном обеспечении.

Честно говоря, подход, основанный на технологии REST, не является истинным методом пересылки сообщений. При этом он позволяет достичь такого же результата, как и стиль ИЗДАТЕЛЬ–ПОДПИСЧИК, поскольку клиент электронной почты является "подписчиком" электронных сообщений, "публикуемых" сервером электронной почты.

Публикация уведомлений в виде ресурсов RESTful

Стиль REST публикации СОБЫТИЙ лучше всего работает, когда он используется в среде, удовлетворяющей условиям шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК (иначе говоря, когда многие потребители заинтересованы в одних и тех же событиях, получаемых от одного производителя). С другой стороны, если попытаться использовать стиль в виде шаблона ОЧЕРЕДЬ, то ничего не получится. Рассмотрим преимущества и недостатки подхода RESTful.

- Если потенциально много клиентов могут зайти на единственный ресурс URI, чтобы запросить один и тот же набор уведомлений, подход RESTful работает хорошо. По существу, уведомления в этом случае распространяются для любого количества запрашивающих потребителей. Это соответствует описанию базового шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК, даже при том, что он использует модель *вытягивания* (pull model), а не *проталкивания* (push model).[2]
- Если один или несколько потребителей должны запрашивать ресурсы у многих производителей, чтобы сформировать отдельный набор задач для их выполнения в определенной последовательности, то подход RESTful может оказаться неудобным. Это напоминает ОЧЕРЕДЬ, в которой потенциально много производителей должны подать уведомления одному или нескольким потребителям, причем порядок получения может иметь значение. Модель опроса обычно плохо подходит для реализации ОЧЕРЕДЕЙ.

Подход RESTful к публикации уведомлений о СОБЫТИИ является противоположностью публикации с помощью типичной инфраструктуры обмена сообщениями. "Издатель" не поддерживает ряд зарегистрированных "подписчиков", потому что заинтересованным сторонам ничего не рассылается. Вместо этого подход требует, чтобы клиенты REST сами запрашивали уведомления, используя известный ресурс URI.

Рассмотрим подход RESTful на верхнем уровне. Если вы знаете, как работают веб-каналы Atom, то этот подход покажется вам очень знакомым. На самом деле он основан на концепции Atom.

Клиент использует HTTP-метод GET для запроса к *текущему журналу уведомлений* (current log). Этот журнал содержит последние уведомления, которые были опубликованы. Клиент получает текущий журнал уведомлений, количество уведомлений в котором не превышает стандартного предела. В нашем примере максимальное количество уведомлений в каждом журнале событий равно 20. Клиент

[2] См. обсуждение этих моделей в рамках шаблона НАБЛЮДАТЕЛЬ (OBSERVER) на веб-странице `http://c2.com/cgi/wiki?ObserverPattern`.

просматривает текущий журнал уведомлений в поисках СОБЫТИЙ, которые еще не были получены ОГРАНИЧЕННЫМ КОНТЕКСТОМ.

Как локальный клиент получает уведомления о СОБЫТИЯХ? Он интерпретирует сериализованное СОБЫТИЕ по типу, транслируя все постоянно релевантные данные как соответствующие локальному ОГРАНИЧЕННОМУ КОНТЕКСТУ. Возможно, при этом необходимо найти соответствующие экземпляры АГРЕГАТА в своей модели и выполнить команды, основываясь на интерпретации подходящих событий. Разумеется, СОБЫТИЯ должны применяться в хронологическом порядке, поскольку самые старые СОБЫТИЯ представляют операции, которые были выполнены раньше других. Если самые старые СОБЫТИЯ не применялись бы первыми в порядке, в котором они возникали, изменения в локальной модели могли бы спровоцировать ошибки.

В нашей реализации текущий журнал содержит не больше 19 уведомлений. Он может содержать меньшее количество уведомлений и даже не содержать ни одного уведомления. Когда количество уведомлений в текущем журнале достигает 20, он автоматически архивируется. Если за время архивирования не пришли новые уведомления, текущий журнал уведомлений окажется пустым.

Что такое заархивированный журнал уведомлений

Архивирование журнала уведомлений не представляет собой ничего таинственного. Это просто значит, что конкретный журнал больше не изменяется никакими действиями, происходящими в системе, а клиенты могут обращаться к заархивированному журналу сколько угодно раз и при этом он всегда будет неизменным.

С другой стороны, текущий журнал уведомлений будет изменяться вплоть до его заполнения, после чего он тоже будет заархивирован. Единственным изменением, которое может происходить в текущем журнале, является добавление нового уведомления.

Уведомления, ранее добавленные в журнал уведомлений, никогда не изменяются, поскольку клиенты должны гарантировать, что локальное применение СОБЫТИЯ происходит только один раз.

Таким образом, текущий журнал уведомлений может не всегда содержать новейшее или самое старое уведомление о событии, которое все же должно применяться локально. Самое старое СОБЫТИЕ может находиться в предыдущем текущем журнале и даже в более ранних журналах. Это лишь вопрос выбора времени, зависящий от того, насколько часто СОБЫТИЯ заполняют данный журнал (в нашем случае он может содержать только 20 записей) и как часто клиенты обращаются к этому журналу. На рис. 8.4 показано, что журналы уведомлений образуют виртуальный массив отдельных уведомлений.

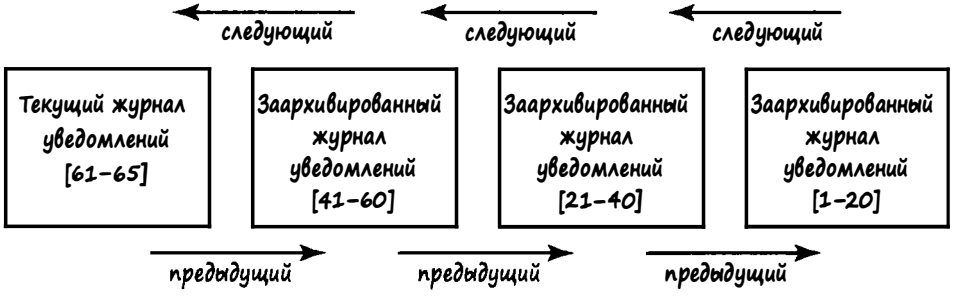


Рис. 8.4. Текущий журнал и любое количество связанных с них заархивированных журналов образуют виртуальный массив *всех* событий, от первого до последнего. Здесь показаны уведомления с номерами от 1 до 65. Каждый из заархивированных журналов полностью исчерпал лимит из 20 уведомлений. Текущий журнал событий еще не заполнен и содержит только пять уведомлений

Допустим, что журнал выглядит так, как показано на рис. 8.4, и предположим, что уведомления 1–58 уже были применены локально. Это означает, что уведомления 59–65 еще не были применены. Если клиент запросит следующий URI, то он получит текущий журнал.

```
//iam/notifications
```

Клиент читает из своей базы данных запись слежения, содержащую идентификационные данные последнего примененного уведомления, которым в нашем примере является уведомление 58. Задача отслеживания следующего уведомления, которое должно применяться, возлагается на клиент, а не на сервер. Клиент просматривает текущий журнал сверху вниз в поисках уведомления с идентификационным номером 58. Он не находит его там и поэтому продолжает перемещаться к предыдущему журналу, который является заархивированным. Доступ к предыдущему журналу обеспечивается с помощью ссылки гипермедиа в текущем журнале. Например, можно включить в заголовок путь навигации по гипермедиа.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.saasovation.idovation+json
...
Link: <http://iam/notifications/61,80>; rel=self
Link: <http://iam/notifications/41,60>; rel=previous
...
```

Почему идентификатор URI не отражает реальную информацию о содержимом текущего журнала

Обратите внимание на то, что в текущем журнале присутствуют только уведомления с номерами от 61 до 65, но несмотря на это его идентификатор URI содержит весь диапазон идентификаторов, например от 61 до 80.

```
Link: <http://iam/notifications/61,80>; rel=self
```

Это объясняется тем, что ресурс должен оставаться устойчивым на всем протяжении своего существования. Благодаря этому обеспечиваются согласованный доступ и правильное кеширование.

По ссылке `Link`, содержащей `rel=previous`, определяется идентификатор URI, который используется в методе `GET`, извлекающем предыдущий журнал уведомлений.

```
//iam/notifications/41,60
```

Теперь, используя этот заархивированный журнал, клиент находит искомое уведомление с идентификационным номером 58 после трех проверок отдельных уведомлений (60, 59 и наконец 58). Так как этот клиент уже применил данное уведомление (с идентификационным номером 58), он не применяет уведомление 58 снова. Вместо этого он перемещается в другом направлении в поисках новых уведомлений. В этом заархивированном журнале он находит уведомление с идентификационным номером 59 и применяет его. Затем он находит уведомление 60 и применяет его. После этого он достигает вершины заархивированного журнала и, таким образом, перемещается к ресурсу `rel=next`, который является текущим журналом.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.saasovation.idovation+json
...
Link: <http://iam/notifications/61,80>; rel=next
Link: <http://iam/notifications/41,60>; rel=self
Link: <http://iam/notifications/21,40>; rel=previous
...
```

Клиент находит в этом журнале уведомления с идентификационными номерами 61–65, применяя каждое в хронологическом порядке. Затем он достигает конца текущего журнала и прекращает обрабатывать уведомления на данный момент, потому что у текущего журнала никогда нет заголовка ссылки на ресурс `rel=next`.

Позднее процесс повторяется. Текущий журнал запрашивается с помощью идентификатора URI. Возможно, к текущему моменту времени действие в

исходном ОГРАНИЧЕННОМ КОНТЕКСТЕ уже вызвало генерацию существенно отличающихся журналов, произведя много новых уведомлений. Теперь, когда потребуется текущий журнал, в нем может быть любое количество новых уведомлений. Клиенту, вероятно, придется переместиться назад на один, два или более заархивированных журналов, чтобы найти последнее примененное уведомление, которое в настоящий момент имеет идентификационный номер 65. Как и прежде, когда клиент найдет уведомление 65, он применит все более новые уведомления в хронологическом порядке.

К журналам уведомлений может обращаться любое количество разных клиентов. Фактически любой ОГРАНИЧЕННЫЙ КОНТЕКСТ, который должен знать, какие СОБЫТИЯ были произведены любым другим ОГРАНИЧЕННЫМ КОНТЕКСТОМ, содержащим издателя уведомлений данного типа, может получить уведомления от "начала времен". Конечно, каждый клиентский ОГРАНИЧЕННЫЙ КОНТЕКСТ может на самом деле быть клиентом, только если у него есть надлежащий доступ к исходной системе (например, права безопасности).

Но не будет ли клиентский опрос ресурсов уведомлений вызывать огромные объемы нежелательного трафика на вашем веб-сервере? Нет, если ваши ресурсы RESTful эффективно используют кеширование. Например, текущий журнал мог бы кешироваться самим клиентом в течение приблизительно одной минуты.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.saasovation.idovation+json
...
Cache-Control: max-age=60
...
```

Каждый раз, когда запросу клиента предшествует одноминутное кеширование, сам клиентский кеш предоставляет ранее полученный текущий журнал. Когда время жизни кеша истекает, последний текущий журнал может быть получен от серверного ресурса. Заархивированные журналы могут кешироваться дольше, так как их содержимое никогда не изменяется, что демонстрируется следующим одночасовым значением переменной `max-age`.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.saasovation.idovation+json
...
Cache-Control: max-age=3600
...
```

Клиент может использовать текущее значение `max-age` журнала в качестве порога таймера/сна, поскольку непрерывно выполнять запросы с помощью метода GET на кешируемых ресурсах необязательно. Вызванное сном уменьшение

интенсивности запросов может снизить нагрузку на клиентский ОГРАНИЧЕННЫЙ КОНТЕКСТ и на исходный сервер. Провайдер ресурсов никогда не будет получать запросы, пока не будет достигнуто значение `max-age` кеша. Таким образом, злонамеренный клиент никогда не сможет ухудшить производительность или доступность источника уведомления при условии надлежащего использования клиентского кеширования. Это подчеркивает преимущества использования сети и ее встроенной инфраструктуры для достижения огромной производительности и высокой гибкости.

Сервер может также обеспечивать собственный кеш. Кеширование журнала уведомлений на сервере действительно является очень эффективным, потому что содержание заархивированных журналов никогда не изменяется. Любой клиент, запрашивающий данное заархивированное уведомление, не только получает ресурс, но и подготавливает кеш для всех других клиентов, нуждающихся в том же ресурсе. Для того чтобы обновить заархивированный журнал, кеш совершенно не нужен, потому что неизменность журнала гарантируется.

Мы всего лишь коснулись данной темы. Более подробно она будет рассмотрена в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**. Я предлагаю читателям обратиться к книге [Parastatidis et al., RiP], в которой описаны различные стратегии разработки эффективных систем уведомления о СОБЫТИЯХ на основе стиля RESTful. Там вы найдете обсуждение преимуществ и недостатков стандартных журналов уведомлений на основе веб-каналов Atom, а также несколько образцов реализации. Кроме того, Джим Уэббер (Jim Webber) еще более углубил понимание этого подхода в своей презентации [Webber, REST & DDD]. Одна из самых ранних ссылок на этот подход восходит к статье Стефана Тилкова (Stefan Tilkov) об InfoQ [Tilkov, RESTful Doubts]. Кроме того, вы можете посмотреть мою личную презентацию на основе этого подхода [Vernon, RESTful DDD].

Публикация уведомлений с помощью промежуточного программного обеспечения для обмена сообщениями

Не удивительно, что промежуточное программное обеспечение для обмена сообщениями, такое как RabbitMQ, снимает с вас обязанность самостоятельно разбираться с деталями, возложенную на вас стилем REST. Система обмена сообщениями также позволяет довольно легко поддерживать шаблоны ИЗДАТЕЛЬ–ПОДПИСЧИК и ОЧЕРЕДИ, в зависимости от того, какой из них лучше удовлетворяет ваши потребности. В обоих случаях система обмена сообщениями использует модель проталкивания, чтобы отправить сообщения о СОБЫТИЯХ зарегистрированным подписчикам или слушателям.

Рассмотрим требования к публикации СОБЫТИЙ из ХРАНИЛИЩА СОБЫТИЙ с помощью промежуточного программного обеспечения для обмена сообщениями.

Мы будем придерживаться шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК, используя подход, который в контексте программного обеспечения RabbitMQ называется *веерной рассылкой* (fanout exchange). Нам потребуется набор компонентов, которые работают вместе для выполнения следующих действий.

1. Запросить все те объекты СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ из ХРАНИЛИЩА СОБЫТИЙ, которые еще не были опубликованы в рамках конкретной рассылки. Упорядочить запрошенные объекты в порядке возрастания их уникальных идентификаторов.
2. Выполнить итерации по запрошенным объектам в порядке возрастания, отправляя каждый из них в рассылку.
3. Когда система обмена сообщениями указывает, что сообщение было успешно опубликовано, отследить то СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, которое было опубликовано в рамках данной рассылки.

Мы не ждем подтверждения получения сообщения от подписчиков. Абонентские системы даже могут вообще не работать, когда издатель отправляет сообщения посредством рассылки. Каждый подписчик отвечает за обработку сообщений в рамках собственного периода времени, гарантируя, что он должным образом учитывает все необходимые функции предметной области в собственной модели. Мы просто позволяем механизму обмена сообщениями гарантировать доставку.

Заметки на доске

- Нарисуйте КАРТУ КОНТЕКСТА для ОГРАНИЧЕННОГО КОНТЕКСТА, в котором вы работаете и который вы хотите интегрировать с остальными. Убедитесь, что вы показали связи между взаимодействующими КОНТЕКСТАМИ.
- Обозначьте отношения между ними, например **ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (3)**.
- Покажите, как бы вы интегрировали эти КОНТЕКСТЫ. Стали бы вы использовать обозначения стилей RPC и RESTful или инфраструктуры обмена сообщениями? Нарисуйте их.

Помните, что при интеграции с устаревшими системами у вас может быть небольшой выбор.

Реализация

Выбрав архитектурные стили для публикации СОБЫТИЙ, команда SaaSOvation сосредоточилась на реализации соответствующих компонентов...

Ядро функции, публикующей уведомления, размещается позади ПРИКЛАДНОЙ СЛУЖБЫ `NotificationService`. Это позволило команде управлять изменениями в источнике данных, происходящими в процессе транзакции. Кроме того, упор был сделан на прикладных аспектах, а не на прикладной области, несмотря на то что СОБЫТИЯ, издаваемые как уведомления, происходят внутри модели.

ПРИКЛАДНАЯ СЛУЖБА `NotificationService` не обязана иметь **выделенный интерфейс** [Fowler, P of EAA]. Пока мы рассматриваем одну реализацию ПРИКЛАДНОЙ СЛУЖБЫ, поэтому все следует делать как можно проще. В то же время каждый простой класс имеет открытый интерфейс, который ниже представлен фиктивными методами.

```
package com.saasovation.identityaccess.application;
...
public class NotificationService {
    ...
    @Transactional(readOnly=true)
    public NotificationLog currentNotificationLog() {
        ...
    }

    @Transactional(readOnly=true)
    public NotificationLog notificationLog(String aNotificationLogId) {
        ...
    }

    @Transactional
    public void publishNotifications() {
        ...
    }
    ...
}
```

Первые два метода будут использоваться для обращения к экземплярам класса `NotificationLog`, которые предоставлены клиентам как ресурсы RESTful, а третий будет использоваться для публикации отдельных экземпляров класса `Notification` с помощью механизма обмена сообщениями. Команда решила сначала разработать методы запроса для получения экземпляров класса `NotificationLog`, а

затем обратить внимание на методы, которые взаимодействуют с инфраструктурой обмена сообщениями.

Некоторые из интересных реализаций мы вскоре рассмотрим.

Публикация экземпляров класса NotificationLog

Напомним, что существует два вида журналов уведомлений — текущий и заархивированные. Таким образом, интерфейс класса NotificationService должен предоставлять метод запроса для каждого вида журнала.

```
public class NotificationService {
    @Transactional(readOnly=true)
    public NotificationLog currentNotificationLog() {
        EventStore eventStore = EventStore.instance();

        return this.findNotificationLog(
               this.calculateCurrentNotificationLogId(eventStore),
               eventStore);
    }

    @Transactional(readOnly=true)
    public NotificationLog notificationLog(String aNotificationLogId) {
        EventStore eventStore = EventStore.instance();

    return this.findNotificationLog(
        new NotificationLogId(aNotificationLogId),
        eventStore);
    }
    ...
}
```

В результате оба эти метода должны найти экземпляр класса NotificationLog. Фактически это означает обнаружение раздела экземпляров класса DomainEvent, сериализованных в ХРАНИЛИЩЕ СОБЫТИЙ и инкапсулированных в экземпляре класса Notification, а также их сборку в экземпляр класса NotificationLog. Когда экземпляр класса NotificationLog создан, его можно представить как ресурс RESTful и предоставить запрашивающему клиенту.

Поскольку текущий журнал может быть постоянно перемещающейся целью, его идентичность следует вычислять при каждом запросе. Вот как выглядит это вычисление.

```
public class NotificationService {
    ...
    protected NotificationLogId calculateCurrentNotificationLogId(
            EventStore anEventStore) {
```

```
        long count = anEventStore.countStoredEvents();

        long remainder = count % LOG_NOTIFICATION_COUNT;

        if (remainder == 0) {
            remainder = LOG_NOTIFICATION_COUNT;
        }

        long low = count - remainder + 1;

        // гарантирует создание идентификатора,
        // даже если набор уведомлений пока не полон

        long high = low + LOG_NOTIFICATION_COUNT - 1;

        return new NotificationLogId(low, high);

    }
    ...
}
```

В других вариантах для архивированного журнала достаточно, чтобы экземпляр класса `NotificationLogId` инкапсулировал нижний и верхний пределы идентификатора. Напомним, что идентификатор кодируется как текстовое представление числа в диапазоне от нижнего до верхнего предела, например 21–40. Таким образом, конструктор для закодированного идентификатора может выглядеть следующим образом.

```
public class NotificationLogId {
    ...
    public NotificationLogId(String aNotificationLogId) {
        super();
        String[] textIds = aNotificationLogId.split(",");
        this.setLow(Long.parseLong(textIds[0]));
        this.setHigh(Long.parseLong(textIds[1]));
    }
    ...
}
```

К какому бы журналу мы ни обращались — текущему или заархивированному, — теперь нужен экземпляр класса `NotificationLogId`, описывающий метод `findNotificationLog()`, который будет выполнять запрос.

```
public class NotificationService {
    ...
    protected NotificationLog findNotificationLog(
         NotificationLogId aNotificationLogId,
         EventStore anEventStore) {
```

```
    List<StoredEvent> storedEvents =
         anEventStore.allStoredEventsBetween(
                aNotificationLogId.low(),
                aNotificationLogId.high());

    long count = anEventStore.countStoredEvents();

    boolean archivedIndicator = aNotificationLogId.high() < count;

    NotificationLog notificationLog =
       new NotificationLog(
                aNotificationLogId.encoded(),
                NotificationLogId.encoded(
                       aNotificationLogId.next(
                              LOG_NOTIFICATION_COUNT)),
                NotificationLogId.encoded(
                       aNotificationLogId.previous(
                              LOG_NOTIFICATION_COUNT)),
                this.notificationsFrom(storedEvents),
                archivedIndicator);

    return notificationLog;
 }
 ...
    protected List<Notification> notificationsFrom(
           List<StoredEvent> aStoredEvents) {
       List<Notification> notifications =
           new ArrayList<Notification>(aStoredEvents.size());

       for (StoredEvent storedEvent : aStoredEvents) {
            DomainEvent domainEvent =
                  EventStore.toDomainEvent(storedEvent);

       Notification notification =
            new Notification(
                  domainEvent.getClass().getSimpleName(),
                  storedEvent.eventId(),
                  domainEvent.occurredOn(),
                  domainEvent);

       notifications.add(notification);
    }

       return notifications;
    }
    ...
}
```

Довольно интересно, что постоянно хранить экземпляры класса `Notification` и все журналы целиком необязательно. Их можно создавать каждый раз, когда они нужны. Очевидно, что это повышает производительность и гибкость кеширования ресурсов `NotificationLog` в точке запроса.

Метод `findNotificationLog()` использует компонент `EventStore` для запроса экземпляров класса `StoredEvent`, необходимых для заданного журнала. Вот как класс `EventStore` находит эти экземпляры.

```
package com.saasovation.identityaccess.application.eventStore;
...
public class EventStore ... {
    ...
    public List<StoredEvent> allStoredEventsBetween(
            long aLowStoredEventId,
            long aHighStoredEventId) {

       Query query =
            this.session().createQuery(
                    "from StoredEvent as _obj_ "
                    + "where _obj_.eventId between ? and ? "
                    + "order by _obj_.eventId");

       query.setParameter(0, aLowStoredEventId);
       query.setParameter(1, aHighStoredEventId);

       List<StoredEvent> storedEvents = query.list();

       return storedEvents;
    }
    ...
}
```

В заключение веб-уровень публикует текущий и заархивированные журналы.

```
@Path("/notifications")
public class NotificationResource {
    ...
    @GET
    @Produces({ OvationsMediaType.NAME })
    public Response getCurrentNotificationLog(
            @Context UriInfo aUriInfo) {

    NotificationLog currentNotificationLog =
             this.notificationService()
                 .currentNotificationLog();

    if (currentNotificationLog == null) {
        throw new WebApplicationException(
                   Response.Status.NOT_FOUND);
    }

    Response response =
        this.currentNotificationLogResponse(
                currentNotificationLog,
                aUriInfo);
```

```
        return response;
    }

    @GET
    @Path("{notificationId}")
    @Produces({ OvationsMediaType.ID_OVATION_NAME })
    public Response getNotificationLog(
            @PathParam("notificationId") String aNotificationId,
            @Context UriInfo aUriInfo) {

        NotificationLog notificationLog =
            this.notificationService()
                .notificationLog(aNotificationId);

        if (notificationLog == null) {
            throw new WebApplicationException(
                Response.Status.NOT_FOUND);
        }

        Response response =
            this.notificationLogResponse(
                notificationLog,
                aUriInfo);

        return response;
    }
    ...
}
```

Для генерирования ответа команда могла бы использовать экземпляр класса `MessageBodyWriter`, но это усложнило бы методы.

Перейдем к описанию важных компонентов, которые используются для публикации текущего и заархивированных журналов уведомлений для клиентов RESTful.

Публикация уведомлений на основе сообщений

Класс `NotificationService` содержит отдельный метод для публикации экземпляров класса `DomainEvent` с помощью инфраструктуры передачи сообщений. Рассмотрим этот вспомогательный метод.

```
public class NotificationService {
    ...
    @Transactional
    public void publishNotifications() {
        PublishedMessageTracker publishedMessageTracker =
            this.publishedMessageTracker();

        List<Notification> notifications =
            this.listUnpublishedNotifications(
```

```
                    publishedMessageTracker
                        .mostRecentPublishedMessageId());

        MessageProducer messageProducer = this.messageProducer();

        try {
            for (Notification notification : notifications) {
                this.publish(notification, messageProducer);
            }

            this.trackMostRecentPublishedMessage(
                publishedMessageTracker,
                notifications);
        } finally {
            messageProducer.close();
        }
    }
    ...

}
```

Метод `publishNotifications()` сначала получает экземпляр класса `PublishedMessageTracker`. Это объект, в котором хранится запись о том, какие СОБЫТИЯ уже были опубликованы.

```
package com.saasovation.identityaccess.application.notifications;
...
public class PublishedMessageTracker {
    private long mostRecentPublishedMessageId;
    private long trackerId;
    private String type;
    ...
}
```

Обратите внимание на то, что этот класс не является частью модели предметной области, а принадлежит приложению. Атрибут `trackerId` — это всего лишь уникальный идентификатор объекта (по сути, СУЩНОСТЬ). Атрибут `type` содержит описание в виде объекта класса `String` типа темы или канала, в рамках которого было опубликовано СОБЫТИЕ. Атрибут `mostRecentPublishedMessageId` соответствует уникальному идентификатору индивидуального экземпляра класса `DomainEvent`, который был сериализован и сохранен как экземпляр класса `StoreEvent`. Таким образом, он хранит значение атрибута `StoredEventeventId` последнего опубликованного события. Кроме того, служебный метод гарантирует, что после поступления новых сообщений типа `Notification` экземпляр класса `PublishedMessageTracker` будет сохранен вместе с идентификатором последнего опубликованного СОБЫТИЯ.

Идентификатор СОБЫТИЯ вместе с атрибутом type *позволяет публиковать одни и те же уведомления разное количество раз для любого количества тем и каналов.* Мы просто создаем новый экземпляр класса PublishedMessageTracker с именем темы или канала в качестве его атрибута type и начинаем работу снова с первого экземпляра класса StoredEvent. Вот как это делает метод publishedMessageTracker().

```
public class NotificationService {
    private static final String EXCHANGE_NAME =
        "saasovation.identity_access";
        ...
    private PublishedMessageTracker publishedMessageTracker() {
        Query query =
            this.session().createQuery(
                  "from PublishedMessageTracker as _obj_ "
                  + "where _obj_.type = ?");

        query.setParameter(0, EXCHANGE_NAME);

        PublishedMessageTracker publishedMessageTracker =
           (PublishedMessageTracker) query.uniqueResult();

        if (publishedMessageTracker == null) {
           publishedMessageTracker =
              new PublishedMessageTracker(EXCHANGE_NAME);
        }

        return publishedMessageTracker;
    }
    ...
}
```

Многоканальная публикация пока не поддерживается, но ее можно легко добавить с помощью небольшого рефакторинга.

Далее, метод listUnpublishedNotifications() отвечает за обращение к упорядоченному списку, содержащему все неопубликованные экземпляры класса Notification.

```
public class NotificationService {
    ...
    protected List<Notification> listUnpublishedNotifications(
          long aMostRecentPublishedMessageId) {
       EventStore eventStore = EventStore.instance();

       List<StoredEvent> storedEvents =
             eventStore.allStoredEventsSince(
                  aMostRecentPublishedMessageId);

       List<Notification> notifications =
```

```
            this.notificationsFrom(storedEvents);

        return notifications;
    }
    ...
}
```

На самом деле он запрашивает у экземпляра класса `EventStore` экземпляры класса `StoredEvent`, содержащие значения `eventId`, превышающие значение, заданное параметром `aMostRecentPublishedMessageId`. Объекты, полученные из экземпляра класса `EventStore`, используются для создания новой коллекции экземпляров класса `Notification`.

Вернемся к основному вспомогательному методу `publishNotifications()`. Он просматривает коллекции объектов `DomainEvent`, содержащиеся в экземплярах класса `Notification` и вызывает метод `publish()`.

```
...
for (Notification notification : notifications) {
     this.publish(notification, messageProducer);
}
```

Это метод, публикующий отдельные экземпляры класса `Notification` с помощью системы RabbitMQ, используя при этом очень простую библиотеку объектов для создания объектно-ориентированного интерфейса.

```
public class NotificationService {
    ...
    protected void publish(
          Notification aNotification,
          MessageProducer aMessageProducer) {

       MessageParameters messageParameters =
           MessageParameters.durableTextParameters(
                  aNotification.type(),
                  Long.toString(aNotification.notificationId()),
                  aNotification.occurredOn());

       String notification =
           NotificationService
                  .objectSerializer()
                  .serialize(aNotification);

       aMessageProducer.send(notification, messageParameters);
    }
    ...
}
```

Метод publish() создает экземпляр класса MessageParameters, а затем посылает объект DomainEvent, сериализованный в формате JSON с помощью экземпляра класса MessageProducer.[3] Класс MessageParameters содержит свойства, которые посылаются вместе с телом сообщения. Эти специальные параметры включают в себя строку type, описывающую тип СОБЫТИЯ, идентификатор уведомления, используемый в качестве уникального идентификатора сообщения, а также временную метку СОБЫТИЯ occurredOn. Эти параметры позволяют подписчикам распознавать важные факты о каждом событии без лексического разбора тела сообщения в формате JSON, которое представляет собой сериализованное СОБЫТИЕ. Как будет показано далее, уникальный идентификатор сообщения (идентификатор уведомления) допускает дедубликацию.

Рассмотрим еще один метод для полной реализации публикации.

```
public class NotificationService {
    ...
    private MessageProducer messageProducer() {

       // создаем рассылку, если она не существовала
       Exchange exchange =
           Exchange.fanOutInstance(
                    ConnectionSettings.instance(),
                    EXCHANGE_NAME,
                    true);

        // создает источник сообщения, который используется
        // для пересылки СОБЫТИЙ
        MessageProducer messageProducer =
            MessageProducer.instance(exchange);

        return messageProducer;
    }
    ...

}
```

Метод publishNotifications() использует метод messageProducer(), чтобы гарантировать, что рассылка существует и получает экземпляр класса MessageProducer, используемый для публикации сообщений. Система RabbitMQ поддерживает идемпотентность рассылки. Таким образом, будучи один раз созданной, она каждый раз будет предоставляться вам как ранее существовавшая. Мы не сохраняем открытый экземпляр класса MessageProducer в

[3] Классы Exchange, ConnectionSettings, MessageProducer, MessageParameters и другие являются частью библиотеки, образующей уровень абстракции над системой RabbitMQ. Эту библиотеку, облегчающую использование системы RabbitMQ, а также примеры из книги можно найти на веб-странице автора.

случае разрыва соединения. Восстановление соединения при каждой публикации помогает предотвращать абсолютную недееспособность издателя. Если постоянное восстановление соединения становится узким местом, возможно, потребуется решить вопросы, связанные с производительностью. Однако пока мы будем рассчитывать на такую конфигурацию пауз между публикациями, которая снижает затраты на восстановление соединения.

Говоря о паузах между публикациями, следует иметь в виду, что приведенный выше код никак не указывает на то, как именно СОБЫТИЯ публикуются для рассылки на регулярной, повторяющейся основе. Эта операция может быть выполнена несколькими различными способами и может зависеть от вашей операционной среды. Например, для управления повторяющимися временными интервалами может использоваться класс `TimerMBean` из каркаса JMX.

Прежде чем представить следующее решение для таймера, необходимо отметить важный контекст. Стандарт Java MBean также использует термин *уведомление*, но это не то уведомление, которое используется в нашем процессе публикации. В этом случае слушатель получает уведомление о каждом срабатывании таймера. Просто будьте готовы учесть это обстоятельство.

Независимо от того, как определен и сконфигурирован подходящий интервал для данного таймера, экземпляр класса `NotificationListener` регистрируется так, что класс `MBeanServer` может выдать уведомление о каждом случае, когда достигается конец интервала.

```
mbeanServer.addNotificationListener(
       timer.getObjectName(),
       new NotificationListener() {
           public void handleNotification(
                  Notification aTimerNotification,
                  Object aHandback) {
              ApplicationServiceRegistry
                        .notificationService()
                        .publishNotifications();
           }
       },
       null,
       null);
```

В данном примере, когда метод `handleNotification()` вызывается из-за срабатывания таймера, он просит объект класса `NotificationService` выполнить свой метод `publishNotifications()`. Это все, что необходимо. Поскольку объект класса `TimerMBean` продолжает срабатывать через регулярные, повторяющиеся интервалы времени, СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ будут продолжать публиковаться с помощью рассылки и использоваться подписчиками по всему предприятию.

Использование таймера, управляемого прикладным сервером, имеет дополнительное преимущество, которое заключается в том, что вы не должны создавать компонент, чтобы контролировать жизненный цикл вашего процесса публикации сообщений. Если бы, например, метод `publishNotifications()` по какой-то причине столкнулся бы с какими-то проблемами и должен был бы прекратить выполнение, сгенерировав исключение, то объект класса `TimerMBean` продолжал бы работать и срабатывать на последующих интервалах времени. Администраторы, возможно, должны были бы исправить ошибки инфраструктуры, например, с помощью системы RabbitMQ, но после устранения проблем сообщения продолжали бы публиковаться. Впрочем, существуют и другие способы создания таймеров, такие как технология Quartz [Quartz].

Однако все еще остаются нерешенными вопросы дедупликации сообщений. Что такое дедупликация сообщения? И почему она необходима для рассылки сообщений подписчикам?

Дедубликация СОБЫТИЯ. Дедупликация необходима в средах, где единственное сообщение, опубликованное через систему обмена сообщениями, может быть передано подписчикам несколько раз. Есть различные причины возникновения дубликатов сообщений. Например, это может произойти следующим образом.

1. Система RabbitMQ доставляет новые сообщения одному или нескольким подписчикам.
2. Подписчики обрабатывают сообщения.
3. Прежде чем подписчики поймут, что эти сообщения уже были получены и подтверждены, происходит сбой.
4. Система RabbitMQ рассылает неподтвержденные сообщения снова.

Существует также возможность, что при публикации сообщений из ХРАНИЛИЩА СОБЫТИЙ система обмена сообщениями не использует совместный механизм постоянного хранения, на котором основано ХРАНИЛИЩЕ СОБЫТИЙ, а глобальные транзакции XA не управляют атомарными фиксациями в ХРАНИЛИЩЕ СОБЫТИЙ и происходящими там изменениями. Ранее в разделе "Публикация уведомления с помощью промежуточного программного обеспечения для обмена сообщениями" была описана точно такая ситуация. Рассмотрим сценарий, в котором сообщение могло быть отправлено несколько раз.

1. Экземпляр класса `NotificationService` запрашивает из хранилища и публикует три ранее неопубликованных экземпляра класса `Notification`. Он обновляет запись об этом с помощью экземпляра класса `PublishedMessageTracker`.
2. Брокер системы RabbitMQ получает все три сообщения и готовит их для рассылки всем подписчикам.

3. Однако из-за исключительных обстоятельств, возникших на сервере приложения, в классе `NotificationService` происходит сбой. Модификация объекта класса `PublishedMessageTracker` не подтверждается.
4. Система RabbitMQ доставляет подписчикам вновь посланные сообщения.
5. Исключительные обстоятельства на сервере приложения устраняются. Процесс публикации начинается снова и объект класса `NotificationService` успешно посылает сообщения обо всех неопубликованных СОБЫТИЯХ. К ним относятся (снова!) сообщения обо всех СОБЫТИЯХ, которые были опубликованы ранее, но не известны экземпляру класса `PublishedMessageTracker`.
6. Система RabbitMQ доставляет подписчикам вновь посланные сообщения, среди которых как минимум три представляют собой дубликаты.

В этом сценарии число "три" выбрано произвольно. Событий может быть сколько угодно. Их количество не имеет значения. Важен лишь факт, что в таких ситуациях может возникать повторная доставка. Когда вы сталкиваетесь с этой и подобными ситуациями, возникает необходимость в дедубликации. Более подробно эта тема изложена при описании шаблона **ИДЕМПОТЕНТНЫЙ ПОЛУЧАТЕЛЬ (IDEMPOTENT RECEIVER)** в работе [Hohpe & Woolf].

Идемпотентная операция

Идемпотентная операция — это операция, которая может выполняться несколько раз, но при этом все результаты идентичны результату ее первого применения.

Один из возможных способов решения проблемы двойной доставки сообщений — сделать операции подписчика идемпотентными. Ответ подписчика на все сообщения мог бы быть идемпотентной операцией по отношению к своей модели предметной области. Проблема состоит в том, что разработка идемпотентного объекта предметной области или любого аналогичного объекта может оказаться трудной, непрактичной или даже невозможной. Попытка так разработать СОБЫТИЕ, чтобы оно само несло информацию об идемпотентной операции, также может быть проблематичной. Например, отправитель должен полностью понимать текущую бизнес-ситуацию всех получателей относительно состояния СОБЫТИЯ, которое они получат. Кроме того, получение СОБЫТИЙ, которые выпали из расписания из-за задержки, повторений и других событий, также могло бы вызвать ошибки.

Когда идемпотентность объекта предметной области обеспечить невозможно, можно вместо этого разработать подписчика/получателя так, чтобы она сам был идемпотентным. Получателя можно спроектировать так, чтобы он отказывался выполнять работу в ответ на получение дубликата сообщения. Прежде всего

необходимо проверить, поддерживает ли ваше программное обеспечение такую функцию. В противном случае получатель должен будет сам отслеживать, какие сообщения уже были обработаны. Например, можно выделить область в механизме постоянного хранения подписчика для хранения имени темы/рассылки вместе с уникальным идентификатором всех обработанных сообщений, аналогичным классу `PublishedMessageTracker`. Тогда вы сможете запрашивать дубликаты прежде, чем обрабатывать каждое сообщение. Если обнаружится, что сообщение уже было обработано, то подписчик просто проигнорирует его. Отслеживание обработанных сообщений не является частью модели предметной области. Его следует рассматривать только как техническое обходное решение для устранения проблем, связанных с механизмом обмена сообщениями.

При использовании типичного промежуточного программного обеспечения для обмена сообщениями недостаточно сохранять только запись о последнем обработанном сообщении, потому что сообщения могут быть получены не по порядку. Таким образом, запрос дедупликации, который проверяет все идентификаторы сообщений, не превышающие новый, заставил бы вас игнорировать некоторые сообщения, которые были получены не по порядку. Кроме того, следует учитывать, что иногда вы будете отбрасывать все обработанные сообщения, отслеживающие устаревшие записи, как в механизме сборки мусора в базе данных.

В рамках подхода к рассылке уведомлений, основанного на стиле REST, дедубликация не имеет значения. Клиентские получатели должны сохранить только идентификатор самого последнего примененного уведомления, так как они всегда будут применять только уведомления о событиях, которые произошли после них. Каждый журнал уведомлений всегда будет содержать идентификаторы уведомлений в обратном хронологическом порядке (убывающем).

В обоих случаях — когда подписчики используют промежуточное программное обеспечение для обмена сообщениями и когда клиенты уведомлений используют стиль REST — важно, чтобы отслеживание идентификационных данных обработанных сообщений фиксировалось вместе с любыми изменениями в локальном состоянии модели предметной области. В противном случае вы будете неспособны обеспечить непротиворечивость согласованности отслеживания с модификациями, сделанными в ответ на `СОБЫТИЯ`.

Резюме

В главе рассмотрена концепция СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ и показано, в каких ситуациях может оказаться полезным моделирование СОБЫТИЙ.

- Вы узнали, что такое СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, а также когда и почему их следует использовать.
- Вы увидели, как можно моделировать СОБЫТИЯ в виде объектов и когда их следует однозначно идентифицировать.
- Вы узнали, когда СОБЫТИЕ должно обладать характеристиками АГРЕГАТА, а когда лучше, чтобы событие представляло собой простое значение.
- Вы увидели, как в модели используются компоненты шаблона ИЗДАТЕЛЬ–ПОДПИСЧИК.
- Вы поняли, какие компоненты публикуют СОБЫТИЯ, а какие являются их подписчиками.
- Вы узнали, почему необходимо проектировать ХРАНИЛИЩЕ СОБЫТИЙ, как это можно сделать и как его можно использовать.
- Вы освоили два подхода к публикации СОБЫТИЙ за пределами ОГРАНИЧЕННОГО КОНТЕКСТА: с помощью уведомлений на основе стиля REST и с помощью промежуточного программного обеспечения для обмена сообщениями.
- Вы узнали несколько способов дедубликации сообщений в системах подписчиков.

Далее мы немного изменим направление изложения и посмотрим, как организовать объекты предметной области с помощью МОДУЛЕЙ.

Глава 9

Модули

Секрет победы лежит в организации неочевидного.
Марк Аврелий

Если вы используете языки Java и C#, то уже знакомы с понятием **МОДУЛЕЙ**, хотя они и называются иначе. В языке Java они называются пакетами, а в C# — пространствами имен. Как ни странно в языке Ruby вы можете использовать языковую конструкцию `module` для создания пространства имен для классов. В языке Ruby название шаблона DDD соответствует названию языковой конструкции. С учетом нашего контекста DDD в большинстве случаев я буду продолжать называть их МОДУЛЯМИ. Вам будет легко перевести это название в термины того языка, который вы регулярно используете. Я не буду тратить много времени, пытаясь формально объяснить, что делают МОДУЛИ, потому что вы, вероятно, уже давно это поняли.

Назначение главы

- Понять разницу между традиционными МОДУЛЯМИ и современным подходом к обеспечению модульности.
- Осознать важность выбора названий для МОДУЛЕЙ в соответствии с **ЕДИНЫМ ЯЗЫКОМ (1)**.
- Понять, что механическое проектирование МОДУЛЕЙ ограничивает возможности моделирования.
- Узнать о проектных решениями и компромиссах, принятых компанией SaaS-Ovation.
- Выяснить роль, которую МОДУЛИ играют за пределами модели предметной области, а также понять, когда следует выбирать новый МОДУЛЬ, а не новый ОГРАНИЧЕННЫЙ КОНТЕКСТ.

Разработка МОДУЛЕЙ

В контексте подхода DDD МОДУЛИ внутри модели являются именованными контейнерами для классов объектов предметной области, тесно связанных друг с

другом. Их цель — ослабление связей между классами, которые находятся в различных МОДУЛЯХ. Так как МОДУЛИ в подходе DDD — это не формальные или обобщенные разделы, их следует правильно называть. Выбор их имен является важной функцией ЕДИНОГО ЯЗЫКА.

> Выберите такие МОДУЛИ, которые бы рассказывали историю системы и содержали связные наборы понятий. От этого часто сама собой возникает низкая зависимость МОДУЛЕЙ друг от друга. Но если это не так, найдите способ изменить модель таким образом, чтобы отделить понятия друг от друга... Дайте модулям такие имена, которые войдут в ЕДИНЫЙ ЯЗЫК. Как сами МОДУЛИ, так и их имена должны отражать знание и понимание предметной области [Эванс, с. 112].

Несколько простых правил, которым необходимо следовать при проектировании МОДУЛЕЙ, приведены в табл. 9.1.

Таблица 9.1. Простые правила проектирования модулей

Что делает и чего не делает МОДУЛЬ	Причина
МОДУЛИ должны отражать концепции моделирования	Обычно МОДУЛЬ содержит один или несколько **АГРЕГАТОВ (10)**, связанных друг с другом хотя бы по ссылке
Имена МОДУЛЕЙ должны соответствовать ЕДИНОМУ ЯЗЫКУ	Это не только основная цель подхода DDD, но и вполне естественно при размышлении о моделируемых концепциях
Не создавайте МОДУЛИ механически в соответствии с общим типом компоненты или с шаблоном, используемым для моделирования	Модель ничего не выиграет, если, например, все АГРЕГАТЫ будут объединены в один МОДУЛЬ, все **СЛУЖБЫ (7)** — в другой, а все **ФАБРИКИ (11)** — в третий. Это не соответствует цели использования МОДУЛЕЙ в подходе DDD и ограничивает возможности моделирования. Вместо общих рассуждений о предметной области в целом вам следует рассматривать только виды компонентов или шаблоны, позволяющие решать текущие проблемы
Проектируйте слабо связанные МОДУЛИ	Обеспечение независимости МОДУЛЕЙ дает такие же преимущества, как и слабая связанность классов. Это облегчает поддержку и рефакторинг концепций моделирования и применение средств крупномасштабной модуляризации, таких как каркасы OSGi и Jigsaw

Окончание табл. 9.1

Что делает и чего не делает МОДУЛЬ	Причина
Если связанность необходима, то следует бороться за ациклические зависимости между одноранговыми МОДУЛЯМИ. (Одноранговыми называются МОДУЛИ, расположенные на одном и том же уровне или имеющие одинаковый вес или влияние на проект.)	На практике модули редко оказываются совершенно независимыми друг от друга. В конце концов, сама модель предметной области подразумевает определенную связанность. Впрочем, связанность компонентов можно уменьшить, если стремиться только к однонаправленной зависимости между одноранговыми модулями (например, продукция зависит от команды, но команда не зависит от продукции)
Ослабляйте правила, касающиеся дочерних и родительских МОДУЛЕЙ. (Родительский МОДУЛЬ — это МОДУЛЬ, находящийся на более высоком уровне, а дочерний МОДУЛЬ — это модуль, находящийся на более низком уровне, например `parent.child`.)	Избежать зависимостей между родительскими и дочерними МОДУЛЯМИ очень трудно. Если это возможно, стремитесь к ациклическим зависимостям между родительскими и дочерними МОДУЛЯМИ. Если же это невозможно, разрешайте циклические ссылки (например, родительский МОДУЛЬ создает ДОЧЕРНИЙ, при этом дочерний должен иметь ссылку на родительский МОДУЛЬ, хотя бы по идентификатору)
Не делайте МОДУЛИ статичными концепциями модели, разрешайте им изменяться в соответствии с объектами, которые они организовывают	Если концепции модели носят изменчивый характер, имеют разные формы, функции и имена, очень вероятно, что организовывающие их МОДУЛИ должны создаваться, переименовываться и удаляться в соответствии с концепциями. Это не обязательно, но желательно. Если увидите неправильное имя, выполните рефакторинг. Возможно, это трудно, но плохо названные МОДУЛИ еще хуже

МОДУЛИ следует считать основными компонентами модели. При их создании необходимо придавать им такой же смысл и имена, как **СУЩНОСТЯМ (5)**, **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**, **СЛУЖБАМ** и **СОБЫТИЯМ (8)**. При переименовании МОДУЛЕЙ следует проявлять такую же решительность, как и при создании новых. Решительно включайте в модель новые МОДУЛИ и обновляйте понятия предметной области, чтобы сохранить согласованность.

Ни один из нас не хотел бы, открыв ящик кухонного стола, увидеть мешанину вилок, ножей, ложек, ключей, отверток, розеток и молотков. В этом случае, даже если бы столовый набор оказался серебряным, вы, вероятно, отказались бы им пользоваться. Хотелось бы избежать необходимости рыться в таком ящике в поисках отвертки, рискуя порезаться ножом.

Сравните это с ящиком кухонного стола, в котором столовое серебро аккуратно рассортировано на вилки, ножи и ложки, и рабочим шкафчиком в вашем гараже, где у каждого типа инструмента есть собственная ниша. Мы могли бы без проблем найти то, что требуется в данный момент. Все хорошо организовано, не перепутано. Все на месте, никто не ожидает обнаружить чашки и блюдца в ящике со столовым серебром, несмотря на то что эти предметы являются кухонными принадлежностями. Аккуратные стопки столовой посуды, вероятно, убедили бы нас, что чашки и блюдца находятся на своих местах. Быстро проверив соседние отсеки, мы увидели бы, что и там все в порядке. Кроме того, мы хотели бы, чтобы режущие столовые приборы находились там, где о них невозможно порезаться, чтобы защитить тех, кто будет ими пользоваться.

С другой стороны, мы, вероятно, не стали бы организовывать свою кухню, используя механический подход, при котором все прочные предметы находятся в одной секции, а все хрупкие — в другом. Мы не хотели бы запоминать, что наши цветочные вазы хранятся вместе с нашими прекрасными чайными чашками просто потому, что эти предметы легко разбить. И при этом мы не хотели бы помнить, что мы храним молоток из нержавеющей стали вместе с прекрасными столовыми приборами просто потому, что эти предметы трудно повредить.

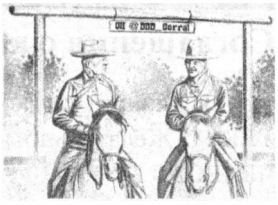
Для моделирования кухни естественно создать МОДУЛЬ с именем `placesettings` и включить в него объекты с именами `Fork` (Вилка), `Spoon` (Ложка) и `Knife` (Нож). Вероятно, мы даже решим включить в него объект `Serviette` (Салфетка), доказав тем самым, что объекты в МОДУЛЕ `placesettings` объединяет не то, что они созданы из металла. С другой стороны, было бы менее удобно моделировать столовые наборы, если в нашей модели есть разные модули с именами `pronged` (зубчатые), `scooping` (зачерпывающие) и `blunt` (тупые).

Обратите внимание на то, что современные успехи в области модуляризации программного обеспечения вывели нас на другой уровень модульности программного обеспечения. Этот подход позволяет упаковывать слабо связанные, но все же логически связные блоки программного обеспечения в модуль развертывания по их версиям. В среде Java мы все еще мыслим в терминах файлов JAR, но эти файлы должны быть собраны с учетом версий и модульной структуры OSGi или Jigsaw Java 8. Таким образом, различными высокоуровневыми модулями, их версиями и зависимостями можно управлять как пакетами/модулями. Эти виды модулей/пакетов несколько отличаются от МОДУЛЕЙ DDD, но могут дополнять друг друга. Конечно, имеет смысл связывать слабо связанные части модели предметной области в более крупные модули в соответствии с их МОДУЛЯМИ DDD. В конце концов, именно слабо связанный проект ваших МОДУЛЕЙ DDD поможет вам создать пакеты с помощью каркаса OSGi или построить модули с помощью каркаса Jigsaw.

Ковбойская логика

LB: Тебе, должно быть, интересно, почему на этой заправке туалет такой чистый.

AJ: Просто через заправку прошел торнадо и хозяевам пришлось вложить 10 тысяч долларов в ее ремонт.



Мы сосредоточимся на способах использования МОДУЛЕЙ DDD. Размышления о *предназначении конкретных* ОБЪЕКТОВ, ОБЪЕКТОВ-ЗНАЧЕНИЙ, СЛУЖБ и СОБЫТИЙ в вашей модели приносят пользу проекту МОДУЛЯ. Рассмотрим примеры продуманного проектирования МОДУЛЯ.

Основные правила именования модулей

В языках Java и C# имена МОДУЛЕЙ отражают иерархическую форму их организации.[1] Каждый уровень иерархии отделяется от других точкой. Иерархия имен обычно начинается с доменного имени организации, в которой разрабатывается проект. Доменное имя обычно начинается с верхнего уровня.

```
com.saasovation   // Java
SaaSOvation       // C#
```

Использование уникальных имен верхнего уровня предотвращает конфликт имен при использовании в вашем проекте МОДУЛЕЙ, разработанных сторонними организациями, а также при использовании сторонними организациями МОДУЛЕЙ, созданных вами. Если у вас есть вопросы, связанные с правилами выбора имен, обратитесь к стандарту.[2]

Вполне вероятно, что ваша организация уже приняла определенные правила выбора имен верхнего уровня для МОДУЛЕЙ. Не нарушайте их.

[1] Между пакетами языка Java и пространствами имен языка C# есть небольшие различия. Например, если вы программируете на языке C#, то можете использовать нашу книгу как справочник, а если вы программируете на других языках и платформах, то примеры придется немного адаптировать.

[2] http://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html. Существует перевод на русский язык: Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли. *Язык программирования Java SE 8. Подробное описание, 5-е издание* (ИД "Вильямс", 2015).

Соглашения о выборе имен для МОДУЛЕЙ

Следующий сегмент имени МОДУЛЯ идентифицирует ОГРАНИЧЕННЫЙ КОНТЕКСТ. Желательно, чтобы имя этого сегмента основывалось на имени ОГРАНИЧЕННОГО КОНТЕКСТА.

Команды компании SaaSOvation назвали свои МОДУЛИ так.

```
com.saasovation.identityaccess
com.saasovation.collaboration
com.saasovation.agilepm
```

Они подумывали назвать свои МОДУЛИ так, как показано ниже, но решили, что эти имена хуже предыдущих. Несмотря на то что в них используется точное имя КОНТЕКСТА, их наглядность снижается.

```
com.saasovation.identityandaccess
com.saasovation.agileprojectmanagement
```

Интересно также то, что они не использовали в именах МОДУЛЕЙ названия своих торговых марок. Эти названия могут изменяться, причем некоторые названия торговых марок могут вообще не иметь практически никакого отношения к базовым ОГРАНИЧЕННЫМ КОНТЕКСТАМ. Важность идентификации КОНТЕКСТА подчеркивается еще и тем, что именно это имя команда будет использовать в своих дискуссиях. Основная цель этих имен — соответствие ЕДИНОМУ ЯЗЫКУ. Если бы команда использовала следующие имена, то вряд ли кто-нибудь понял предназначение МОДУЛЕЙ.

```
com.saasovation.idovation
com.saasovation.collabovation
com.saasovation.projectovation
```

Имя первого МОДУЛЯ, `com.saasovation.idovation`, практически не имеет ничего общего с соответствующим ОГРАНИЧЕННЫМ КОНТЕКСТОМ. Второе имя намного точнее. Третье имя чуть-чуть лучше первого. По крайней мере оно содержит слово *project*. Тем не менее команда решила, что эти имена не обязаны буквально воспроизводить имена ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Если бы отдел маркетинга решил, что названия продуктов следует изменить — например, из-за нарушения авторского права или культурных особенностей, — эти имена МОДУЛЕЙ оказались бы совершенно устаревшими. По этой причине команда остановилась на первом решении.

На следующем шаге к имени присоединяется важный квалификатор. Он идентифицирует конкретный модуль в предметной области.

```
com.saasovation.identityaccess.domain
com.saasovation.collaboration.domain
com.saasovation.agilepm.domain
```

Правило образования этих имен напоминает правило выбора имен в традиционной **МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ (4)** и **ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ (4)**. В настоящее время системы, использующие МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ, обычно управляют уровнями с помощью ГЕКСАГОНАЛЬНОГО СТИЛЯ. Этот механизм образует внутреннюю часть приложения и включает в себя часть предметной области. Этим он похож на другие архитектурные стили.

Раздел `domain` в интерфейсах и классах может отсутствовать, служа лишь контейнером для МОДУЛЕЙ нижнего уровня. Рассмотрим МОДУЛИ нижнего уровня.

```
com.saasovation.identityaccess.domain.model
com.saasovation.collaboration.domain.model
com.saasovation.agilepm.domain.model
```

Здесь начинается определение модульных классов. Этот уровень пакета может содержать повторно используемые интерфейсы и абстрактные классы.

Проектировщики компании SaaSOvation решили включить в этот МОДУЛЬ общие интерфейсы, например интерфейсы, которые используются при публикации СОБЫТИЙ, а также абстрактные базовые классы для СУЩНОСТЕЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ.

```
ConcurrencySafeEntity
DomainEvent
DomainEventPublisher
DomainEventSubscriber
DomainRegistry
Entity
IdentifiedDomainObject
IdentifiedValueObject
```

Если вам нравится стиль размещения СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ за пределами МОДУЛЯ `domain.model`, вы можете выбрать следующие имена.

```
com.saasovation.identityaccess.domain.service
com.saasovation.collaboration.domain.service
com.saasovation.agilepm.domain.service
```

Совершенно не обязательно размещать СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ именно здесь. Это имеет смысл, если вы рассматриваете такие службы как среднемасштабный вспомогательный мини-уровень, надстроенный над моделью или окружающий ее [Эванс, с. 110, "Степень модульности"]. Однако следует иметь в виду, что такой подход быстро приводит к **АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ**, которая обсуждалась в главе, посвященной **СЛУЖБАМ (7)**.

Если вы не разделяете модель и службы на два разных пакета, можно просто отбросить МОДУЛЬ `model` и просто поместить все МОДУЛИ модели непосредственно в разделе `domain`.

```
com.saasovation.identityaccess.domain.conceptname
```

Это позволит исключить один уровень, который кажется излишним. А что же произойдет, если впоследствии вы решите разместить несколько СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ в ПОДМОДУЛЕ `domain.service`? Вы, наверное, будете разочарованы тем, что не сможете создать подмодуль `domain.model`.

Существует еще более важное обстоятельство, которое следует учесть. Напомним, что мы не разрабатываем предметную область. **ПРЕДМЕТНАЯ ОБЛАСТЬ (2)** — это бизнес-среда, в которой мы работаем. То, что мы проектируем и реализуем, представляет собой *модель предметной области*. Таким образом, лучше всего назвать МОДУЛЬ модели именем `domain.model`. Впрочем, это дело вкуса.

Модули контекста управления гибким проектированием

Текущим **СМЫСЛОВЫМ ЯДРОМ (2)** компании SaaSOvation является *Контекст управления гибким проектированием*, поэтому целесообразно рассмотреть, как спроектированы его модули.

Команда ProjectOvation выбрала три МОДУЛЯ верхнего уровня: `tenant`, `team` и `product`. Вот как выглядит первый МОДУЛЬ.

```
com.saasovation.agilepm.domain.model.tenant
    <<value object>> TenantId
```

Он содержит простой ОБЪЕКТ-ЗНАЧЕНИЕ `TenantId`, который содержит уникальный идентификатор конкретного арендатора, существующего в *Контексте идентификации и доступа*. Это МОДУЛЬ, от которого зависят все остальные МОДУЛИ в модели. Он важен для отделения одного объекта арендатора от других. Впрочем, это ациклическая зависимость. Модуль `tenant` не зависит от остальных.

Модуль team содержит АГРЕГАТЫ и СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ, которая используется для управления продукцией команд.

```
com.saasovation.agilepm.domain.model.team
<<service>> MemberService
<<aggregate root>> ProductOwner
<<aggregate root>> Team
<<aggregate root>> TeamMember
```

В модели существуют три АГРЕГАТА и интерфейс СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Класс Team содержит один экземпляр класса ProductOwner и любое количество экземпляров класса TeamMember в виде коллекции. Экземпляры классов ProductOwner и TeamMember созданы классом MemberService. Все три СУЩНОСТИ КОРНЯ АГРЕГАТА ссылаются на класс TenantId в МОДУЛЕ tenant.

```
package com.saasovation.agilepm.domain.model.team;
import com.saasovation.agilepm.domain.model.tenant.TenantId;
public class Team extends ConcurrencySafeEntity {
    private TenantId tenantId;
    ...
}
```

Класс MemberService является фасадом **ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (3)**, который согласовывает работу членов команды с точки зрения идентификаторов и ролей в *Контексте идентификации и доступа*. Синхронизация происходит в фоновом режиме, за рамками запросов обычных пользователей. Эта служба является профилактической. Она создает членов, зарегистрировавшихся в удаленном КОНТЕКСТЕ, обеспечивая синхронизацию с удаленной системой за короткое время, которое проходит от момента реальных изменений до отражения в удаленном КОНТЕСТЕ. Кроме того, при необходимости эта служба обновляет информацию о членах, например их имена и адреса электронной почты

Контекст управления гибким проектированием имеет родительский МОДУЛЬ с именем product и три дочерних МОДУЛЯ.

```
com.saasovation.agilepm.domain.model.product
   <<aggregate root>> Product
   ...
   com.saasovation.agilepm.domain.model.product.backlogitem
       <<aggregate root>> BacklogItem
       ...
   com.saasovation.agilepm.domain.model.product.release
       <<aggregate root>> Release
       ...
   com.saasovation.agilepm.domain.model.product.sprint
       <<aggregate root>> Sprint
       ...
```

Именно здесь происходит моделирование ядра системы Scrum. Здесь вы видите АГРЕГАТЫ Product, BacklogItem, Release и Sprint. Причины, по которым эти концепции моделируются разными АГРЕГАТАМИ, указаны в главе, посвященной **АГРЕГАТАМ (10)**.

Команде нравится, что имена МОДУЛЕЙ естественно согласованы с ЕДИНЫМ ЯЗЫКОМ: "product", "product backlog item", "product release" и "product sprint".

Если эти четыре АГРЕГАТА так тесно связаны, то почему не включить их в МОДУЛЬ product? Здесь не показаны все остальные части АГРЕГАТОВ, например СУЩНОСТЬ ProductBacklogItem, содержащуюся в классе Product, СУЩНОСТЬ Task в классе BacklogItem, СУЩНОСТЬ ScheduledBacklogItem в классе Release и СУЩНОСТЬ CommittedBacklogItem в классе Sprint. Каждый тип АГРЕГАТА содержит и другие СУЩНОСТИ и ОБЪЕКТЫ-ЗНАЧЕНИЯ. Кроме того, существует множество СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ, публикуемых некоторыми АГРЕГАТАМИ. Итак, если поместить почти 60 классов и интерфейсов в один МОДУЛЬ, то он станет довольно громоздким и будет производить впечатление плохо организованного. По этим причинам команда распределила сущности по связанным между собою МОДУЛЯМ.

Как и классы ProductOwner, Team и TeamMember, все типы АГРЕГАТОВ Product, BacklogItem, Release и Sprint ссылаются на класс TenantId. Существуют и другие зависимости. Рассмотрим класс Product.

```
package com.saasovation.agilepm.domain.model.product;

import com.saasovation.agilepm.domain.model.tenant.TenantId;

public class Product extends ConcurrencySafeEntity {
    private ProductId productId;
    private TeamId teamId;
    private TenantId tenantId;
    ...
}
```

Посмотрим также на класс BacklogItem.

```
package com.saasovation.agilepm.domain.model.product.backlogitem;

import com.saasovation.agilepm.domain.model.tenant.TenantId;

public class BacklogItem extends ConcurrencySafeEntity {
    private BacklogItemId backlogItemId;
    private ProductId productId;
```

```
    private TeamId teamId;
    private TenantId tenantId;
    ...
}
```

Ссылки на классы `TenantId` и `TeamId` представляют собой ациклические зависимости; они направлены в одну сторону. Впрочем, поскольку ссылка класса `BacklogItem` на класс `ProductId` кажется ациклической ссылкой МОДУЛЯ `backlogItem` на МОДУЛЬ `product`, на самом деле она является двусторонней. Каждый объект класса `Product` служит ФАБРИКОЙ, создающей экземпляры класса `BacklogItem` (а также `Release` и `Sprint`). Таким образом, эти зависимости являются двусторонними. С другой стороны, три этих ПОДМОДУЛЯ являются дочерними по отношению к МОДУЛЮ `product`, и существует возможность ослабить эти зависимости. Здесь необходимо достичь компромисса между строгостью организации и связанностью. Классы `BacklogItem`, `Release` и `Sprint` являются вполне естественными дочерними концепциями класса `Product`, поэтому нецелесообразно пытаться разделить эти концепции границами АГРЕГАТОВ.

Не могла бы команда разработчиков обеспечить слабую связанность между этими элементами с помощью обобщенного типа идентификатора, в котором классы `BacklogItem`, `Release` и `Sprint` ссылались бы на свой дочерний класс `Product`, не прибегая к связыванию?

```
public class BacklogItem extends ConcurrencySafeEntity {
    private Identity backlogItemId;
    private Identity productId;
    private Identity teamId;
    private Identity tenantId;
    ...
}
```

Действительно, команда могла бы обеспечить слабую связанность. Однако это открыло бы возможность для ошибок в коде, в котором каждый тип `Identity` был бы неотличимым от других.

Контекст управления гибким проектированием будет развиваться. Компания SaaSOvation планирует поддержку других подходов к гибкому проектированию и других инструментов. В результате придется создавать новые МОДУЛИ и, вероятно, вносить изменения в существующие. Придерживаясь принципов гибкого проектирования, команда приняла решение выполнить рефакторинг МОДУЛЕЙ и их тщательную проверку.

Далее мы рассмотрим, как можно использовать МОДУЛИ в других компонентах системы с помощью ее исходного кода.

МОДУЛИ на других уровнях

Независимо от выбора **АРХИТЕКТУРЫ (4)** вы всегда обязаны создавать и называть МОДУЛИ в компонентах, не относящихся к вашей модели. Здесь мы рассмотрим некоторые возможности традиционной **МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ (4)**, которые, впрочем, можно использовать и в других архитектурных стилях.

В типичной МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ, используемой для приложения и реализующей модель предметной области, вы организовали бы уровни следующим образом: ПОЛЬЗОВАТЕЛЬСКИЙ ИНТЕРФЕЙС, ПРИЛОЖЕНИЕ, ПРЕДМЕТНАЯ ОБЛАСТЬ и ИНФРАСТРУКТУРА. В зависимости от видов компонентов, определяемых потребностями вашего приложения, на каждом уровне МОДУЛИ будут изменяться.

Для начала рассмотрим **УРОВЕНЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (14)** и влияние поддержки ресурсов RESTful. Возможно, ваши ресурсы будут использоваться для обслуживания графического пользовательского интерфейса клиентов системы, создавая представление состояний в форматах XML, JSON и HTML. Однако в случае поддержки графического пользовательского интерфейса ресурсы RESTful не обязаны создавать представления, которые включают в себя их схему. Вместо этого они создают только свободные представления в разнообразных форматах разметки (XML, HTML) и сериализации (XML, JSON, Protocol Buffers). Все графические разметки, влияющие на состояние представления клиента, могут поступать из разных каналов. Таким образом, на УРОВНЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА, поддерживающего стиль REST, вы можете принять решение иметь по крайней мере два МОДУЛЯ, которые можно было бы назвать следующим образом.

```
com.saasovation.agilepm.resources
com.saasovation.agilepm.resources.view
```

Поддержка ресурсов RESTful реализуется в пакете `resources`. Вопросы, связанные с представлениями, решают компоненты во вложенном пакете `view` (или `presentation`, если хотите). В зависимости от количества ресурсов REST, требуемых вашей системой, каждый базовый МОДУЛЬ может иметь много ПОДМОДУЛЕЙ. Учитывая, что один класс провайдера ресурсов может поддерживать несколько идентификаторов URI, для поддержки всех базовых МОДУЛЕЙ можно ограничиться лишь несколькими классами провайдеров ресурсов. Определив реальные требования к ресурсам, легко решить, стоит ли продолжать дробить МОДУЛИ.

На ПРИКЛАДНОМ УРОВНЕ могут существовать другие МОДУЛИ, каждый из которых может содержать типы служб.

```
com.saasovation.agilepm.application.team
com.saasovation.agilepm.application.product
...
com.saasovation.agilepm.application.tenant
```

Согласно принципам проектирования с помощью ресурсов RESTful службы на ПРИКЛАДНОМ УРОВНЕ можно разделять на ПОДМОДУЛИ, только если это полезно. Например, в *Контексте идентификации и доступа* есть всего несколько ПРИКЛАДНЫХ СЛУЖБ, поэтому команда решила оставить их в главном МОДУЛЕ.

```
com.saasovation.identityaccess.application
```

Степень модульности проекта можно повысить еще больше. Например, если у вас есть несколько служб, скажем, двенадцать или больше, то было бы полезно осуществить еще более подробное разделение на МОДУЛИ.

МОДУЛИ перед ОГРАНИЧЕННЫМ КОНТЕКСТОМ

Принимая решение о разделении связных объектов модели предметной области на разные модели или объединении их в одной модели, следует проявлять осторожность. Иногда терминология предметной области сама подсказывает ответ, а иногда лишь запутывает процесс решения. Если терминология запутана и непонятно, следует ли проводить четкие границы контекстов, сначала следует обдумать возможность объединить все концепции в одной модели. Этот подход приводит к необходимости проводить более точные границы между МОДУЛЯМИ, а не грубые контуры ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Это не значит, что многочисленные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ практически не используются. Границы между моделями легко обосновать. Следует иметь в виду, что ОГРАНИЧЕННЫЕ КОНТЕКСТЫ нельзя считать заменой МОДУЛЕЙ. МОДУЛИ используются для агрегации связанных объектов предметной области и отделения от объектов, которые не являются связанными или являются слабо связанными.

Резюме

Мы рассмотрели вопросы объединения объектов предметной области в МОДУЛИ, объяснили, почему это важно и как это можно сделать.

- Вы поняли разницу между традиционными МОДУЛЯМИ и более современным подходом к обеспечению модульности.
- Вы осознали важность выбора имен для МОДУЛЕЙ в соответствии с правилами ЕДИНОГО ЯЗЫКА.
- Вы увидели, что неправильное проектирование МОДУЛЕЙ, даже автоматическое, ограничивает возможности моделирования.
- Вы узнали, как были спроектированы МОДУЛИ в *Контексте управления гибким проектированием* и почему были приняты те или иные решения.
- Вы получили представление о работе с МОДУЛЯМИ, находящимися за пределами модели.
- В заключение вы узнали, что, в первую очередь, следует думать об использовании МОДУЛЕЙ, а не о создании новых ОГРАНИЧЕННЫХ КОНТЕКСТОВ, если терминология предметной области не требует более грубого подразделения.

Далее мы приступим к освоению наиболее сложных инструментов предметно-ориентированного моделирования — АГРЕГАТОВ.

Глава 10

Агрегаты

Вселенная создана как совокупность постоянных объектов, связанных причинными отношениями, которые независимы от предмета и помещены в объективное пространство и время.

Жан Пиаже

Кластеризация **СУЩНОСТЕЙ (5)** и **ОБЪЕКТОВ-ЗНАЧЕНИЙ (6)** в рамках **АГРЕГАТА** с четко определенными границами согласованности, на первый взгляд, кажется довольно простой задачей, но среди всех тактических DDD-инструментов этот шаблон является одним из наименее понятных.

Назначение главы

- На примере проекта компании SaaSOvation выявить негативные последствия неправильного моделирования агрегатов.
- Освоить главные правила проектирования агрегатов, представляющие собой набор лучших практических советов.
- Научиться моделировать истинные инварианты внутри границ согласованности в соответствии с реальными бизнес-правилами.
- Рассмотреть преимущества проектирования небольших агрегатов.
- Выяснить, почему следует проектировать агрегаты так, чтобы ссылаться на другие агрегаты по идентификатору.
- Выявить важность итоговой согласованности за пределами границы агрегата.
- Освоить методы реализации агрегатов с помощью принципов совмещения данных и *поведения* (приказывай, а не спрашивай) и закона Деметры.

Для начала целесообразно рассмотреть несколько общих вопросов. Является ли АГРЕГАТ лишь способом создания *кластеров* на диаграмме тесно связанных объектов, объединенных общим предком? Если так, то существует ли некий практический предел количества объектов, которые разрешено размещать на диаграмме? Поскольку один экземпляр АГРЕГАТА может ссылаться на другие экземпляры АГРЕГАТА, можно ли перемещаться по связям достаточно далеко, изменяя различные объекты на пути следования? Что означают понятия *инвариантов* и

границы согласованности? Ответ на последний вопрос в значительной степени определяет ответы на остальные вопросы.

Существует множество способов неправильно моделировать АГРЕГАТЫ. С одной стороны, можно соблазниться удобством проектирования слишком больших АГРЕГАТОВ. С другой стороны, существует опасность создания "пустых" АГРЕГАТОВ, неспособных защитить истинные инварианты. Как мы увидим, необходимо избежать обеих этих крайностей и уделить внимание бизнес-правилам.

Использование АГРЕГАТОВ в СМЫСЛОВОМ ЯДРЕ системы Scrum

Мы внимательно изучим то, как АГРЕГАТЫ используются компанией SaaSOvation, в частности в приложении под названием ProjectOvation внутри *Контекста управления гибким проектирования*. Компания использует традиционную модель управления проектами Scrum, в которую входят продукт, владельцы продукта, команда, задачи, списка заданий, запланированные выпуски и спринты. Если взглянуть на модель Scrum в целом, то легко понять, что именно она является целью проекта ProjectOvation; эта ситуация знакома большинству из нас. Терминология модели Scrum образует первое приближение **ЕДИНОГО ЯЗЫКА (1)**. Так как эта модель представляет собой программное приложение, предоставляемое по подписке как услуга (software as a service — SaaS), каждая организация-подписчик регистрируется как *арендатор* (tenant). Это еще один термин нашего ЕДИНОГО ЯЗЫКА.

Представьте себе, что компания собрала группу разработчиков и талантливых экспертов по модели Scrum. Поскольку их опыт работы с инструментами DDD довольно ограничен, команда сделает несколько ошибок, продвигаясь по трудному пути обучения. Они растут как специалисты, учась на своем опыте работы с АГРЕГАТАМИ, и мы вместе с ними. Их усилия помогут нам распознавать и изменять аналогичные неблагоприятные ситуации, которые могут возникнуть в нашем собственном программном обеспечении.

Концепции предметной области Scrum, а также требования, предъявляемые к производительности и масштабируемости, сложнее, чем те, с которыми команда столкнулась в первоначальном **СМЫСЛОВОМ ЯДРЕ (2)** — *Контексте сотрудничества*. Для того чтобы решить эти проблемы, команда применит АГРЕГАТЫ, один из тактических инструментов DDD.

Как лучше выбрать кластеры объектов? Шаблон АГРЕГАТ решает проблемы композиции и сокрытия информации. Команда знает, как решить эти проблемы. Шаблон АГРЕГАТ также очерчивает границы согласованности и определяет транзакции. Эти вопросы до сих пор не слишком беспокоили членов команды. Они выбрали механизм постоянного хранения, который помогает управлять атомарными фиксациями их данных. Однако это оказалось главной ошибкой команды, вызванной непониманием принципов работы шаблона АГРЕГАТ. Вот что произошло. Команда рассматривала следующие предложения на ЕДИНОМ ЯЗЫКЕ.

- Продукты имеют списки заданий, выпуски и спринты.
- Списки заданий для нового продукта планируются.
- Выпуски нового продукта назначаются.
- Спринты нового продукта назначаются.
- Запланированный список заданий может быть назначен для выпуска.
- Назначенный список заданий может быть зарегистрирован для спринта.

По этим утверждениям они обрисовали модель и сделали первое проектное приближение. Посмотрим, что у них получилось.

Первая попытка: крупнокластерный агрегат

Команда придала слишком большое значение слову *продукты* в первом предложении, что сильно повлияло на первый проект АГРЕГАТА для данной предметной области.

Некоторым проектировщикам показалось, что речь идет о композиции, а объекты следует связать с помощью графа. В таком случае очень большое значение приобретает поддержка жизненных циклов этих объектов. В результате разработчики добавили в спецификацию следующие правила согласованности.

- Если список заданий зарегистрирован для спринта, то нельзя допускать его удаление из системы.
- Если спринт имеет зарегистрированные списки заданий, то нельзя допускать его удаление из системы.
- Если выпуск имеет назначенные списки заданий для спринта, то нельзя допускать его удаление из системы.
- Если список заданий для спринта назначен для выпуска, то нельзя допускать его удаление из системы.

В результате класс `Product` сначала был смоделирован как очень большой АГРЕГАТ. Объект КОРНЯ `Product` содержит все объекты классов `BacklogItem`,

Release и Sprint, связанные с ними. Проект интерфейса защищал все эти части от несанкционированного удаления клиентами.

Этот проект представлен в следующем коде, а также в виде диаграммы UML на рис. 10.1.

```
public class Product extends ConcurrencySafeEntity {
    private Set<BacklogItem> backlogItems;
    private String description;
    private String name;
    private ProductId productId;
    private Set<Release> releases;

    private Set<Sprint> sprints;
    private TenantId tenantId;
    ...
}
```

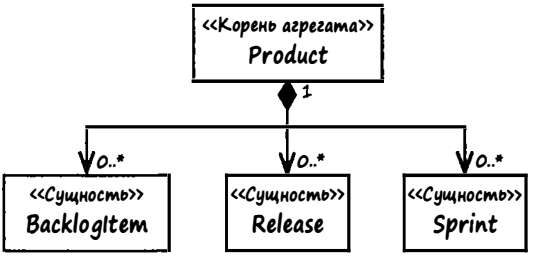


Рис. 10.1. Класс Product моделируется как очень большой АГРЕГАТ

Большой АГРЕГАТ выглядел привлекательным, но не был действительно практичным. Как только приложение начинало работать в многопользовательской среде, регулярно возникали сбои транзакций. Рассмотрим поближе несколько клиентских шаблонов использования и посмотрим, как они взаимодействуют с нашей технической моделью. Наши экземпляры АГРЕГАТА используют оптимистичный параллелизм для защиты объектов постоянного хранения от одновременных перекрывающихся модификаций различными клиентами, не прибегая к блокировкам базы данных. Как указано в главе, посвященной **СУЩНОСТЯМ (5)**, объект содержит номер версии, который постепенно увеличивается по мере внесения и проверки изменений, прежде чем он будет сохранен в базе данных. Если версия на сохраненном объекте больше, чем версия на копии клиента, то клиентская версия считается устаревшей и изменения отвергаются.

Рассмотрим распространенный сценарий, в котором одновременно работают много клиентов.

- Два пользователя, Билл и Джо, видят один и тот же экземпляр класса `Product`, помеченный как "версия 1", и начинают с ним работать.
- Билл планирует новый экземпляр класса `BacklogItem` и регистрирует его. Номер версии экземпляра класса `Product` увеличивается до 2.
- Джо назначает новый экземпляр класса `Release` и пытается его сохранить, но его регистрация заканчивается неудачей, потому что он использовал первую версию экземпляра класса `Product`.

Для параллельной работы в таких ситуациях используются механизмы постоянного хранения.[1] Если вы считаете, что стандартные конфигурации параллелизма могут изменяться, не торопитесь с выводами. Этот подход действительно важен для защиты инвариантов АГРЕГАТОВ от параллельных изменений.

Проблемы с параллельной работой возникли уже при работе всего двух пользователей. Чем больше пользователей, тем сложнее становится проблема. В модели Scrum многие пользователи часто выполняют перекрывающиеся изменения в ходе планирования спринта и его выполнения. Сбой хотя бы одного из параллельных запросов совершенно недопустим.

Планирование нового списка заданий для спринта никак не должно влиять на планирование нового выпуска! Почему регистрация Джо завершилась сбоем? Главная причина заключается в том, что крупный АГРЕГАТ был разработан на основе ложных инвариантов, а не реальных бизнес-правил. Эти ложные инварианты — искусственные ограничения, наложенные разработчиками. Есть и другие способы, с помощью которых команда может предотвратить недопустимое удаление, не прибегая к слишком строгим ограничениям. Помимо транзакционных проблем, у проекта также есть недостатки, связанные с недостаточной масштабируемостью и производительностью.

Вторая попытка: множество агрегатов

Теперь рассмотрим альтернативную модель, показанную на рис. 10.2, в которой фигурируют четыре разных АГРЕГАТА. Каждая из зависимостей косвенно связана с общим экземпляром класса `ProductId`, представляющим собой идентификатор класса `Product`, который считается родительским классом для трех остальных классов.

---

[1] Например, библиотека Hibernate в этом случае обеспечивает оптимистический параллелизм. Это относится и к хранилищам "ключ–значение", потому что весь АГРЕГАТ часто сериализуется как одно значение, если он не был спроектирован так, чтобы его составные части сохранялись по отдельности.

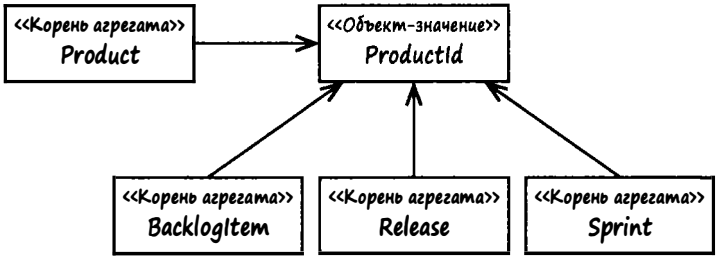


Рис. 10.2. Класс `Product` и связанные с ним концепции моделируются отдельными типами АГРЕГАТОВ

Разделение одного большого АГРЕГАТА на четыре отдельных агрегата приводит к изменению контрактов методов класса `Product`. В крупнокластерном АГРЕГАТЕ сигнатуры методов выглядят следующим образом.

```
public class Product ... {
    ...
    public void planBacklogItem(
        String aSummary, String aCategory,
        BacklogItemType aType, StoryPoints aStoryPoints) {
            ...
    }
    ...

    public void scheduleRelease(
        String aName, String aDescription,
        Date aBegins, Date anEnds) {
        ...
    }

    public void scheduleSprint(
        String aName, String aGoals,
        Date aBegins, Date anEnds) {
        ...
    }
    ...
}
```

Все эти методы представляют собой команды **CQS** [Fowler, CQS]; иначе говоря, они модифицируют состояние экземпляра класса `Product`, добавляя в коллекцию новый элемент, поэтому в качестве типа возвращаемого значения указано ключевое слово `void`. В схеме с несколькими АГРЕГАТАМИ мы получаем следующий код.

```
public class Product ... {
    ...
    public BacklogItem planBacklogItem(
```

```
        String aSummary, String aCategory,
        BacklogItemType aType, StoryPoints aStoryPoints) {
        ...
    }

    public Release scheduleRelease(
        String aName, String aDescription,
        Date aBegins, Date anEnds) {
        ...
    }

    public Sprint scheduleSprint(
        String aName, String aGoals,
        Date aBegins, Date anEnds) {
        ...
    }
    ...
}
```

Эти переделанные методы имеют контракт запроса CQS и действуют как **ФАБРИКИ (11)**; иначе говоря, они создают новый экземпляр АГРЕГАТА и возвращают ссылку на него. Теперь, когда клиент захочет запланировать список заданий для спринта, транзакционная **ПРИКЛАДНАЯ СЛУЖБА (14)** должна выполнить следующие действия.

```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void planProductBacklogItem(
           String aTenantId, String aProductId,
           String aSummary, String aCategory,
           String aBacklogItemType, String aStoryPoints) {

           Product product =
               productRepository.productOfId(
                      new TenantId(aTenantId),
                      new ProductId(aProductId));

           BacklogItem plannedBacklogItem =
               product.planBacklogItem(
                      aSummary,
                      aCategory,
                      BacklogItemType.valueOf(aBacklogItemType),
                      StoryPoints.valueOf(aStoryPoints));

          backlogItemRepository.add(plannedBacklogItem);
    }
 ...
}
```

Таким образом, мы решили проблему сбоя транзакции *с помощью моделирования*. Теперь одновременные пользовательские запросы могут безопасно создать любое количество экземпляров классов `BacklogItem`, `Release` и `Sprint`. Это довольно просто.

Однако, несмотря на очевидные транзакционные преимущества, четыре АГРЕГАТА меньшего размера менее удобны с точки зрения клиентов. Возможно, вместо этого мы могли настроить большой АГРЕГАТ, чтобы устранить проблемы параллелизма. Устанавливая флаг `optimistic-lock` библиотеки Hibernate равным `false`, мы предотвращаем цепную реакцию сбоев транзакции. Поскольку все создаваемые экземпляры классов `BacklogItem`, `Release` или `Sprint` не имеют ни одного инварианта, почему бы просто не допустить неограниченный рост коллекций и проигнорировать эти конкретные модификации класса `Product`? С какими накладными расходами связано использование крупнокластерного АГРЕГАТА? Проблема заключается в том, что рост коллекции может стать неконтролируемым. Прежде чем всесторонне исследовать эту проблему, рассмотрим самую важную подсказку, необходимую команде SaaSOvation для моделирования.

Правило: моделируйте истинные инварианты в границах согласованности

Пытаясь выявить АГРЕГАТЫ в **ОГРАНИЧЕННОМ КОНТЕКСТЕ (2)**, мы должны понять, что собой представляют истинные инварианты модели. Только в этом случае мы сможем определить, какие объекты следует объединить в конкретный АГРЕГАТ.

Инвариант — это бизнес-правило, которое всегда сохраняет свою непротиворечивость. Это явление называется *транзакционной согласованностью* (transactional consistency), которая считается мгновенной и атомарной. Кроме того, существует *итоговая согласованность* (eventual consistency). Обсуждая инварианты, мы говорим о транзакционной согласованности. Мы можем иметь инвариант

```
c = a + b
```

Следовательно, если `a` равно 2 и `b` равно 3, то `c` должно быть равным 5. В соответствии с этим правилом и условиями, если `c` не равно 5, то системный инвариант нарушается. Для того чтобы сохранить согласованность значения `c`, мы очерчиваем границу вокруг этих конкретных атрибутов модели.

```
AggregateType1 {

    int a;

    int b;

    int c;

    operations ...

}
```

С логической точки зрения граница согласованности означает, что все находящееся внутри нее подчиняется определенному набору деловых инвариантных правил независимо от того, какие операции при этом выполняются. Согласованность всего, что находится вне этой границы, для АГРЕГАТА не важна. Таким образом, *АГРЕГАТ* можно считать синонимом *границы транзакционной согласованности.* (В нашем ограниченном примере тип `AggregateType1` имеет три атрибута типа `int`, но любой такой АГРЕГАТ мог бы содержать атрибуты различных типов.)

При использовании типичного механизма постоянного хранения для управления согласованностью мы используем отдельную транзакцию[2]. После завершения транзакции все, что находится внутри границы, должно быть согласованным. *Правильно разработанный АГРЕГАТ — это такой агрегат, который может как угодно изменяться в соответствии с инвариантами, абсолютно непротиворечивыми в рамках отдельной транзакции.* В то же время правильно разработанный ОГРАНИЧЕННЫЙ КОНТЕКСТ всегда изменяет только один экземпляр АГРЕГАТА за транзакцию. Более того, *мы не можем правильно рассуждать о проектировании агрегата, не применяя транзакционный анализ.*

Ограничение модификации одним экземпляром АГРЕГАТА за транзакцию может показаться чрезмерно строгим. Однако это цель, к которой следует стремиться в большинстве случаев. Это правило отображает саму суть использования АГРЕГАТОВ.

Заметки на доске

- Выпишите на доске все крупные АГРЕГАТЫ, существующие в вашей системе.
- Напишите примечания к каждому из АГРЕГАТОВ, объясняющие, почему он представляет собой большой кластер, и описывающие все потенциальные проблемы, вызванные его размером.

[2] Эту транзакцию можно обработать с помощью шаблона `UNIT OF WORK` [Fowler, P of EAA].

- Затем перечислите названия всех АГРЕГАТОВ, которые одновременно изменяются в рамках одной и той же транзакции.
- Напишите примечания к каждому из этих АГРЕГАТОВ, в котором укажите, нарушаются ли инварианты внутри границ согласованности из-за неправильно спроектированных АГРЕГАТОВ.

Тот факт, что разрабатывать АГРЕГАТЫ необходимо с упором на согласованность, подразумевает, что пользовательский интерфейс должен концентрировать каждый запрос на выполнении единственной команды, которая применяется только к одному экземпляру АГРЕГАТА. Если запросы пользователя попытаются достигнуть слишком многого, то приложение будет вынуждено одновременно модифицировать многочисленные экземпляры.

Следовательно, АГРЕГАТЫ в основном ориентируются на границы согласованности и не сводятся к желанию проектировать объектные графы. В реальной жизни некоторые инварианты будут более сложными, чем показанный выше. Несмотря на это, как правило, инварианты будут менее требовательны к нашим усилиям по моделированию, позволяя *проектировать небольшие АГРЕГАТЫ*.

Правило: проектируйте небольшие агрегаты

Мы можем теперь тщательно исследовать следующий вопрос: "Какие накладные расходы связаны с хранением крупнокластерного АГРЕГАТА?" Даже если мы гарантируем, что каждая транзакция будет успешно выполнена, большой кластер все еще ограничивает производительность и масштабируемость системы. Поскольку компания SaaSOvation разрабатывает свой рынок, он собирается привлечь много арендаторов. Поскольку все арендаторы проявляют твердую приверженность проекту ProjectOvation, компания SaaS будет размещать на своем веб-сайте все больше проектов и элементов управления, чтобы удовлетворить потребности клиентов. Это приведет к огромному количеству продуктов, списков заданий, выпусков, спринтов и других объектов. Производительность и масштабируемость — это нефункциональные требования, которые не могут быть проигнорированы.

Имея в виду производительность и масштабируемость, попробуем представить себе, что произойдет, если один пользователь одного арендатора захочет добавить единственный список заданий к продукту, который существует многие годы и уже имеет тысячи списков заданий для спринта. Предположим, что механизм постоянного хранения допускает отложенную загрузку (например,

библиотека Hibernate). Мы почти никогда не загружаем все списки заданий, выпуски и спринты одновременно. Однако тысячи неудовлетворенных элементов были бы загружены в память только для того, чтобы добавить один новый элемент к уже и без того большой коллекции. Еще хуже, если механизм постоянного хранения не поддерживает отложенную загрузку. Несмотря на стремление правильно использовать память, иногда мы должны были бы загружать многочисленные коллекции, например, планируя список заданий к выпуску или регистрируя его для спринта; в этом случае были бы загружены все списки или все выпуски или все спринты.

Для того чтобы яснее представить себе эту ситуацию, взгляните на диаграмму, изображенную на рис. 10.3. Эта диаграмма представляет композицию в увеличенном масштабе. Не обманывайтесь *отношением 0..*; количество связей практически никогда не бывает равным 0 и со временем будет расти.* Похоже, что даже при выполнении относительно простых операций нам придется одновременно загружать в память многие тысячи объектов. И все это относится лишь к одному члену команды одного арендатора одного продукта. Следует иметь в виду, что таких арендаторов может быть сотни и тысячи, причем каждый из них может иметь много команд и много продуктов. Со временем ситуация будет только ухудшаться.

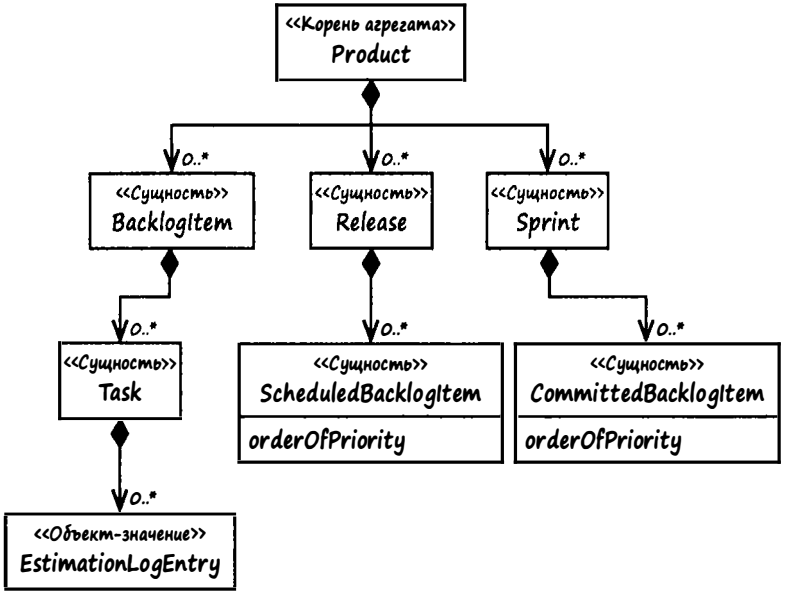


Рис. 10.3. В этой модели класса `Product` многочисленные крупные коллекции загружаются в память при выполнении многих базовых операций

Этот крупнокластерный АГРЕГАТ никогда не будет хорошо работать или масштабироваться. Скорее всего, он станет причиной крупных неприятностей которые приведут к отказу всей системы. Он был неправильным с самого начала, потому что основывался на ложных инвариантах и ориентировался на удобство проектирования, а не на успех выполнения транзакций, производительность и масштабируемость.

Если мы собираемся разрабатывать небольшие АГРЕГАТЫ, то что значит слово "небольшие"? В экстремальном варианте АГРЕГАТ может содержать только глобально уникальный идентификатор и один дополнительный атрибут. Этот вариант является *нежелательным* (конечно, если АГРЕГАТ действительно не должен содержать только один атрибут). Достаточно просто ограничить АГРЕГАТ КОРНЕВОЙ СУЩНОСТЬЮ и минимальным количеством атрибутов и/или свойств, имеющих тип ЗНАЧЕНИЯ.[3] Правильный минимум — это значит столько, сколько необходимо, и не больше.

А сколько же атрибутов необходимо? Ответ простой: эти атрибуты должны быть согласованы с другими, даже если эксперты предметной области не указали это в виде правил. Например, класс `Product` имеет атрибуты `name` и `description`. Невозможно представить себе ситуацию, в которой атрибуты `name` и `description` были бы несогласованными и моделировались в отдельных АГРЕГАТАХ. Изменяя атрибут `name`, вы, вероятно, изменяете атрибут `description`. Если вы изменяете один атрибут и не изменяете другой, то, вероятно, исправляете опечатку или уточняете атрибут `description`, чтобы он лучше соответствовал атрибуту `name`. Даже если эксперты предметной области не считают это бизнес-правило явным, оно остается неявным.

Предположим, вы считаете, что должны моделировать внутреннюю часть как ОБЪЕКТ. Сначала выясните, должна ли эта часть изменяться со временем и можно ли ее полностью заменить при необходимости. Если экземпляры можно полностью заменить, следует использовать ОБЪЕКТ-ЗНАЧЕНИЕ, а не ОБЪЕКТ. Порой часть все же необходимо представлять в виде ОБЪЕКТА. Если же рассмотреть разные сценарии проекта, то многие концепции, моделируемые в виде ОБЪЕКТОВ, можно пересмотреть и представить в виде ОБЪЕКТОВ-ЗНАЧЕНИЙ. Выбор типов ЗНАЧЕНИЯ для частей АГРЕГАТА не означает, что АГРЕГАТ не изменяется, так как сама КОРНЕВАЯ СУЩНОСТЬ изменяется, когда заменяется одно из его свойств, имеющих тип ЗНАЧЕНИЯ.

Ограничение внутренних частей типами ЗНАЧЕНИЯ имеет важные преимущества. В зависимости от используемого механизма постоянного хранения

---

[3] Свойство, имеющее тип ЗНАЧЕНИЯ, — это атрибут, содержащий ссылку на ОБЪЕКТ-ЗНАЧЕНИЕ. Как указал Уорд Каннингем, описывая шаблон WHOLE VALUE [Cunningham, Whole Value], этим он отличается от простого атрибута, например строкового или числового типа.

ЗНАЧЕНИЯ могут быть сериализированы вместе с КОРНЕВОЙ СУЩНОСТЬЮ, тогда как для ОБЪЕКТОВ может потребоваться отдельное отслеживаемое хранение. В этом случае накладные расходы выше, чем расходы, связанные с моделированием частей в виде ОБЪЕКТА, например, потому, что для их чтения с помощью библиотеки Hibernate необходимы объединения SQL. Чтение отдельной строки таблицы базы данных выполняется намного быстрее. ОБЪЕКТЫ-ЗНАЧЕНИЯ меньше и безопаснее (содержат меньше ошибок). Благодаря их неизменности, проще доказать их правильность с помощью модульного тестирования. Эти преимущества обсуждаются в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**.

В отчете об одном из проектов для рынка производных финансовых инструментов, выполненном с помощью каркаса Qi4j [Öberg], Никлас Хедман (Niclas Hedhman)[4] сообщил, что его команда смогла спроектировать около 70% АГРЕГАТОВ в виде КОРНЕВОЙ СУЩНОСТИ, содержащей свойства с типами ЗНАЧЕНИЯ. Остальные 30% содержали всего две-три СУЩНОСТИ. Это не значит, что все модели предметных областей также разделяются в соотношении 70/30. Отсюда следует лишь, что большую часть АГРЕГАТОВ можно моделировать в виде простой СУЩНОСТИ, а не КОРНЯ.

В книге [Эванс] приведен пример АГРЕГАТА, в котором целесообразно хранить несколько СУЩНОСТЕЙ. Например, можно назначить максимально допустимую сумму заказа, которую не должна превышать сумма всех элементов в строке. Если несколько пользователей одновременно добавляют элементы в строку, то правило становится запутанным. Любое отдельное добавление не должно превышать заданного предела, но этому правилу должны подчиняться все пользователи. Я не буду приводить здесь решение этой проблемы, а хочу лишь подчеркнуть, что в большинстве случаев управлять инвариантами бизнес-моделей проще, чем в данном примере. С учетом сказанного, следует стремиться моделировать АГРЕГАТЫ так, чтобы они содержали как можно меньше атрибутов.

Более мелкие АГРЕГАТЫ не только быстрее работают и лучше масштабируются, но и способствуют успеху транзакций, потому что конфликты, препятствующие успешной фиксации транзакции, возникают редко. Это облегчает использование системы. В вашей предметной области не часто будут возникать истинные инвариантные ограничения, вынуждающие использовать крупные АГРЕГАТЫ. По этой причине просто было бы разумно ограничить размер АГРЕГАТОВ. Если вы вдруг столкнетесь с истинным правилом согласованности, добавьте немного других СУЩНОСТЕЙ или, при необходимости, их коллекцию, но продолжайте стремиться к как можно меньшему общему размеру.

---

[4] См. также www.jroller.com/niclas/.

Не доверяйте каждому сценарию использования

Бизнес-аналитики играют важную роль в выработке спецификаций сценариев использования. Большая и подробная спецификация является результатом большой работы. Она будет влиять на многие наши проектные решения. И все же мы не должны забывать, что сценарии использования, полученные таким образом, не отражают точку зрения экспертов в предметной области и разработчиков нашей сплоченной команды моделирования. Мы все еще должны согласовывать каждый сценарий использования с нашей текущей моделью и проектом, включая наши решения, касающиеся АГРЕГАТОВ. Иногда в некотором сценарии использования возникает необходимость создавать многочисленные экземпляры АГРЕГАТОВ. В таком случае мы должны определить, распространяется ли цель пользователя на многократные транзакции, связанные с использованием механизма постоянного хранения, или это происходит всего один раз. В последнем случае стоит занять скептическую позицию. Независимо от того, насколько хорошо написан такой сценарий, он может неточно отражать истинные АГРЕГАТЫ в нашей модели.

Если границы АГРЕГАТОВ соответствуют реальным деловым ограничениям, а бизнес-аналитики рисуют картину, изображенную на рис. 10.4, то у вас проблемы. Продумывая перестановку порядка фиксации, вы увидите, что есть ситуации, в которых два из трех запросов не будут выполнены.[5] Как это отражается на вашем проекте? Ответ на этот вопрос может привести к более глубокому пониманию предметной области. Попытка сохранить согласованность многочисленных экземпляров АГРЕГАТОВ может свидетельствовать о том, что ваша команда пропустила инвариант. В итоге вы можете свернуть многочисленные АГРЕГАТЫ в одно новое понятие с новым именем, чтобы реализовать новое бизнес-правило. (И конечно, они могут быть только частями старых АГРЕГАТОВ, которые свернуты в новый.)

Таким образом, новый сценарий использования может привести к осознанию необходимости реконструкции АГРЕГАТА, но здесь также следует быть скептичными. Хотя формирование одного АГРЕГАТА, состоящего из многочисленных АГРЕГАТОВ, может привести к абсолютно новому понятию с новым именем, моделирование этого нового понятия приведет вас к разработке крупнокластерного АГРЕГАТА, который может породить все проблемы, характерные для этого подхода. Нельзя ли применить другой подход?

---

[5] Это не отменяет того факта, что некоторые сценарии использования, описывающие модификации многочисленных АГРЕГАТОВ, которые охватывают транзакции, могут быть очень хорошими. Цель пользователя нельзя отождествлять с транзакцией. Нас интересуют только такие пользовательские сценарии, которые действительно свидетельствуют о необходимости модификации многочисленных АГРЕГАТОВ в рамках одной транзакции.

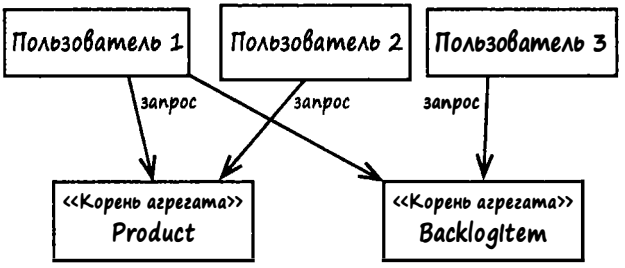


Рис. 10.4. Три пользователя параллельно работают с одним и тем же содержимым, пытаясь получить доступ к двум экземплярам АГРЕГАТОВ, что повышает количество сбоев транзакций

Если вам предъявляют сценарий использования, который требует поддержания согласованности в рамках единственной транзакции, это еще не означает, что вы обязаны его реализовывать. Часто в таких случаях бизнес-цели можно достичь, одновременно обеспечив итоговую согласованность между АГРЕГАТАМИ. Команда должна критически исследовать сценарии использования и проверить их предположения, особенно когда следование им привело бы к громоздким проектам. Команде, вероятно, придется переписать сценарий использования (или по крайней мере повторно рассмотреть его с точки зрения независимого бизнес-аналитика). Новый сценарий использования обеспечил бы *итоговую согласованность и приемлемую задержку обновления*. Это одна из проблем, которые мы рассмотрим далее в этой главе.

Правило: ссылайтесь на другие АГРЕГАТЫ по идентификаторам

При разработке АГРЕГАТОВ может возникнуть желание создать структуру, допускающую обход по глубоким графам объектов, но назначение этого шаблона другое. В книге [Эванс] утверждается, что один АГРЕГАТ может содержать ссылки на КОРНИ других АГРЕГАТОВ. Однако мы должны иметь в виду, что это не помещает АГРЕГАТ, на который ссылаются, в границы согласованности АГРЕГАТА, который на него ссылается. Ссылка не порождает один целостный АГРЕГАТ. Вместо него существуют два (или больше) АГРЕГАТА, как показано в рис. 10.5.

В языке Java эта связь моделируется следующим образом.

```
public class BacklogItem extends ConcurrencySafeEntity {
    ...
    private Product product;
    ...
}
```

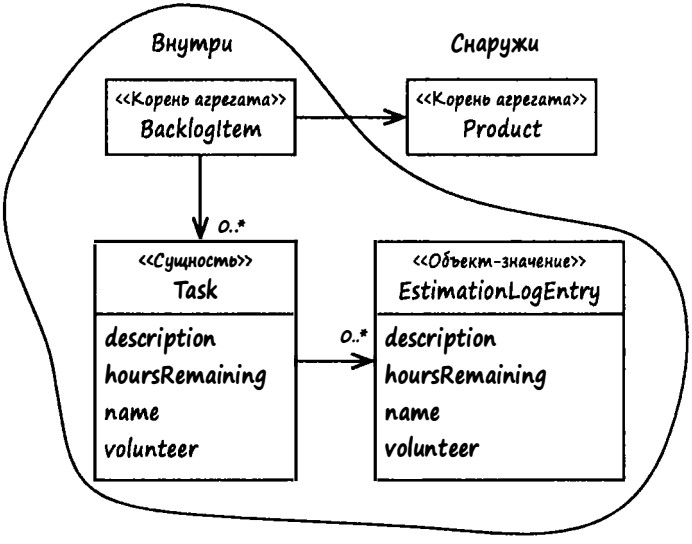


Рис. 10.5. Существуют два, а не один АГРЕГАТ

Иначе говоря, объект класса `BacklogItem` содержит прямую ссылку на объект класса `Product`.

Это и другие обстоятельства, которые мы еще обсудим, вызывают такие последствия.

1. И АГРЕГАТ (`BacklogItem`), который ссылается, и АГРЕГАТ (`Product`), на который ссылаются, *не должны* изменяться в рамках одной и той же транзакции. В отдельно взятой транзакции может изменяться только один из них.
2. Если вы изменяете несколько экземпляров в рамках одной транзакции, значит, ваши границы согласованности проведены неправильно. В этом случае, вероятно, в ходе моделирования вы пропустили какую-то концепцию; некая концепция ЕДИНОГО ЯЗЫКА еще не выявлена, хотя она бросается в глаза (такая ситуация была описана выше).
3. Если вы пытаетесь применить п. 2 и получаете крупнокластерный АГРЕГАТ со всеми его недостатками, указанными ранее, возможно, вам следует использовать концепцию итоговой, а не атомарной согласованности (см. ниже).

Если вы не будете хранить ссылки, то не сможете модифицировать другой АГРЕГАТ. Следовательно, желание модифицировать несколько АГРЕГАТОВ в рамках одной и той же транзакции можно подавить, вообще избежав

возникновения такой ситуации. Однако требование, чтобы модели предметной области всегда содержали несколько связей, является слишком обременительным. Как облегчить выполнение этого требования, защитить транзакцию от неправильного использования или сбоя, в то же время обеспечив высокую производительность и масштабируемость модели?

Ссылки на АГРЕГАТЫ по их идентификаторам

Старайтесь ссылаться на внешние АГРЕГАТЫ только по их глобальным уникальным идентификаторам, а не храните прямые ссылки на них (или указатели). Эта ситуация изображена на рис. 10.6.

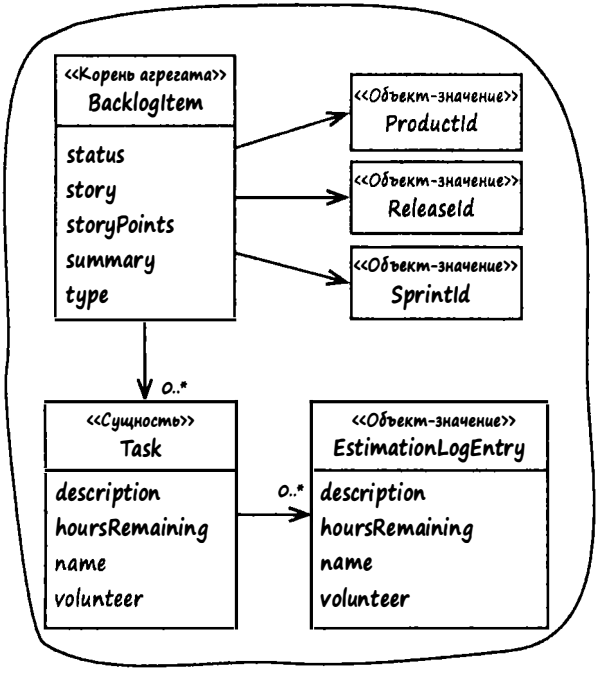


Рис. 10.6. АГРЕГАТ `BacklogItem`, ссылающийся за пределы своей границы с помощью идентификаторов

Исходный код можно переделать следующим образом.

```
public class BacklogItem extends ConcurrencySafeEntity {
    ...
    private ProductId productId;
    ...
}
```

Размеры АГРЕГАТОВ с косвенными ссылками на объекты, меньше, чем размеры АГРЕГАТОВ с прямыми ссылками, потому что первые не подразумевают загрузку ссылок. Поскольку экземпляры требуют меньше времени для загрузки и занимают меньше памяти, модель может работать быстрее. Использование меньшего размера памяти также положительно влияет на механизмы распределения памяти и сбора мусора.

Навигация по модели

Ссылки по идентификатору не могут полностью исключить навигацию по модели. Для выполнения поиска к **ХРАНИЛИЩУ (12)** иногда обращаются из АГРЕГАТА. Этот способ называется **ОТКЛЮЧЕННОЙ МОДЕЛЬЮ ПРЕДМЕТНОЙ ОБЛАСТИ (DISCONNECTED DOMAIN MODEL)**. По существу, он представляет собой одну из форм отложенной загрузки. Впрочем, существуют разные подходы к поиску зависимых объектов с помощью АГРЕГАТОВ: использовать ХРАНИЛИЩЕ или **СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ (7)**. Этой службой может управлять клиент ПРИКЛАДНОЙ СЛУЖБЫ, который потом передает управление АГРЕГАТУ.

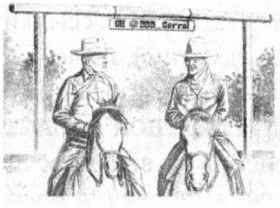
```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void assignTeamMemberToTask(
        String aTenantId,
        String aBacklogItemId,
        String aTaskId,
        String aTeamMemberId) {

    BacklogItem backlogItem =
           backlogItemRepository.backlogItemOfId(
                new TenantId(aTenantId),
                new BacklogItemId(aBacklogItemId));

    Team ofTeam =
        teamRepository.teamOfId(
                backlogItem.tenantId(),
                backlogItem.teamId());

    backlogItem.assignTeamMemberToTask(
                new TeamMemberId(aTeamMemberId),
                ofTeam,
                new TaskId(aTaskId));
    }
    ...
}
```

Наличие ПРИКЛАДНОЙ СЛУЖБЫ делает АГРЕГАТ независимым от РЕПОЗИТАРИЯ или СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Однако в очень сложных и

проблемно-ориентированных системах для освобождения от зависимости, возможно, лучше всего передать СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ в командный метод АГРЕГАТА. Тогда АГРЕГАТ сможет выполнить двойную диспетчеризацию, обращаясь к СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ, и разрешить ссылки. Повторим, каким бы способом один АГРЕГАТ ни получал доступ к другим АГРЕГАТАМ, ссылка на многочисленные АГРЕГАТЫ в одном запросе не дает оснований для модификации нескольких агрегатов.

Ковбойская логика

LB: Когда я гуляю ночью, у меня есть два ориентира. Если пахнет навозом, значит, я иду по направлению к стаду. Если же пахнет жареным мясом, значит, я иду домой.



Ограничение модели использовать исключительно ссылки с помощью идентификаторов затрудняет обслуживание клиентов, собирающих и прорисовывающих представления **ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (14)**. Для рассылки представлений в отдельном сценарии использования, вероятно, придется использовать многочисленные ХРАНИЛИЩА. Если затраты, связанные с выполнением запросов, снижают производительность системы, то, может быть, целесообразно рассмотреть возможность использования *тета-объединений* (theta joins) или принципа CQRS. Библиотека Hibernate, например, использует тета-объединения как средство сборки многих связанных ссылками АГРЕГАТОВ в один объединенный запрос, который может обеспечить просмотр необходимых частей. Если принцип CQRS и тета-объединения не подходят, вы, возможно, должны достичь баланса между косвенными и прямыми ссылками.

Если вам кажется, что наши советы приводят к менее удобной модели, рассмотрите дополнительные преимущества, которые они приносят. Уменьшение размеров АГРЕГАТОВ повышает производительность моделей, а также облегчает их масштабируемость и распределение.

Масштабируемость и распределение

Поскольку АГРЕГАТЫ используют не прямые ссылки на другие АГРЕГАТЫ, а ссылки по идентификаторам, их связанные состояния можно перемещать, чтобы достичь крупного масштаба. Как указывает Пат Хелланд из компании Amazon.com в своем меморандуме "Life beyond Distributed Transactions: An Apostate's Opinion" [Helland], с помощью непрерывного перераспределения хранилища АГРЕГАТОВ можно обеспечить практически бесконечную масштабируемость. То, что мы называем *АГРЕГАТОМ*, он называет *сущностью*. Просто он описывает

АГРЕГАТ под другим именем: структурный модуль, обладающий транзакционной согласованностью. Некоторые механизмы постоянного хранения NoSQL поддерживают распределенное хранение в стиле Amazon. Они обеспечивают большую часть того, что Хелланд называет низким уровнем, учитывающим масштаб. При использовании распределенного хранилища или базы данных SQL с аналогичной целью важную роль играют ссылки с помощью идентификаторов.

Распределение — это процесс рассылки данных из хранилища. Поскольку в одной ПРЕДМЕТНОЙ ОБЛАСТИ всегда существует несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, ссылки по идентификаторам позволяют распределенным моделям предметной области иметь связи с внешней средой. Если используется событийно-ориентированный обход, то за пределы предприятия рассылаются **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**, основанные на сообщениях. Подписчики на сообщения во внешних ОГРАНИЧЕННЫХ КОНТЕКСТАХ используют идентификаторы для выполнения операций в собственных моделях предметной области. Ссылка по идентификатору формирует удаленные связи, или *партнеров*. Распределенные операции управляются тем, что Хелланд называет *двусторонними действиями*, а в шаблонах **ИЗДАТЕЛЬ–ПОДПИСЧИК** [Buschmann et al.] или **НАБЛЮДАТЕЛЬ** [Гамма и др.] — *многосторонними* (две или больше). Транзакции в распределенных системах не являются атомарными. В конечном счете различные системы приводят многочисленные АГРЕГАТЫ в согласованное состояние.

Правило: используйте принцип итоговой согласованности за пределами границы

В определении шаблона АГРЕГАТ, приведенном в книге [Эванс], многие часто не обращают внимания на одно утверждение. Оно имеет отношение к тому, что мы должны обеспечить итоговую согласованность, если отдельный запрос клиента влияет на несколько АГРЕГАТОВ.

> Не всякое правило, распространяющееся на АГРЕГАТ, обязано выполняться непрерывно. Восстановить нужные взаимосвязи за определенное время можно с помощью обработки событий, пакетной обработки и других механизмов обновления системы [Эванс, p. 127].

Таким образом, если выполнение команды на одном экземпляре АГРЕГАТА подразумевает, чтобы на одном или нескольких других АГРЕГАТАХ выполнялись дополнительные бизнес-правила, используйте принцип итоговой согласованности. Признание того факта, что все экземпляры АГРЕГАТОВ на крупномасштабном предприятии с интенсивным производством никогда не являются абсолютно согласованными, поможет нам понять, что итоговая согласованность

целесообразна и в меньшем масштабе, в котором участвуют всего несколько экземпляров.

Спросите экспертов в предметной области, допускают ли они определенный интервал времени между модификацией одного экземпляра и всех остальных экземпляров. Эксперты в предметной области иногда намного снисходительнее к принципу итоговой согласованности, чем разработчики. Они знают о реальных задержках, которые все время возникают в их бизнесе, тогда как разработчики обычно склонны к атомарным изменениям. Эксперты в предметной области часто помнят времена, когда их бизнес-операции еще не были автоматизированы компьютерами, когда постоянно возникали различные виды задержек и согласованность никогда не была мгновенной. Таким образом, эксперты в предметной области часто готовы допустить разумные задержки — довольно большое количество секунд, минут, часов или даже дней, — прежде чем будет достигнута согласованность.

Существует практический способ поддерживать итоговую согласованность в модели DDD. Командный метод АГРЕГАТА публикует СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ, которое своевременно поставляется одному или более асинхронным подписчикам.

```
public class BacklogItem extends ConcurrencySafeEntity {
    ...
    public void commitTo(Sprint aSprint) {
        ...
        DomainEventPublisher
            .instance()
            .publish(new BacklogItemCommitted(
                    this.tenantId(),
                    this.backlogItemId(),
                    this.sprintId()));
    }
    ...
}
```

Каждый из этих подписчиков получает свой, но согласованный с остальными экземпляр АГРЕГАТА и выполняет его функцию. Каждый из подписчиков выполняет свои функции в рамках отдельной транзакции, соблюдая правило, в соответствии с которым можно изменять только один экземпляр АГРЕГАТА за транзакцию.

Что произойдет, если у подписчика возникнет рассогласование с другим клиентом и модификация завершится неудачей? Если подписчик не подтверждает успех модификации с помощью механизма передачи сообщений, то модификация может быть повторена. В этой ситуации выполнена повторная доставка сообщения, начата новая транзакция, предпринята новая попытка выполнить

необходимую команду и в итоге будет сделано соответствующее подтверждение. Этот процесс повторной попытки может продолжаться, пока не будет достигнута согласованность или пока не будет превышен предел повторных попыток.[6] Если происходит полный отказ, возможно, необходимо выполнить компенсацию или как минимум сообщить о "зависании".

Что происходит при публикации СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ `BacklogItemCommitted` в нашем примере? Напомним, что класс `BacklogItem` уже содержит идентификатор объекта класса `Sprint`, для которого он предназначен, и нас нисколько не интересует бессмысленная двусторонняя связь. Вместо этого СОБЫТИЕ позволяет в итоге создать экземпляр класса `CommittedBacklogItem`, чтобы экземпляр класса `Sprint` мог сделать запись о назначении работы. Поскольку каждый экземпляр класса `CommittedBacklogItem` имеет атрибут `ordering`, экземпляр класса `Sprint` может присвоить каждому экземпляру класса `BacklogItem` порядковый номер, который отличается от порядковых номеров экземпляров классов `Product` и `Release`, и этот номер никак не связан с оценкой приоритета `BusinessPriority`, записанной в экземпляре класса `BacklogItem`. Таким образом, экземпляры классов `Product` и `Release` содержат аналогичные связи, а именно — `ProductBacklogItem` и `ScheduledBacklogItem` соответственно.

Заметки на доске

- Вернитесь к своему списку крупнокластерных АГРЕГАТОВ и к нескольким АГРЕГАТАМ, которые модифицируются в рамках одной транзакции.
- Постройте диаграмму разбиения крупных кластеров. Обведите кружком и снабдите примечаниями все истинные инварианты внутри всех новых небольших АГРЕГАТОВ.
- Постройте диаграмму, демонстрирующую сохранение итоговой согласованности отдельных АГРЕГАТОВ.

---

[6] Рассмотрите возможность повторения попыток с помощью алгоритма ОГРАНИЧЕННОГО СВЕРХУ ЭКСПОНЕНЦИАЛЬНОГО ОТКАТА (CAPPED EXPONENTIAL BACK-OFF). Вместо повторения попытки через фиксированное количество секунд N, принятое по умолчанию, алгоритм экспоненциального отката выполняет попытки, пока не будет превышено предельное количество попыток. Например, алгоритм начинает повторение попыток с интервала в одну секунду, выполняет экспоненциальный откат, а затем удваивает интервал, пока не достигнет успеха или не будет превышен 32-секундный предел задержки.

В этом примере продемонстрировано использование принципа итоговой согласованности в отдельном ОГРАНИЧЕННОМ КОНТЕКСТЕ, но этот же прием можно применять и в распределенной системе, описанной выше.

Спрашивайте, чья эта обязанность

Некоторые сценарии предметной области могут чрезвычайно усложнить выбор используемого вида согласованности: транзакционной или итоговой. Те, кто используют подход DDD традиционным способом, могут сделать выбор в пользу транзакционной согласованности. Те, кто используют принцип CQRS, могут склоняться к итоговой согласованности. Какая из них правильная? Откровенно говоря, ни одна из этих тенденций не дает предметно-ориентированного ответа, отражая лишь техническое предпочтение. Есть ли лучший способ разорвать связь?

Ковбойская логика

LB: Сын сказал мне, что нашел в Интернете способ сделать коров более плодовитыми. Я ответил, что это обязанность быков.

Обсуждение этого вопроса с Эриком Эвансом выявило очень простой и надежный принцип. При исследовании сценария или истории использования спрашивайте, является ли это заданием пользователя, выполняющего сценарий использования, чтобы сделать данные согласованными. Если да, попытайтесь сделать его транзакционно согласованным, но не нарушайте другие правила работы с АГРЕГАТАМИ. Если это обязанность другого пользователя или задание системы, то придерживайтесь принципа итоговой согласованности. Это мудрое решение не только обеспечивает удобное разрывание связи, но и помогает достичь более глубокого понимания предметной области. Предложенное правило выявляет реальные системные инварианты: те, которые должны быть сохранены транзакционно согласованными. Это понимание намного более ценно, чем решение, принятое на основе технических соображений.

Это правило стоит добавить в набор ГЛАВНЫХ ПРАВИЛ ПРОЕКТИРОВАНИЯ АГРЕГАТОВ. Поскольку есть и другие факторы, которые необходимо рассмотреть, это правило не всегда позволяет сделать окончательный выбор между транзакционной и итоговой согласованностью, но обычно обеспечивает более глубокое понимание модели. Этот принцип будет использован позже, когда мы расскажем, как команда изменила границы своих АГРЕГАТОВ.

Причины нарушения правил

Опытный проектировщик, использующий принципы DDD, может иногда разрешить сохранение изменений в многочисленных экземплярах АГРЕГАТОВ в рамках одной транзакции, но для этого должна быть веская причина. Что может быть такой причиной? Я обсуждаю здесь четыре возможные причины. Вы можете предложить и другие.

Причина 1: удобство пользовательского интерфейса

Иногда пользовательские интерфейсы для удобства позволяют пользователям определять общие характеристики многих вещей сразу, чтобы создавать их пакеты. Если это происходит часто, то члены команды могут решить создать несколько списков заданий и объединить их в пакет. Пользовательский интерфейс позволяет им заполнить все общие свойства в одном разделе, а затем заполнить один за другим несколько различающихся свойств каждого элемента, устраняя повторяющиеся действия. В таком случае все новые списки заданий для спринта будут запланированы (созданы) одновременно.

```
public class ProductBacklogItemService ... {
    ...
    @Transactional
    public void planBatchOfProductBacklogItems(
         String aTenantId, String productId,
         BacklogItemDescription[] aDescriptions) {

    Product product =
         productRepository.productOfId(
                 new TenantId(aTenantId),
                 new ProductId(productId));
         for (BacklogItemDescription desc : aDescriptions) {
              BacklogItem plannedBacklogItem =
                  product.planBacklogItem(
                      desc.summary(),
                      desc.category(),
                      BacklogItemType.valueOf(
                      desc.backlogItemType()),
                      StoryPoints.valueOf(
                      desc.storyPoints()));

              backlogItemRepository.add(plannedBacklogItem);
         }
   }
   ...
}
```

Вызывает ли это проблему с инвариантами управления? В данном случае нет, поскольку не имеет значения, создаются ли списки заданий для спринта по одному или в пакете. Созданные объекты представляют собой заполненные АГРЕГАТЫ, которые поддерживают собственные инварианты. Таким образом, если создание пакета экземпляров АГРЕГАТОВ не отличается от их неоднократного создания по одному, эмпирическое правило можно безнаказанно нарушить.

Причина 2: отсутствие технических механизмов

Итоговая согласованность требует использования определенных возможностей внеполосной обработки данных, такой как пересылка сообщений, таймеры или фоновые потоки. Что произойдет, если в проекте, который вы разрабатываете, нет условий ни для одного из этих механизмов? Большинству из нас это показалось бы странным, но я сталкивался именно с таким ограничением. Что можно сделать без механизма передачи сообщений, фоновых таймеров и других возможностей потоковой обработки?

Если не проявить осторожность, то может возникнуть крупнокластерный АГРЕГАТ. В то же время оказалось бы, что мы придерживаемся правила единственной транзакции, которое, как указывалось ранее, снижает производительность и масштабируемость системы. Для того чтобы этого не произошло, можно было бы изменить АГРЕГАТЫ системы в целом, вынуждая саму модель решить наши проблемы. Мы уже рассмотрели возможность защиты спецификаций проекта, при которой остается небольшое пространство для согласования ранее непредвиденных понятий предметной области. Это не совсем предметно-ориентированный подход, но иногда так действительно происходит. Условия могут не допускать разумного способа для изменения среды моделирования в нашу пользу. В таких случаях динамика проекта может вынудить нас изменить два или больше экземпляров АГРЕГАТА в рамках одной транзакции. Однако каким бы очевидным ни казалось это решение, его нельзя принимать второпях.

Ковбойская логика

AJ: Если вы считаете, что правила создаются, чтобы их нарушать, лучше поищите хорошего мастера по ремонту.

Рассмотрим еще один фактор, который еще больше мог бы спровоцировать нарушение правил: *структурная близость пользователя и агрегата* (user-aggregate affinity). Допускают ли потоки бизнес-операций, что в любой момент времени на

одном наборе экземпляров АГРЕГАТОВ фокусируется только один пользователь? Обеспечение структурной близости пользователя и агрегата делает решение изменять многочисленные экземпляры АГРЕГАТОВ в рамках единственной транзакции более обоснованным, потому что это позволяет предотвратить нарушение инвариантов и транзакционные коллизии. Даже при структурной близости пользователя и агрегата конфликты при параллельной работе возникают редко. И все же было бы желательно защитить каждый АГРЕГАТ от этого с помощью оптимистичного параллелизма. Так или иначе, конфликты при параллельной работе могут произойти в любой системе, причем это происходит еще более часто, когда структурная близость пользователя и агрегата не является нашим союзником. Кроме того, восстановление после конфликтов при параллельной работе не вызывает проблем, если это событие происходит редко. Таким образом, под давлением обстоятельств имеет смысл изменять многочисленные экземпляры АГРЕГАТОВ в одной транзакции.

Причина 3: глобальные транзакции

Другой фактор, заслуживающий внимания, — это влияние устаревших технологий и правил, принятых на предприятии. Например, предприятие может строго придерживаться использования глобальных транзакций с двухфазной фиксацией. Это одна из тех ситуаций, которых невозможно избежать, по крайней мере в ближайшей перспективе.

Даже если приходится использовать глобальную транзакцию, вы не обязаны одновременно изменять многочисленные экземпляры АГРЕГАТА в своем локальном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Если вы можете избежать этого, то по крайней мере вы сможете предотвратить транзакционную конкуренцию в своем СМЫСЛОВОМ ЯДРЕ и фактически выполнить правила проектирования АГРЕГАТОВ. Недостатком глобальных транзакций является то, что ваша система, вероятно, никогда не будет масштабироваться, хотя это было бы возможно, если бы вы смогли избежать двухфазных фиксаций и связанной с ними мгновенной согласованности.

Причина 4: производительность запросов

Иногда лучше хранить прямые объектные ссылки на другие АГРЕГАТЫ. С их помощью можно облегчить выполнение запросов к ХРАНИЛИЩУ. Принимая это решение, следует тщательно взвесить все "за" и "против" в свете потенциального размера и последствий для общей производительности системы. Один пример нарушения правила ссылки по идентификатору приводится ниже в этой главе.

Выполнение правил

Проектные решения, касающиеся интерфейса пользователя, технические ограничения, жесткие правила и другие факторы могут вынудить вас пойти на компромиссы. Разумеется, мы не ищем повода отказаться от выполнения ГЛАВНЫХ ПРАВИЛ ПРОЕКТИРОВАНИЯ АГРЕГАТОВ. В долгосрочной перспективе выполнение этих правил принесет пользу вашему проекту. Благодаря им мы обеспечим требуемую согласованность, оптимальную производительность и высокую масштабируемость системы.

Понимание через открытие

Посмотрим, как соблюдение правил проектирования АГРЕГАТОВ повлияло на проект модели Scrum в компании SaaSOvation. Мы увидим, как проектировщики пересмотрели проект заново, применяя новые методы. Эти усилия привели к новому пониманию модели. Для этого проектировщики выдвигали разные идеи, а потом заменяли их новыми.

Пересмотр проекта

После рефакторинга, в ходе которого был разрушен крупнокластерный класс `Product`, класс `BacklogItem` остался наедине со своим АГРЕГАТОМ. Эта ситуация изображена на рис. 10.7. Внутри АГРЕГАТА `BacklogItem` команда образовала коллекцию экземпляров класса `Task`. Каждый экземпляр класса `BacklogItem` содержит глобальный уникальный идентификатор класса `BacklogItemId`. Все связи с другими АГРЕГАТАМИ устанавливаются по идентификаторам. Это значит, что ссылка на экземпляр родительского класса `Product`, на экземпляр класса `Release`, запланированный для него, а также на экземпляр класса `Sprint`, для которого он зарегистрирован, осуществляется по идентификатору. Этот АГРЕГАТ кажется довольно маленьким.

Насколько далеко может зайти команда, решив проектировать небольшие АГРЕГАТЫ?

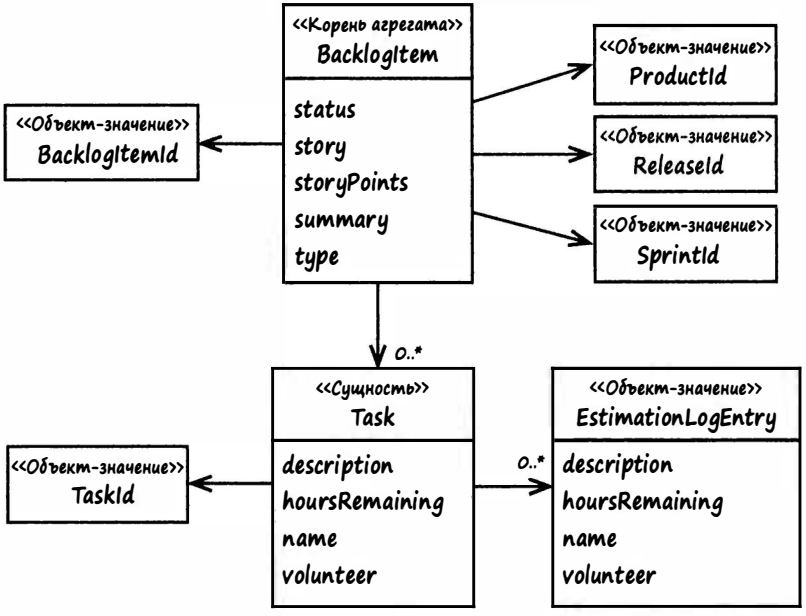


Рис. 10.7. АГРЕГАТ `BacklogItem`, имеющий внешние связи по идентификаторам, выходящие за его границы

Несмотря на удовлетворенность результатами предыдущей итерации, команда все еще испытывала некоторое беспокойство. Например, атрибут `story` позволял хранить много текста. Команды, выполняющие гибкое проектирование, не пишут многословные сочинения. Несмотря на это в проекте существует дополнительный компонент редактора, поддерживающий длинные описания сценариев использования. Они могут занимать многие тысячи байтов. Эти возможные издержки достойны пристального внимания.

Столкнувшись с возможными накладными расходами и сделав ошибки при разработке крупнокластерного класса `Product` (см. рис. 10.1 и 10.3), команда намеревалась уменьшить размер каждого АГРЕГАТА в ОГРАНИЧЕННОМ КОНТЕКСТЕ. Перед ней встали крайне важные вопросы. Существует ли истинный инвариант между классами `BacklogItem` и `Task`, который должно поддерживать данное отношение? Нет ли еще одного сценария, в котором данную связь также можно было бы разорвать, безопасно образовав два отдельных АГРЕГАТА? Какова была бы общая стоимость проекта?

Правильное решение можно найти с помощью ЕДИНОГО ЯЗЫКА. Команда сформулировала следующие инварианты.

- Когда достигается прогресс в разработке списка заданий для спринта, член команды оценивает количество оставшихся рабочих часов.
- Если времени до завершения конкретной задачи не осталось, то проверяется время всех оставшихся задач в текущем списке заданий. Если времени не осталось ни для одной задачи, то состояние списка заданий автоматически изменяется и принимает значение "сделано".
- Если член команды приходит к выводу, что для выполнения конкретной задачи остался один час или более, а состояние списка заданий равно значению "сделано", то оно автоматически отменяется.

Это очень похоже на истинный инвариант. Правильное состояние списка заданий корректируется автоматически и полностью зависит от общего количества часов, оставшегося для выполнения всех его задач. Если общее количество часов, отведенных на выполнение задачи, и состояние списка заданий должны оставаться согласованными, то кажется, что рис. 10.7 действительно предусматривает корректную границу согласованности АГРЕГАТА. Однако команда все еще должна определить стоимость кластера с точки зрения производительности и масштабируемости. Этот показатель необходимо сравнить с экономией, которую можно получить, если оставить состояние списка заданий согласованным с общим количеством часов, оставшихся на выполнение задачи.

Может показаться, что перед нами классическая возможность использовать принцип итоговой согласованности, но мы не можем просто принять этот вывод. Необходимо проанализировать подход, основанный на согласованности транзакций, а затем исследовать то, что могло быть выполнено на основе принципа итоговой согласованности. Тогда мы сможем сделать самостоятельный вывод о том, какой подход является более предпочтительным.

Оценка стоимости агрегата

Как показано на рис. 10.7, каждый экземпляр класса `Task` содержит коллекцию экземпляров класса `EstimationLogEntry`. Эти журналы моделируют конкретные ситуации, в которых члены команды вводят новую оценку оставшихся часов работы над задачей. С практической точки зрения имеет значение, сколько элементов класса `Task` будет храниться в каждом экземпляре класса `BacklogItem` и сколько элементов класса `EstimationLogEntry` будет содержать экземпляр класса `Task`? Точно ответить на эти вопросы трудно. Это зависит от сложности каждой задачи и продолжительности спринта. Однако нам помогут приблизительные вычисления (back-on-the-envelope — BOTE) [Bentley].

Часы, отведенные на выполнение задачи, обычно повторно оцениваются каждый день после того, как член команды поработает над данной задачей. Допустим, что продолжительность большинства спринтов — две или три недели. Будут и

более длинные спринты, но самые распространенные — двух- и трехнедельные. Давайте выберем какое-нибудь количество дней между 10 и 15. Не стремясь к особой точности, будем считать, что 12 дней работы достаточно, потому что двухнедельных спринтов на самом деле больше, чем трехнедельных.

Затем подсчитаем количество часов, предназначенных для выполнения каждой задачи. Помня, что задачи должны быть разделены на управляемые модули, мы обычно используем оценку в интервале от 4 до 16 часов. Обычно, если время выполнения задачи превышает 12-часовую оценку, эксперты по Scrum предлагают разбивать ее на более мелкие подзадачи. Однако использование 12-часовой оценки в качестве первого приближения упрощает равномерное распределение работы. Например, мы можем сказать, что работаем над задачей в течение одного часа в каждый из 12 дней спринта. Этот подход позволяет справляться с более сложными задачами. Таким образом, мы примем 12-часовую оценку времени выполнения задачи, предполагая, что вначале на выполнение каждой задачи выделяется 12 часов.

Остается вопрос: сколько задач приходится на один список заданий? На этот вопрос очень трудно ответить. Мы могли бы подумать о двух или трех задачах на один **УРОВЕНЬ (4)** или рассмотреть **ГЕКСАГОНАЛЬНЫЙ ПОРТ-АДАПТЕР (4)** для данного функционального аспекта. Например, мы могли бы оценить **УРОВЕНЬ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (14)** тремя задачами, **УРОВЕНЬ ПРИЛОЖЕНИЙ (14)** — двумя, УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ — тремя и **УРОВЕНЬ ИНФРАСТРУКТУРЫ (14)** — тремя. Это привело бы в итоге к 11 задачам. Возможно, оценка могла бы быть более точной, но мы и так уже допустили ошибку при оценке количества задач. Давайте для простоты округлим эту оценку до 12 задач на один список заданий. Итак, мы допускаем 12 задач, у каждой из них 12 журналов оценок, т.е. в итоге мы получаем *144 объекта на один список заданий*. Возможно, это больше обычного, но это лишь примерная оценка.

Есть и другая переменная, которую следует рассмотреть. Если бы специалист по системе Scrum посоветовал определить более мелкие задачи, то ситуация несколько изменилась бы. Удвоение количества задач (24) и сокращение вдвое количества записей в журналах оценок (6) все еще давали бы в итоге 144 объекта. Однако это вынудило бы загружать вдвое больше задач (24, а не 12) во время выполнения всех запросов на оценку, требуя больше памяти для каждого запроса. Команда попробует различные комбинации, чтобы увидеть, насколько существенно они влияют на производительность. Однако для начала члены команды будут рассматривать 12 задач, на выполнение каждой из которых отведено 12 часов.

Типичные сценарии использования

Теперь важно рассмотреть типичные сценарии использования. Как часто при выполнении одного пользовательского запроса возникает необходимость загружать в память все 144 объекта сразу? Происходит ли это вообще когда-нибудь? Кажется, что нет, но команда должна это проверить. В противном случае какова вероятная верхняя оценка количества объектов? Кроме того, необходимо выяснить, как часто возникает одновременная работа нескольких клиентов, вызывающая конкуренцию за список заданий? Давайте посмотрим.

Следующие сценарии основываются на использовании библиотеки Hibernate в качестве механизма постоянного хранения данных. Кроме того, у каждого типа СУЩНОСТИ есть собственный атрибут версии оптимистичного параллелизма. Это вполне осуществимо, потому что инвариант изменения состояния управляется КОРНЕВОЙ СУЩНОСТЬЮ `BacklogItem`. Когда состояние автоматически изменяется (к сделанному или назад к назначенному), корневая версия удаляется. Таким образом, изменения в задачах могут происходить независимо друг от друга и не влиять на КОРЕНЬ, если в результате не происходит изменение состояния. (Если в качестве механизма постоянного хранения данных используется хранилище документов, следующий анализ можно пересмотреть, так как в этом случае КОРЕНЬ изменяется каждый раз при изменении собранной части.)

Когда список заданий создается впервые, количество задач равно нулю. Обычно задачи определяются только в ходе планирования спринта на совещании команды. После выявления каждой задачи член команды добавляет ее к соответствующему списку заданий. Двум членам команды нет никакой необходимости спорить друг с другом об АГРЕГАТЕ, словно мчась наперегонки, чтобы выяснить, кто сможет быстрее ввести новые задачи. Это вызвало бы коллизию, и тогда один из двух запросов перестал бы работать (по той же причине ранее завершались неудачей попытки одновременно добавить разные части класса `Product`). Однако эти два члена команды, вероятно, скоро и сами выяснили бы, насколько непродуктивной является их избыточная работа.

Если бы разработчики выяснили, что многочисленные пользователи действительно регулярно хотят одновременно добавлять задачи, то это значительно изменило бы анализ. Это понимание могло бы сразу склонить чашу весов в пользу разделения классов `BacklogItem` и `Task` на два отдельных АГРЕГАТА. С другой стороны, это могло бы также быть идеальным моментом для настройки отображения Hibernate и установки флага `optimistic-lock` равным `false`. В этом случае целесообразно разрешить одновременное увеличение задач, особенно если они не влияют на масштабируемость и производительность.

Если оценка первоначального времени выполнения задач равна нулю и позднее будет уточняться, то мы все еще не склонны допускать параллельную

конкуренцию, несмотря на то что это добавило бы всего одну дополнительную запись в журнал оценки, увеличив нашу примерную оценку до 13. Одновременное использование в данном случае не изменяет состояние списка заданий. Состояние изменяется на "сделано", только если время выполнения задачи снижается до нуля, или изменяется на "поставлена", если задача уже выполнена, а время изменилось от нуля до единицы или больше, — это два редких события.

Ежедневные оценки вызывают проблемы? В первый день спринта журнал оценок для конкретной задачи из списка заданий обычно содержит нулевые записи. В конце дня любой желающий член команды, работающий над задачей, уменьшает предполагаемые часы работы на единицу. Это добавляет новый журнал оценок для каждой задачи, но состояние списка заданий остается незатронутым. Состязание за задачу не возникает, потому что время ее выполнения корректирует только один член команды. Однако такая ситуация сохраняется только до 12-го дня, когда мы достигаем точки изменения состояния. Поскольку время выполнения каждой из любых 11 задач уменьшено до нуля часов, состояние списка заданий не изменяется. Только самая последняя оценка, 144-я на 12-й задаче, вызывает автоматический переход состояния в значение "сделано".

Этот анализ привел команду к важному выводу. Даже если бы они изменили сценарии использования, вдвое ускорив выполнение задачи (до шести дней), или даже полностью его переделали, ничего не изменилось бы. К изменению состояния, которое, в свою очередь, изменяет КОРЕНЬ, приводит только заключительная оценка. Этот подход кажется вполне безопасным, несмотря на то что затраты памяти мы пока не оценили.

Затраты памяти

Теперь рассмотрим затраты памяти. Здесь важно то, что оценки регистрируются по датам как ОБЪЕКТЫ-ЗНАЧЕНИЯ. Если член команды несколько раз выполняет переоценку в течение дня, то сохраняется только последняя оценка. Последнее ЗНАЧЕНИЕ, соответствующее той же дате, заменяет предыдущее в этой коллекции. Пока мы не требуем отслеживать ошибки при оценке задачи. Будем предполагать, что количество записей в журнале оценок задачи никогда не будет превышать продолжительность спринта, измеренную в днях. Если задачи были определены за один или несколько дней до планирования спринта и часы их выполнения были повторно оценены в любой из этих предшествующих дней, то это предположение не выполняется. В таком случае для каждого прошедшего дня следует завести отдельный журнал оценок.

Что можно сказать об общем количестве задач и оценок, хранящихся в памяти при каждой переоценке? При отложенной загрузке задач и журналов оценок в каждый момент времени в памяти находилось бы 12 плюс 12 объектов из коллекции на запрос. Это вызвано тем, что при обращении к коллекции были бы загружены все 12 задач. Для того чтобы добавить последнюю запись в журнал оценок одной из задач, мы должны были бы загрузить коллекцию записей в журнале оценок. В такой коллекции может быть до 12 объектов. И наконец, для одного АГРЕГАТА требуется один список заданий, 12 задач, 12 записей в журнале, т.е. максимум 25 объектов. Это не очень много; это небольшой АГРЕГАТ. Кроме того, верхняя оценка количества объектов (например, 25) не достигается до последнего дня спринта. Таким образом, на протяжении большой части спринта АГРЕГАТ будет еще меньше.

Не возникнут ли проблемы с производительностью из-за отложенных загрузок? Это возможно, потому что фактически требуются две отложенные загрузки: одна — для задач и одна — для записей в журнале оценок для одной из задач. Команда должна будет выполнить тестирование, чтобы исследовать возможные издержки многократных выборок.

Есть еще одно обстоятельство. Технология Scrum позволяет командам экспериментировать, чтобы идентифицировать правильную модель планирования. Как указано в работе [Sutherland], опытные команды, хорошо знающие скорость своей работы, могут оценивать время выполнения пунктов сценариев использования, а не часы, требуемые для выполнения задачи. Во время определения каждой задачи они могут присвоить каждой задаче всего один час. Во время спринта они могут только один раз повторно оценить задачу, изменяя время ее выполнения с одного часа до нуля, когда задача выполнена. Поскольку эти вопросы относятся к проектированию АГРЕГАТА, использование пунктов сценариев уменьшает общее количество журналов оценок на задачу до всего одного журнала и почти устраняет перерасход памяти.

Позже, исследуя реальные производственные данные, разработчики ProjectOvation смогут аналитически определить, сколько фактических задач и записей в журнале оценок (в среднем) приходится на один список заданий.

Предшествующего анализа было достаточно, чтобы заставить команду проверить свои примерные расчеты. Однако, имея неокончательные результаты, они решили, что переменных было все еще слишком много, чтобы быть уверенными в успехе проекта. Неизвестных факторов было достаточно, чтобы рассмотреть альтернативный проект.

Исследование альтернативного проекта

Существует ли другой вариант проекта, который позволил бы точнее определить границы агрегата в соответствии со сценариями использования?

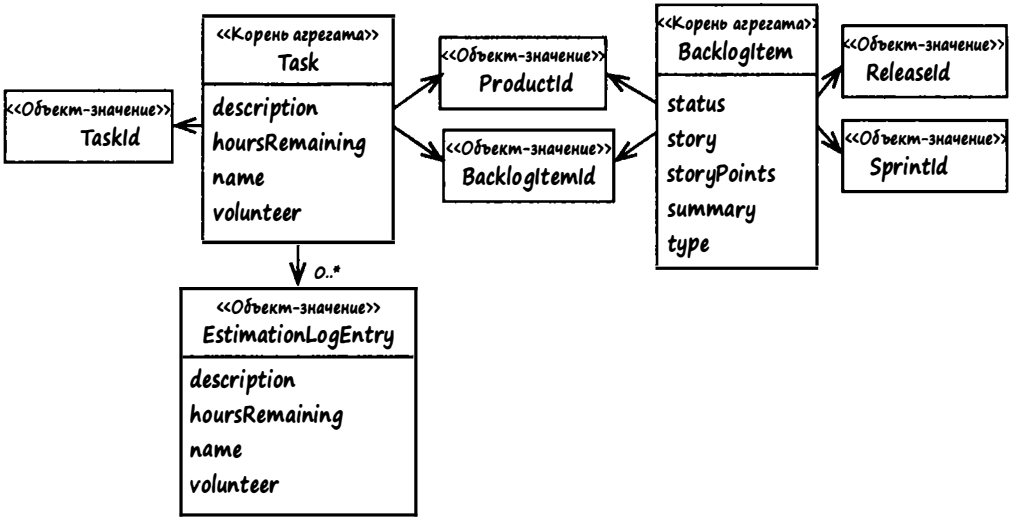


Рис. 10.8. Классы `BacklogItem` и `Task`, моделируемые как отдельные АГРЕГАТЫ

Для полноты картины команда решила выяснить, что следовало бы сделать, чтобы класс `Task` стал независимым АГРЕГАТОМ, и понять, принесет ли это реальную пользу. Их предложение показано на рис. 10.8. Его реализация уменьшила бы состав композиции в верхней части на 12 объектов и уменьшило бы объем ленивой загрузки. Фактически эта схема дала им возможность оценить объем немедленной загрузки записей из журнала оценок во всех случаях, когда она обеспечивает максимальную производительность.

Разработчики согласились не изменять отдельные АГРЕГАТЫ — `Task` и `BacklogItem` — в рамках одной и той же транзакции. Теперь они должны были выяснить, возможно ли выполнить необходимое автоматическое изменение состояния за приемлемый период времени. Можно было бы ослабить требование согласованности инварианта, так как состояние не могло оставаться согласованным в рамках транзакции. Допустимо ли это? Они обсудили этот вопрос с экспертами в проблемной области и выяснили, что эксперты допускают некоторую задержку между заключительной оценкой времени выполнения, когда она становится равной нулю, и присвоением состоянию значения "сделано", и наоборот.

Обеспечение итоговой согласованности

Похоже, возникла возможность обоснования итоговой согласованности между отдельными АГРЕГАТАМИ. Посмотрим, как это можно сделать.

Когда экземпляр класса `Task` выполняет команду `estimateHoursRemaining()`, он публикует соответствующее СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ. Теперь команда может использовать это СОБЫТИЕ для достижения итоговой СОГЛАСОВАННОСТИ. СОБЫТИЕ моделируется со следующими свойствами.

```
public class TaskHoursRemainingEstimated implements DomainEvent {
    private Date occurredOn;
    private TenantId tenantId;
    private BacklogItemId backlogItemId;
    private TaskId taskId;
    private int hoursRemaining;
    ...
}
```

Специализированный подписчик теперь мог бы прослушивать СОБЫТИЯ и делегировать их СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ для координации согласованной обработки. СЛУЖБА могла бы выполнять следующие действия.

- Использовать экземпляр класса `BacklogItemRepository` для извлечения идентифицированного объекта класса `BacklogItem`.
- Использовать экземпляр класса `TaskRepository` для извлечения всех экземпляров класса `Task`, связанных с идентифицированным экземпляром класса `BacklogItem`.
- Выполнить команду класса `BacklogItem` с именем `estimateTaskHoursRemaining()`, передав ей атрибут `hoursRemaining` СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ и извлеченные экземпляры класса `Task`. Экземпляр класса `BacklogItem` может изменять свое состояние в зависимости от параметров.

Команда должна найти способ оптимизировать эту схему. Эта трехэтапная схема требует загрузки всех экземпляров класса `Task` каждый раз, когда возникает необходимость в повторной оценке. Используя приблизительную оценку, приходим к выводу, что 143 из 144 объектов являются ненужными. Оптимизировать эту схему довольно легко. Вместо использования ХРАНИЛИЩА для того, чтобы получить все экземпляры класса `Task`, достаточно просто попросить его просуммировать часы выполнения всех задач в базе данных.

```
public class HibernateTaskRepository implements TaskRepository {
    ...
    public int totalBacklogItemTaskHoursRemaining(
            TenantId aTenantId,
            BacklogItemId aBacklogItemId) {
```

```
        Query query = session.createQuery(
            "select sum(task.hoursRemaining) from Task task "
            + "where task.tenantId = ? and "
            + "task.backlogItemId = ?");
        ...
    }
}
```

Итоговая согласованность немного усложняет пользовательский интерфейс. Как пользовательский интерфейс выводил бы на экран новое состояние, если изменение состояния невозможно выполнить в течение нескольких сотен миллисекунд? Может быть, следует поместить бизнес-логику внутрь представления, чтобы определить текущее состояние? Это стало бы антишаблоном интеллектуального пользовательского интерфейса. Возможно, представление должно было бы просто выводить на экран устаревшее состояние и позволить пользователям смириться с визуальным несоответствием. Это можно было бы считать ошибкой. По крайней мере, это может очень раздражать пользователей.

Представление могло бы использовать запрос Ajax в фоновом режиме, но это было бы довольно неэффективно. Поскольку компонент представления не может точно определить момент, когда именно необходимо изменить состояние, большинство запросов Ajax оказались бы ненужными. Приблизительные вычисления показывают, что 143 из 144 повторных оценок не привели бы к изменению состояния, а значит, на ярусе веб-узлов выполнялось бы большое количество избыточных запросов. При правильной серверной поддержке клиенты могли бы использовать механизм Comet (иначе говоря, Ajax Push). Несмотря на то что решение этой проблемы было бы хорошим поводом для внедрения новой технологии, у команды не было опыта ее использования.

С другой стороны, лучшее решение — это простейшее решение. Команда могла бы поместить на экран визуальный индикатор, который сообщал бы пользователю, что текущее состояние не определено. Представление могло бы выделить период времени для его перепроверки или обновления. В качестве альтернативы измененное состояние можно было бы показывать на следующем прорисованном представлении. Это безопасно. Команда должна была бы выполнить некоторые тесты приемлемости для пользователя, но все выглядело обнадеживающим.

Должен ли член команды делать эту работу

До сих пор мы совершенно не обращали внимания на один очень важный вопрос: кто именно должен согласовывать состояние списка заданий со временем

выполнения всех остальных задач? Должен ли член команды, использующей систему Scrum, сам позаботиться о том, чтобы состояние родительского списка заданий принимало значение "сделано" сразу после того, как он установит оставшееся время работы равным нулю? Всегда ли члены команды будут знать, что работают с последней задачей, время выполнения которой не равно нулю? Возможно, они будут знать об этом и, возможно, каждый член команды будет обязан официально завершить работу над каждым списком заданий.

С другой стороны, что если в процессе участвует другой участник проекта? Например, владелец продукта или кто-то другой может пожелать проверить, удачно ли завершена работа над списком заданий. Возможно, кто-то захочет сначала использовать эту возможность на сервере непрерывной интеграции. Если остальные сотрудники будут довольны заявлением разработчика о завершении работы, то они вручную присвоят состоянию значение "сделано". Разумеется, это все меняет, делая ненужными ни транзакционную, ни итоговую согласованность. Задачи можно отделить от их родительских списков заданий, поскольку новый сценарий использования это допускает. Однако, если именно член команды обязан обеспечить автоматическое присвоение состоянию значения "сделано", то это значит, что задачи внутри списка заданий допускают транзакционную согласованность. Интересно, что на этот вопрос нет четкого ответа. Вероятно, это означает, что такая возможность должна быть дополнительным преимуществом приложения.

Оставив задачи внутри их списков заданий, можно решить проблему согласованности, сохранив при этом возможность изменять состояние как автоматически, так и вручную.

Это полезное упражнение раскрывает совершенно новый аспект предметной области. Кажется, что команды должны иметь возможность конфигурировать параметры рабочего процесса. Пока команда не была готова реализовать эту возможность и решила отложить ее на будущее. *Вопрос "кто должен это делать?" привел их к очень важным выводам об их предметной области.*

Затем в качестве альтернативы описанному выше анализу один из разработчиков выдвинул очень практичное предложение. Если их так сильно заботят возможные затраты памяти, связанные с атрибутом `story`, то почему бы не предпринять что-нибудь конкретное, чтобы решить эту проблему? Например, можно было бы уменьшить общую емкость атрибута `story` и в дополнение создать новое свойство `useCaseDefinition`. Их можно спроектировать так, чтобы существовала возможность отложенной загрузки, поскольку большую часть времени они не будут

использоваться. Кроме того, можно было бы даже разработать отдельный АГРЕГАТ, загружая его при необходимости. Эта идея натолкнула команду на мысль, что настал удобный момент нарушить правило ссылки на внешний АГРЕГАТ только по идентификатору. Кажется, возникла удобная возможность использовать прямую ссылку на объект и объявить объектно-реляционное отображение так, чтобы выполнить его отложенную загрузку. Возможно, это имеет смысл.

Время принимать решения

Анализ такого уровня невозможно проводить весь день. Необходимо принять решение. Это не значит, что выбор одного направления исключает возможность передумать и выбрать другое решение. В данный момент широта взглядов важнее прагматизма.

Основываясь на результатах анализа, команда решила не отделять класс `Task` от класса `BacklogItem`. Ее члены не уверены, что в данный момент отделение заслуживает внимания, учитывая дополнительные усилия, риск оставить незащищенным инвариант задачи и необходимость смириться с тем, что представление будет демонстрировать пользователям устаревшее состояние. В данный момент АГРЕГАТ довольно мал. Даже если в худшем случае будет загружено 50 объектов, а не 25, размер кластера останется разумным. *В текущий момент команда запланировала работу над специализированным хранилищем сценариев использования.* Это принесло бы много пользы. В то же время, если команда решит все-таки отделить класс `Task` от класса `BacklogItem`, то риск немного возрастет, потому что объем работы немного увеличится как в данный момент, так и в будущем.

На всякий случай команда оставила возможность разделить классы. В будущем команда сможет принять окончательное решение, основываясь на получившемся проекте, его производительности и результатах проверки загрузки, а также после оценки впечатления, которое пользователи получат от итоговой согласованности состояния. Приблизительные оценки могут доказать, что решение было ошибочным, если АГРЕГАТ окажется больше, чем ожидалось. В этом случае команде ничего не останется, как разделить его на две части.

А какое решение приняли бы вы, если бы были членом команды ProjectOvation? Не уклоняйтесь от размышлений, описанных в этой врезке. На это потребуется в среднем 30 минут, в худшем случае — 60. Затраченное время компенсируется более глубоким пониманием СМЫСЛОВОГО ЯДРА.

Реализация

Важные факторы, описанные в данном разделе, влияют на надежность реализации и более глубоко исследуются в главах, в которых изучаются **СУЩНОСТИ (5)**, **ОБЪЕКТЫ-ЗНАЧЕНИЯ (6)**, **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**, **МОДУЛИ (9)**, **ФАБРИКИ (11)** и **ХРАНИЛИЩА (12)**. Этот раздел можно использовать в качестве отправной точки.

Создайте корневую сущность с уникальным идентификатором

Моделируйте одну из СУЩНОСТЕЙ как КОРЕНЬ АГРЕГАТА. Примерами КОРНЕВЫХ СУЩНОСТЕЙ в предыдущих примерах были классы `Product`, `BacklogItem`, `Release` и `Sprint`. Если команда решит отделить класс `Task` от класса `BacklogItem`, класс `Task` также может стать КОРНЕМ.

Уточненная модель `Product` может привести к следующему определению КОРНЕВОЙ СУЩНОСТИ.

```
public class Product extends ConcurrencySafeEntity {
    private Set<ProductBacklogItem> backlogItems;
    private String description;
    private String name;
    private ProductDiscussion productDiscussion;
    private ProductId productId;
    private TenantId tenantId;
    ...
}
```

Класс `ConcurrencySafeEntity` — это **СУПЕРТИП УРОВНЯ** [Fowler, P of EAA], используемый для управления суррогатным идентификатором и версиями оптимистичного параллелизма, как указано в главе, посвященной **СУЩНОСТЯМ (5)**.

В КОРЕНЬ добавляется класс `Set`, содержащий экземпляры класса `ProductBacklogItem`. Этот класс нами пока не рассматривался. Он имеет специальное предназначение. Это не просто коллекция экземпляров класса `BacklogItem`. Он должен поддерживать отдельный порядок списков заданий.

Каждый КОРЕНЬ должен проектироваться с глобальным уникальным идентификатором. Класс `Product` моделировался с типом ЗНАЧЕНИЯ, которое называлось `ProductId`. Этот тип является идентификатором предметной области и отличается от суррогатного идентификатора, предоставляемого классом `ConcurrencySafeEntity`. Способ проектирования, размещения в памяти и поддержки идентификатора описан в главе, посвященной **СУЩНОСТЯМ (5)**. Реализация класса `ProductRepository` содержит метод `nextIdentity()`, генерирующий экземпляр класса `ProductId` как идентификатор UUID.

```
public class HibernateProductRepository implements ProductRepository {
    ...
    public ProductId nextIdentity() {
        return new ProductId(java.util.UUID.randomUUID().
                .toString().toUpperCase());
    }
    ...

}
```

С помощью метода `nextIdentity()` клиент ПРИКЛАДНОЙ СЛУЖБЫ может создать экземпляр класса `Product` с глобальным уникальным идентификатором.

```
public class ProductService ... {
    ...
    @Transactional
    public String newProduct(
        String aTenantId, aProductName, aProductDescription) {
        Product product =
            new Product(
                new TenantId(aTenantId),
                this.productRepository.nextIdentity(),
                "My Product",
                "This is the description of my product.",
                new ProductDiscussion(
                    new DiscussionDescriptor(
                        DiscussionDescriptor.UNDEFINED_ID),
                        DiscussionAvailability.NOT_REQUESTED));

        this.productRepository.add(product);

        return product.productId().id();
    }
    ...
}
```

ПРИКЛАДНАЯ СЛУЖБА использует класс `ProductRepository` как для генерации идентификатора, так и для постоянного хранения нового экземпляра класса `Product`. Он возвращает простое представление нового экземпляра класса `ProductId` в виде объекта класса `String`.

Полезные части ОБЪЕКТОВ-ЗНАЧЕНИЙ

По возможности содержимое АГРЕГАТА следует моделировать как ОБЪЕКТ-ЗНАЧЕНИЕ, а не как СУЩНОСТЬ. Лучше всего выбирать такое содержимое, полная замена которого не вызывает значительной перегрузки модели или инфраструктуры.

Текущая модель Product имеет два простых атрибута и три свойства с типами ЗНАЧЕНИЙ. И description, и name — это атрибуты типа String, которые можно полностью заменить. ЗНАЧЕНИЯ productId и tenantId поддерживаются как постоянные идентификаторы; иначе говоря, после их создания они никогда не изменяются. Они содержат ссылку по идентификатору, а не непосредственно на объект. На самом деле АГРЕГАТ Tenant, на который они ссылаются, даже не находится в том же самом ограниченном контексте, и поэтому ссылаться на него можно только по идентификатору. Свойство productDiscussion соответствует принципу итоговой согласованности и имеет тип ЗНАЧЕНИЙ. При первом создании экземпляра класса Product может потребоваться обсуждение, которое впоследствии больше не понадобится. Этот экземпляр должен создаваться в *Контексте сотрудничества*. После завершения процесса создания экземпляра в другом ОГРАНИЧЕННОМ КОНТЕКСТЕ в экземпляре класса Product устанавливаются идентификатор и состояние.

Класс ProductBacklogItem должен моделироваться как СУЩНОСТЬ, а не как ЗНАЧЕНИЕ. Это объясняется весомыми причинами. Как указывалось в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**, поскольку для обращения к базам данных, стоящих за сценой, используется библиотека Hibernate, коллекции ЗНАЧЕНИЙ необходимо моделировать как сущности базы данных. Изменение порядка следования хотя бы одного элемента может привести к удалению и перемещению большого количества экземпляров класса ProductBacklogItem, а возможно, и всех экземпляров этого класса. Это может вызвать значительную перегрузку инфраструктуры. СУЩНОСТЬ позволяет использовать атрибут ordering, который изменяется во всех элементах коллекции так часто, как потребует владелец продукта. Однако если мы откажемся от библиотеки Hibernate и системы MySQL в пользу хранилища типа "ключ–значение", то сможем изменить решение и легко сделать класс ProductBacklogItem типом ЗНАЧЕНИЙ. При использовании хранилищ типа "ключ–значение" или документных хранилищ экземпляры АГРЕГАТА обычно сериализуются как одно представление значения на хранилище.

Использование закона Деметры и принципов совмещения данных и поведения

ЗАКОН ДЕМЕТРЫ (LAW OF DEMETER) [Appleton, LoD] и **СОВМЕЩЕНИЕ ДАННЫХ И ПОВЕДЕНИЯ (TELL, DON'T ASK)** [PragProg, TDA] — это принципы проектирования, которые можно применять при реализации АГРЕГАТОВ и в которых основной упор делается на сокрытии информации. Рассмотрим эти высокоуровневые принципы, чтобы показать, какую выгоду можно извлечь из их применения.

- *Закон Деметры.* Этот закон делает акцент на *принципе минимального знания*. Представьте себе объект *клиента* и другой объект, который клиент

использует для выполнения какой-нибудь системной функции; назовем этот второй объект *сервером*. Когда клиент использует сервер, он должен знать о структуре сервера как можно меньше. Атрибуты и свойства сервера — его форма — должны оставаться совершенно неизвестными клиенту. Клиент может попросить сервер выполнить команду, которая объявлена в его внешнем интерфейсе. Однако клиент не должен проникать на сервер, запрашивать у него какую-то его внутреннюю часть, а затем применять к этой части команду. Если клиенту нужна услуга, которая предоставляется внутренними частями сервера, то клиент не должен получать доступ к этим внутренним частям, для того чтобы попросить их выполнить функцию. Вместо этого сервер должен лишь предоставить внешний интерфейс и после обращения к нему делегировать вызов соответствующим внутренним частям, реализующим его интерфейс.

- Вот как вкратце формулируется закон Деметры: любой метод любого объекта может вызывать методы только из следующих объектов: 1) из своего объекта, 2) из переданных ему параметров, 3) из любого созданного им объекта, 4) из автономных объектов, к которым у него есть прямой доступ.
- *Принцип совмещения данных и поведения* (приказывай, а не спрашивай). Этот принцип просто требует сообщать объектам, что они должны делать. Часть "не спрашивай" применяется к клиентам: клиентский объект не должен спрашивать у серверного объекта о его внутренних частях, принимать решение на основе состояния, которое ему удалось выяснить, а затем просить серверный объект сделать что-то. Вместо этого клиент должен "приказывать" серверу, что тот должен делать, используя команду открытого интерфейса сервера. Этот принцип очень похож на закон Деметры, но проще в реализации.

Применим эти два принципа проектирования к классу `Product`.

```
public class Product extends ConcurrencySafeEntity {
    ...
    public void reorderFrom(BacklogItemId anId, int anOrdering) {
        for (ProductBacklogItem pbi : this.backlogItems()) {
            pbi.reorderFrom(anId, anOrdering);
        }
    }

    public Set<ProductBacklogItem> backlogItems() {
        return this.backlogItems;
    }
    ...
}
```

Класс Product требует, чтобы клиент использовал его метод reorderFrom() для выполнения команды backlogItems(), изменяющей состояние его внутренних объектов. Это хорошее применение описанных выше принципов. Впрочем, метод backlogItems() к тому же является открытым. Нарушает ли это принципы, которым мы пытаемся следовать? Следует ли открывать клиентам экземпляры класса ProductBacklogItem? Да, коллекция открыта, но клиенты могут использовать эти экземпляры только для запроса информации, содержащейся в них. Из-за ограниченности открытого интерфейса класса ProductBacklogItem клиенты не могут определить форму класса Product для глубокой навигации. Клиентам даются *минимальные знания*. Для удовлетворения потребностей клиентов возвращаемые экземпляры коллекции можно создавать только для одной операции, причем они не должны раскрывать состояние класса Product. Клиент никогда не может применять к экземплярам ProductBacklogItem команды, подвергающие их состояние опасности.

```
public class ProductBacklogItem extends ConcurrencySafeEntity {
    ...
    protected void reorderFrom(BacklogItemId anId, int anOrdering) {
        if (this.backlogItemId().equals(anId)) {
            this.setOrdering(anOrdering);
        } else if (this.ordering() >= anOrdering) {
            this.setOrdering(this.ordering() + 1);
        }
    }
    ...
}
```

Единственная функция, изменяющая состояние, объявлена как скрытый, защищенный метод. Таким образом, клиенты не могут видеть эту команду или получить доступ к ней. Для всех практических целей достаточно того, что эту команду видит и может выполнить только класс Product. Клиенты могут использовать только открытый командный метод reorderFrom() класса Product. Для того чтобы выполнить внутренние изменения, экземпляр класса Product делегирует вызов всем своим внутренним экземплярам класса ProductBacklogItem.

Реализация класса Product ограничивает знания о нем, упрощает его тестирование и облегчает поддержку, поскольку она соответствует простым принципам проектирования.

Следует сравнить между собой закон Деметры и принцип совмещения данных и поведения. Определенно, подход, основанный на законе Деметры, является намного более жестким. Он не допускает никакой навигации по частям АГРЕГАТА за пределами его КОРНЯ. С другой стороны, использование принципа совмещения данных и поведения допускает навигацию за пределами КОРНЯ, но ограничивает ее тем, что модификацию состояния АГРЕГАТА должен делать сам АГРЕГАТ, а не

клиент. Это объясняет, почему принцип совмещения данных и поведения более широко используется при реализации АГРЕГАТОВ.

Оптимистический параллелизм

Далее необходимо решить, где следует разместить атрибут оптимистического параллелелизма `version`. При изучении определения АГРЕГАТА может показаться, что самым безопасным было бы хранить версии только КОРНЕВОЙ СУЩНОСТИ. Версию корня можно было бы увеличивать каждый раз, когда *где-то внутри* АГРЕГАТА выполняется команда, изменяющая его состояние, независимо от глубины вложения этой команды. В приведенном выше примере класс `Product` мог бы иметь атрибут `version`, и когда выполняется один из его командных методов `describeAs()`, `initiateDiscussion()`, `rename()` или `reorderFrom()`, атрибут `version` можно было бы увеличить на единицу. Это не позволило бы всем остальным клиентам одновременно модифицировать любые атрибуты или свойства где бы то ни было внутри одного и того же экземпляра класса `Product`. В зависимости от выбранной схемы АГРЕГАТА этим процессом было бы сложно или даже невозможно управлять.

Предположим, что мы используем библиотеку Hibernate, и, когда в экземпляре класса `Product` изменяются атрибуты `name` или `description` или к нему добавляется атрибут `productDiscussion`, атрибут `version` автоматически увеличивается на единицу. Это само собой разумеется, потому что эти элементы хранятся непосредственно в КОРНЕВОЙ СУЩНОСТИ. Однако нельзя ли сделать так, чтобы атрибут `version` в классе `Product` увеличивался каждый раз, когда изменяется порядок следования одного из экземпляров `backlogItems`? Действительно, мы не можем это сделать, или по крайней мере не можем это сделать автоматически. Библиотека Hibernate не считает модификацию части экземпляра класса `ProductBacklogItem` модификацией самого экземпляра класса `Product`. Для решения этой проблемы, возможно, следовало бы просто изменить метод `reorderFrom()` в классе `Product`, включив в него флаг или просто увеличив на единицу атрибут `version`.

```
public class Product extends ConcurrencySafeEntity {
    ...
    public void reorderFrom(BacklogItemId anId, int anOrdering) {
        for (ProductBacklogItem pbi : this.backlogItems()) {
            pbi.reorderFrom(anId, anOrdering);
        }
        this.version(this.version() + 1);
    }
    ...
}
```

Проблема заключается в том, что этот код всегда будет портить класс Product, даже если команда переупорядочения на самом деле не имеет эффекта. Более того, этот код допускает утечку информации об инфраструктуре в модель, которой желательно избегать. Что же еще можно сделать?

Ковбойская логика

AJ: Я думаю, что женитьба — это разновидность оптимистического параллелизма. Когда мужчина женится, он оптимистично думает, что девушка никогда не изменится. В то же самое время она оптимистично думает то же самое о нем.

На самом деле экземпляры классов Product и ProductBacklogItem допускают возможность отказа от модификации версии КОРНЯ при изменении объектов backlogItems. Поскольку экземпляры в коллекции сами являются СУЩНОСТЯМИ, они могут иметь свои атрибуты оптимистического параллелизма version. Если два клиента переупорядочивают одни и те же экземпляры класса ProductBacklogItem, фиксация изменений последним клиентом закончится неудачей. По общему признанию перекрывающееся переупорядочение списков заданий происходит редко, потому что обычно этим занимается только владелец продукта.

Отслеживание версий всех частей СУЩНОСТИ возможно не всегда. Иногда единственным способом защитить инвариант является модификация версии КОРНЯ. Это можно сделать проще, если модифицировать унаследованное свойство КОРНЯ. В данном случае свойство КОРНЯ можно было бы всегда модифицировать в ответ на более глубокую модификацию одной из частей, которая, в свою очередь, заставляет библиотеку Hibernate увеличивать атрибут КОРНЯ version на единицу. Напомним, что этот подход был ранее описан при моделировании изменений состояния экземпляра класса BacklogItem, когда во всех его экземплярах класса Task оставшееся время выполнения устанавливается равным нулю.

Однако этот подход возможен не во всех случаях. Если он невозможен, может возникнуть желание прибегнуть к использованию средств механизма постоянного хранения для изменения КОРНЯ вручную, когда библиотека Hibernate указывает на части, которые необходимо модифицировать. Это становится проблематичным, поскольку для этого необходимо поддерживать двусторонние связи между дочерними частями и родительским КОРНЕМ. Двусторонние связи допускают переход от дочернего объекта назад к КОРНЮ, когда библиотека Hibernate посылает специализированному подписчику событие жизненного цикла. Впрочем, не стоит забывать, что, как указано в книге [Эванс], в большинстве случаев двусторонние

связи являются совершенно нежелательными, особенно если их приходится поддерживать для обеспечения оптимистического параллелизма, который относится к аспектам инфраструктуры.

Поскольку мы не хотим, чтобы аспекты инфраструктуры влияли на решения, касающиеся модели, следует найти менее болезненный путь. Если модификация корня становится слишком трудной и затратной, значит, необходимо разделить АГРЕГАТЫ на собственно КОРНЕВУЮ СУЩНОСТЬ, содержащую только простые атрибуты, и свойства с типами ЗНАЧЕНИЙ. Если АГРЕГАТЫ состоят только из КОРНЕВОЙ СУЩНОСТИ, то при модификации частей всегда будет происходить модификация КОРНЯ.

В заключение следует признать, что предыдущие сценарии не являются проблемой, если весь АГРЕГАТ хранится как одно значение, которое предотвращает конфликты, связанные с параллельной работой. Этот подход можно подкрепить использованием систем MongoDB, Riak, распределенных сетей Coherence компании Oracle и GemFire компании VMware. Например, если КОРЕНЬ АГРЕГАТА реализует интерфейс `Versionable` на основе сети Coherence, а его ХРАНИЛИЩЕ использует процессор входа `VersionedPut`, то КОРЕНЬ всегда будет отдельным объектом, который используется для выявления конфликтов, связанных с параллельной работой. Аналогичные результаты можно получить с помощью хранилищ типа "ключ–значение".

Как избежать внедрения зависимости

Внедрение в АГРЕГАТ зависимости от ХРАНИЛИЩА или СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ обычно выглядит опасным. Обоснованием для этого может быть поиск экземпляра зависимого объекта по запросу из АГРЕГАТА. Зависимым объектом может быть другой АГРЕГАТ или произвольное их количество. Как указывалось ранее в разделе "Правило: ссылайтесь на другие агрегаты по идентификаторам", желательно выполнять поиск зависимого объекта до вызова командного метода АГРЕГАТА и передавать ему этот объект. Менее желательно использовать ОТКЛЮЧЕННУЮ МОДЕЛЬ ПРЕДМЕТНОЙ ОБЛАСТИ.

Кроме того, в предметной области, для которой характерны плотный трафик, большой объем передаваемой информации, высокая производительность, дорогая память и циклы сбора мусора, следует подумать о потенциальных накладных расходах, связанных с внедрением ХРАНИЛИЩ АГРЕГАТОВ и экземпляров СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Сколько потребуется дополнительных ссылок на объекты? Можно возразить, что для описываемой предметной области высокие затраты на эксплуатацию не характерны. Тем не менее следует избегать ненужных накладных расходов, если их легко избежать, изменив принципы проектирования,

например выполняя поиск зависимостей до вызова командного метода агрегата и передавая ему найденный объект.

Мы лишь хотим предостеречь от внедрения ХРАНИЛИЩ и СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ в экземпляры АГРЕГАТОВ. Разумеется, в других ситуациях внедрение зависимостей является довольно удобным. Например, было бы целесообразно внедрить ссылки на ХРАНИЛИЩА и СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ в ПРИКЛАДНЫЕ СЛУЖБЫ.

Резюме

Мы увидели, насколько важным является следование ГЛАВНЫМ ПРАВИЛАМ РАБОТЫ С АГРЕГАТАМИ при проектировании агрегатов.

- Вы узнали о негативных последствиях моделирования крупнокластерных агрегатов.
- Вы научились моделировать истинные инварианты в границах согласованности.
- Вы рассмотрели преимущества проектирования небольших АГРЕГАТОВ.
- Вы теперь знаете, почему следует отдавать предпочтения ссылкам на другие АГРЕГАТЫ по идентификатору.
- Вы осознали важность использования принципа итоговой согласованности за пределами границы АГРЕГАТА.
- Вы узнали о разных методах реализации, включая принцип совмещения данных и поведения (приказывай, а не спрашивай) и закон Деметры.

Если вы будете придерживаться этих правил, то обеспечите необходимую согласованность, оптимальную производительность и высокую масштабируемость систем, воплощая ЕДИНЫЙ ЯЗЫК предметной области в тщательно проработанную модель.

Глава 11

Фабрики

Я не выношу уродства на фабриках! Пожалуйста, заходите!
Но будьте осторожны, дорогие дети! Не теряйте головы!
Не волнуйтесь! Сохраняйте спокойствие!

Вилли Вонка

Среди всех шаблонов DDD **ФАБРИКА (FACTORY)**, вероятно, — один из самых известных. Самыми широко разрекламированными являются описанные в книге *Design Patterns* [Гамма и др.] шаблоны **АБСТРАКТНАЯ ФАБРИКА (ABSTRACT FACTORY)**, **ФАБРИЧНЫЙ МЕТОД (FACTORY METHOD)** и **СТРОИТЕЛЬ (BUILDER)**. Я вовсе не собираюсь отодвигать на задний план советы, данные в книгах [Гамма и др.] и [Эванс]. Вместо этого мы сосредоточимся на примерах использования ФАБРИК в моделях предметной области.

Назначение главы

- Узнать, почему использование ФАБРИК помогает создавать выразительные модели, придерживающиеся **ЕДИНОГО ЯЗЫКА (1)**.
- Увидеть, как компания SaasOvation использует ФАБРИЧНЫЕ МЕТОДЫ в качестве функций **АГРЕГАТА (10)**.
- Рассмотреть, как с помощью ФАБРИЧНЫХ МЕТОДОВ можно создавать экземпляры АГРЕГАТОВ других типов.
- Научиться проектировать СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ как ФАБРИКИ, взаимодействующие с другими **ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (2)** и транслирующие внешние объекты в локальные типы.

ФАБРИКИ в модели предметной области

Рассмотрим основные мотивы использования ФАБРИК.

Передайте обязанности по созданию экземпляров сложных объектов и АГРЕГАТОВ отдельному объекту, который сам по себе может не выполнять

никаких функций в модели предметной области, но, тем не менее, является элементом ее архитектуры. Обеспечьте интерфейс, который бы инкапсулировал все сложные операции сборки объекта и не требовал от клиента ссылаться на конкретные классы создаваемого объекта. Создавайте АГРЕГАТЫ как единое целое, контролируя выполнение инвариантов [Эванс, с. 135].

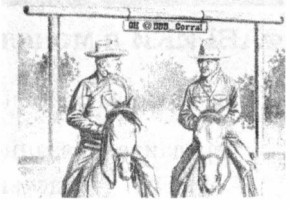
Помимо создания объектов в предметной области, ФАБРИКА может иметь дополнительные обязанности. Объект, единственная цель которого — создание экземпляра конкретного типа АГРЕГАТА, не имеет дополнительных обязанностей и не должен рассматриваться как основной элемент модели. Он просто является ФАБРИКОЙ. Основной обязанностью КОРНЯ АГРЕГАТА, предоставляющего ФАБРИЧНЫЙ МЕТОД для создания экземпляров другого типа АГРЕГАТА (или его внутренних частей), является реализация основной функции АГРЕГАТА, а ФАБРИЧНЫЙ МЕТОД — лишь один из его методов.

В своих примерах я чаще всего рассматриваю именно вторую ситуацию. В основном я рассматриваю несложные процессы конструирования АГРЕГАТОВ. Впрочем, некоторые важные детали процесса создания АГРЕГАТОВ должны быть защищены от неправильного состояния. Рассмотрим требования многоарендной среды. Если экземпляр АГРЕГАТА был создан некорректным арендатором, указавшим неправильный идентификатор `TenantId`, результат может быть катастрофическим. Очень важно сохранять все данные каждого арендатора отдельно и обеспечить их защиту от всех остальных. Размещая тщательно проработанный фабричный МЕТОД в конкретных КОРНЯХ АГРЕГАТОВ, можно гарантировать, что арендатор и другие связанные с ним компоненты будут созданы правильно. Это позволяет упростить работу клиентов, требуя лишь, чтобы они передавали только основные параметры, часто только **ОБЪЕКТЫ-ЗНАЧЕНИЯ (6)**, скрывая от них детали конструкции.

Кроме того, ФАБРИЧНЫЕ МЕТОДЫ АГРЕГАТОВ позволяют выражать ЕДИНЫЙ ЯЗЫК таким способом, который невозможно обеспечить с помощью одних лишь конструкторов. Когда имя поведенческого метода является выразительным с точки зрения ЕДИНОГО ЯЗЫКА, использование ФАБРИЧНОГО МЕТОДА приобретает дополнительную мощь.

Ковбойская логика

LB: Я раньше работал на фабрике пожарных насосов. Вы никогда не смогли бы припарковаться поблизости от этого места.



В некоторых случаях ОГРАНИЧЕННЫЕ КОНТЕКСТЫ имеют сложную конструкцию. Эти ситуации возникают, например, при **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**. В этих ситуациях **СЛУЖБЫ (7)** функционируют как ФАБРИКИ, создавая АГРЕГАТЫ или ОБЪЕКТЫ-ЗНАЧЕНИЯ разных типов.

Классическим примером, демонстрирующим преимущества АБСТРАКТНОЙ ФАБРИКИ, является создание объектов различных типов в иерархии классов. Клиент обязан передать только несколько основных параметров, по которым ФАБРИКА сможет определить конкретный тип, который должен быть создан. Среди моих примеров нет проблемно-ориентированных иерархий классов, поэтому я не буду демонстрировать эту ситуацию. Если вы видите иерархии классов в своих будущих моделях предметной области, я предлагаю обсудить их одновременно с **ХРАНИЛИЩАМИ (12)**. Это поможет вам принимать осознанные решения. Если вы решите использовать иерархии классов в своем проекте, будьте готовы к потенциальным неприятностям.

ФАБРИЧНЫЙ МЕТОД в КОРНЕ АГРЕГАТА

В трех примерах ОГРАНИЧЕННЫХ КОНТЕКСТОВ существуют три места сосредоточения ФАБРИК СУЩНОСТЕЙ КОРНЯ АГРЕГАТА. Их описание приведено в табл. 11.1.

Таблица 11.1. Места сосредоточения ФАБРИЧНЫХ методов агрегата

Ограниченный контекст	Фабрика агрегата	Метод
Контекст идентификации и доступа	`Tenant`	`offerRegistrationInvitation()`
		`provisionGroup()`
		`provisionRole()`
		`registerUser()`
Контекст сотрудничества	`Calendar`	`scheduleCalendarEntry()`
	`Forum`	`startDiscussion()`
	`Discussion`	`post()`
Контекст управления гибким проектированием	`Product`	`planBacklogItem()`
		`scheduleRelease()`
		`scheduleSprint()`

ФАБРИЧНЫЕ МЕТОДЫ класса `Product` обсуждаются в главе, посвященной **АГРЕГАТАМ (10)**. Например, его метод `planBacklogItem()` создает новый экземпляр класса `BacklogItem`, являющийся АГРЕГАТОМ, который впоследствии возвращается клиенту.

Для демонстрации проекта ФАБРИЧНЫХ МЕТОДОВ рассмотрим три метода в *Контексте сотрудничества*.

Создание экземпляров класса `CalendarEntry`

Рассмотрим проект. ФАБРИКА, которую мы в данный момент рассматриваем, расположена в классе `Calendar` и используется для создания экземпляров класса `CalendarEntry`. Команда CollabOvation приступает к реализации.

Рассмотрим тест, разработанный для демонстрации использования ФАБРИЧНЫХ МЕТОДОВ класса `Calendar`.

```
public class CalendarTest extends DomainTest {
    private CalendarEntry calendarEntry;
    private CalendarEntryId calendarEntryId;
    ...
    public void testCreateCalendarEntry() throws Exception {

       Calendar calendar = this.calendarFixture();

       DomainRegistry.calendarRepository().add(calendar);

       DomainEventPublisher
            .instance()
            .subscribe(
                 new DomainEventSubscriber<CalendarEntryScheduled>() {
                 public void handleEvent(
                        CalendarEntryScheduled aDomainEvent) {
                    calendarEntryId = aDomainEvent.calendarEntryId();
                 }
                 public Class<CalendarEntryScheduled>
                        subscribedToEventType() {
                    return CalendarEntryScheduled.class;
                 }
       });
```

```
 calendarEntry =
     calendar.scheduleCalendarEntry(
              DomainRegistry
                    .calendarEntryRepository()
                    .nextIdentity()
              new Owner(
                  "jdoe",
                  "John Doe",
                  "jdoe@lastnamedoe.org"),
                  "Sprint Planning",
              "Plan sprint for first half of April 2012.",
              this.tomorrowOneHourTimeSpanFixture(),
              this.oneHourBeforeAlarmFixture(),
              this.weeklyRepetitionFixture(),
              "Team Room",
              new TreeSet<Invitee>(0));

       DomainRegistry.calendarEntryRepository().add(calendarEntry);

       assertNotNull(calendarEntryId);
       assertNotNull(calendarEntry);
       ...
    }
}
```

Методу `scheduleCalendarEntry()` передается девять параметров. Как будет показано позднее, конструктор класса `CalendarEntry` требует 11 параметров. В свое время мы рассмотрим преимущества такого подхода. После успешного создания нового экземпляра класса `CalendarEntry` клиент должен добавить его в свое ХРАНИЛИЩЕ. В случае неудачи новый экземпляр удаляется механизмом сбора мусора.

Первое утверждение демонстрирует, что идентификатор `CalendarEntryId`, публикуемый вместе с СОБЫТИЕМ, должен быть не нулевым, что свидетельствует об успешной публикации СОБЫТИЯ. Это не значит, что непосредственный клиент класса `Calendar` действительно подписывается на это СОБЫТИЕ, но тест демонстрирует, что СОБЫТИЕ `CalendarEntryScheduled` действительно опубликовано.

Новый экземпляр класса `CalendarEntry` также должен быть ненулевым. Можно было бы сделать дополнительные утверждения, но два показанных утверждения являются самыми важными для документирования проекта ФАБРИЧНОГО МЕТОДА и способа его использования клиентами.

Перейдем к реализации ФАБРИЧНОГО МЕТОДА.

```
package com.saasovation.collaboration.domain.model.calendar;

public class Calendar extends Entity {
    ...
    public CalendarEntry scheduleCalendarEntry(
           CalendarEntryId aCalendarEntryId,
           Owner anOwner,
           String aSubject,
```

```
            String aDescription,
            TimeSpan aTimeSpan,
            Alarm anAlarm,
            Repetition aRepetition,
            String aLocation,
            Set<Invitee> anInvitees) {
            CalendarEntry calendarEntry =
               new CalendarEntry(
                        this.tenant(),
                        this.calendarId(),
                        aCalendarEntryId,
                        anOwner,
                        aSubject,
                        aDescription,
                        aTimeSpan,
                        anAlarm,
                        aRepetition,
                        aLocation,
                        anInvitees);

            DomainEventPublisher
                .instance()
                .publish(new CalendarEntryScheduled(...));

            return calendarEntry;
    }
    ...
}
```

Класс Calendar создает новый экземпляр АГРЕГАТА класса CalendarEntry. Этот новый экземпляр возвращается клиенту вместе с публикацией СОБЫТИЯ CalendarEntryScheduled. (Детали опубликованного СОБЫТИЯ для нашей дискуссии значения не имеют.) В верхней части метода вы могли заметить недостаток предохранителей. Защищать ФАБРИЧНЫЙ МЕТОД необязательно, потому что конструкторы каждого из параметров, являющихся ЗНАЧЕНИЯМИ, конструктор класса CalendarEntry и методы-установщики, к которым этот конструктор выполняет самоделегирование, содержат все необходимые предохранители. (Самоделегирование и защита описаны в главе, посвященной **СУЩНОСТЯМ (5)**.) Если бы вы хотели проявить двойную осторожность, то вставили бы эти предохранители и в ФАБРИЧНЫЙ МЕТОД.

Команда, разрабатывавшая ФАБРИЧНЫЙ МЕТОД, придерживалась ЕДИНОГО ЯЗЫКА.

Эксперты в предметной области, а также остальная часть команды, обсудили следующий сценарий.

Календари планируют календарные записи

Если бы наш проект предусматривал только открытый конструктор класса `CalendarEntry`, это снизило бы выразительность модели и мы не смогли бы выразить ее в терминах ЯЗЫКА предметной области. Наш проект требует, чтобы конструктор АГРЕГАТА был полностью скрыт от клиентов. Мы объявляем защищенный конструктор, чтобы клиенты применяли к объекту класса `Calendar` ФАБРИЧНЫЙ МЕТОД `scheduleCalendarEntry()`.

```
public class CalendarEntry extends Entity {
    ...
    protected CalendarEntry(
        Tenant aTenant, CalendarId aCalendarId,
        CalendarEntryId aCalendarEntryId, Owner anOwner,
        String aSubject, String aDescription, TimeSpan aTimeSpan,
        Alarm anAlarm, Repetition aRepetition, String aLocation,
        Set<Invitee> anInvitees) {
        ...

    }
    ...
}
```

Несмотря на такие преимущества, как продуманная конструкция, сниженная нагрузка на клиентов и выразительная модель, ФАБРИЧНЫЙ МЕТОД класса `Calendar` имеет один недостаток — он снижает производительность. Как и при использовании любого другого ФАБРИЧНОГО МЕТОДА АГРЕГАТА, экземпляр класса `Calendar` должен запрашиваться из хранилища, прежде чем можно будет создать экземпляр класса `CalendarEntry`. Эта дополнительная работа может быть вполне обоснованной, но при возрастании трафика в ОГРАНИЧЕННОМ КОНТЕКСТЕ команда должна тщательно взвесить последствия.

К преимуществам ФАБРИК относится и то, что два параметра конструктора класса `CalendarEntry` клиентами не передаются. С учетом того, что конструктор имеет 11 параметров, такая схема снимает с клиентов нагрузку, требуя от них передать только девять параметров. Большинство из этих параметров клиенты могут создать без каких-либо затруднений. (По общему признанию, использование контейнера `Set`, содержащего экземпляры класса `Invitee`, не является недостатком ФАБРИЧНОГО МЕТОДА. Одновременно с разработкой ФАБРИКИ команда должна обдумать более удобный способ передачи экземпляра класса `Set`.)

Экземпляр класса `Tenant` и связанный с ним идентификатор `CalendarId` предоставляются только ФАБРИЧНЫМ МЕТОДОМ. Это гарантирует, что экземпляры класса

CalendarEntry будут создаваться только для корректных экземпляров класса Tenant в ассоциации с корректным экземпляром класса Calendar.

Теперь рассмотрим еще один пример из *Контекста сотрудничества*.

Создание экземпляров класса Discussion

Взгляните на ФАБРИЧНЫЙ МЕТОД класса Forum. Он имеет ту же мотивацию и очень похожую реализацию, что и ФАБРИЧНЫЙ МЕТОД класса Calendar, поэтому мы не будем вдаваться в его детали. Впрочем, использование ФАБРИЧНОГО МЕТОДА в этом классе имеет дополнительное преимущество.

Рассмотрим ФАБРИЧНЫЙ МЕТОД startDiscussion() класса Forum, соответствующий ЯЗЫКУ.

```
package com.saasovation.collaboration.domain.model.forum;

public class Forum extends Entity {
    ...
    public Discussion startDiscussion(
            DiscussionId aDiscussionId,
            Author anAuthor,
            String aSubject) {
        if (this.isClosed()) {
            throw new IllegalStateException("Forum is closed.");
        }

        Discussion discussion = new Discussion(
                this.tenant(),
                this.forumId(),
                aDiscussionId,
                anAuthor,
                aSubject);

        DomainEventPublisher
            .instance()
            .publish(new DiscussionStarted(...));

        return discussion;
    }
    ...
}
```

Помимо создания экземпляра класса Discussion, этот ФАБРИЧНЫЙ МЕТОД защищает его, если форум закрыт. Класс Forum содержит экземпляр класса Tenant и связанный с ним идентификатор ForumId. Таким образом, клиент должен передать только три из пяти параметров, требующихся для создания экземпляра класса Discussion.

Кроме того, этот ФАБРИЧНЫЙ МЕТОД выражает ЕДИНЫЙ ЯЗЫК *Контекста сотрудничества*. Команда использовала метод `startDiscussion()` класса `Forum` в полном соответствии с выражением экспертов в предметной области:

Дискуссии в форумах начинают авторы.

Это позволяет еще больше упростить клиент.

```
Discussion discussion = agilePmForum.startDiscussion(
    this.discussionRepository.nextIdentity(),
    new Author("jdoe", "John Doe", "jdoe@saasovation.com"),
    "Dealing with Aggregate Concurrency Issues");

    assertNotNull(discussion);
    ...
    this.discussionRepository.add(discussion);
```

Это действительно просто. Именно к простоте всегда стремятся специалисты, моделирующие предметную область.

Шаблон ФАБРИЧНЫЙ МЕТОД можно повторять столько раз, сколько потребуется. Я считаю, что мы достаточно полно продемонстрировали, насколько эффективно ФАБРИЧНЫЕ МЕТОДЫ АГРЕГАТОВ могут выражать ЯЗЫК в КОНТЕКСТЕ, снижать нагрузку на клиентов при создании новых экземпляров АГРЕГАТА и гарантировать правильное создание экземпляров.

ФАБРИКА СЛУЖБ

Поскольку я часто использую СЛУЖБЫ как ФАБРИКИ при **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**, *б*ольшую часть вопросов я перенес именно в эту главу. Здесь же мы рассмотрим вопросы интеграции с **ПРЕДОХРАНИТЕЛЬНЫМ УРОВНЕМ (3)**, **ОБЩЕДОСТУПНЫМ ЯЗЫКОМ (3)** и **СЛУЖБОЙ С ОТКРЫТЫМ ПРОТОКОЛОМ (3)**. Здесь же я хотел бы рассмотреть саму ФАБРИКУ и способы создания СЛУЖБЫ на основе ФАБРИКИ.

Рассмотрим еще один пример из *Контекста сотрудничества*. Это ФАБРИКА в виде интерфейса `CollaboratorService`, создающего экземпляры класса `Collaborator` по идентификаторам арендатора и пользователя.

```
package com.saasovation.collaboration.domain.model.collaborator;

import com.saasovation.collaboration.domain.model.tenant.Tenant;

public interface CollaboratorService {

    public Author authorFrom(Tenant aTenant, String anIdentity);

    public Creator creatorFrom(Tenant aTenant, String anIdentity);

    public Moderator moderatorFrom(Tenant aTenant, String anIdentity);

    public Owner ownerFrom(Tenant aTenant, String anIdentity);

   public Participant participantFrom(
       Tenant aTenant,
       String anIdentity);
}
```

Эта СЛУЖБА обеспечивает трансляцию ОБЪЕКТОВ из *Контекста идентификации и доступа* в *Контекст сотрудничества*. Как показано в главе, посвященной **ОГРАНИЧЕННЫМ КОНТЕКСТАМ** (2), команда CollabOvation не общается с пользователями при обсуждении сотрудничества. Главное то, что люди в предметной области сотрудничества являются авторами, создателями, модераторами, владельцами и участниками. Для того чтобы выполнить это условие, команде необходимо взаимодействовать с *Контекстом идентификации и доступа* через СЛУЖБУ и трансформировать объекты пользователя и роли из модели в соответствующие объекты сотрудничества в собственной модели КОНТЕКСТА.

Поскольку новые объекты, которые выводятся из абстрактного базового класса `Collaborator`, создаются СЛУЖБОЙ, то, по существу, она функционирует как ФАБРИКА. Анализ одного из методов интерфейса раскрывает некоторые детали.

```
package com.saasovation.collaboration.infrastructure.services;

public class UserRoleToCollaboratorService
       implements CollaboratorService {

    public UserRoleToCollaboratorService() {
       super();
    }

    @Override
    public Author authorFrom(Tenant aTenant, String anIdentity) {
       return
          (Author)
              UserInRoleAdapter
                   .newInstance()
                   .toCollaborator(
                         aTenant,
                         anIdentity,
```

```
                        "Author",
                        Author.class);
    }
    ...
}
```

Поскольку эта реализация носит технический характер, класс включается в **МОДУЛЬ (9)** на ИНФРАСТРУКТУРНОМ УРОВНЕ.

В этой реализации экземпляр класса `UserInRoleAdapter` превращает экземпляр класса `Tenant` и идентификатор — имя пользователя — в экземпляр класса `Author`. Этот шаблон **АДАПТЕР (ADAPTER)** [Гамма и др.] взаимодействует со СЛУЖБОЙ С ОТКРЫТЫМ ПРОТОКОЛОМ *Контекста идентификации и доступа* для подтверждения того, что данный пользователь играет роль АВТОРА. Если это правда, то АДАПТЕР выполняет делегирование к классу `CollaboratorTranslator`, чтобы тот транслировал ответ на ОБЩЕПРИНЯТОМ ЯЗЫКЕ в экземпляр класса `Author` в локальной модели. Класс `Author`, как и другие подклассы класса `Collaborator`, является простым ОБЪЕКТОМ-ЗНАЧЕНИЕМ.

```
package com.saasovation.collaboration.domain.model.collaborator;

public class Author extends Collaborator {
    ...
}
```

Кроме конструкторов и методов `equals()`, `hashCode()` и `toString()`, каждый из подклассов получает все состояние и поведенческие функции от класса `Collaborator`.

```
package com.saasovation.collaboration.domain.model.collaborator;

public abstract class Collaborator implements Serializable {
    private String emailAddress;
    private String identity;
    private String name;

    public Collaborator(
           String anIdentity,
           String aName,
           String anEmailAddress) {
       super();
       this.setEmailAddress(anEmailAddress);
       this.setIdentity(anIdentity);
       this.setName(aName);
    }
    ...
}
```

Контекст сотрудничества использует объект username как атрибут Collaboratoridentity. Объекты emailAddress и name являются экземплярами класса String. В этой модели команда решила придерживаться как можно более простых концепций. Имя пользователя, например, хранится как полное имя в текстовом виде. С помощью СЕРВИС-ОРИЕНТИРОВАННОЙ ФАБРИКИ можно было бы разделить жизненные циклы и терминологию двух рассматриваемых ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Существует показатель сложности классов UserInRoleAdapter и CollaboratorTranslator. Коротко говоря, класс UserInRoleAdapter отвечает только за взаимодействие с внешним контекстом, а класс CollaboratorTranslator — только за трансляцию, которая выражается в создании объектов. Подробности описаны в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**.

Резюме

Мы исследовали причины, по которым ФАБРИКИ используются в предметно-ориентированном проектировании, и показали, как они согласуются с моделями.

- Теперь вы знаете, почему ФАБРИКИ помогают создавать выразительные модели, придерживающиеся ЕДИНОГО ЯЗЫКА.
- Вы увидели два разных ФАБРИЧНЫХ МЕТОДА, реализованных как поведенческие функции АГРЕГАТА.
- Вы узнали, как с помощью ФАБРИЧНЫХ МЕТОДОВ можно создавать экземпляры АГРЕГАТОВ других типов, гарантируя правильность их создания и используя уязвимые данные.
- Вы научились проектировать СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ как ФАБРИКИ, взаимодействующие с другими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ и транслирующие внешние объекты в локальные типы.

Теперь мы рассмотрим два основных стиля проектирования ХРАНИЛИЩ, а также другие вопросы их реализации.

Глава 12

Хранилища

Твои глаза такого же цвета,
как мое запоминающее устройство.
Подслушано среди работяг в баре

Хранилищем (repository) обычно называется область памяти, предназначенная для безопасного хранения помещенных в нее элементов. Когда вы сохраняете нечто в хранилище, а позднее возвращаетесь, чтобы извлечь это нечто из него, вы ожидаете, что оно будет в том же состоянии, в котором было, когда вы его поместили в хранилище. В какой-то момент вы можете принять решение удалить сохраненный элемент из хранилища.

Этот основной набор принципов относится и к предметно-ориентированному ХРАНИЛИЩУ. Помещая экземпляр **АГРЕГАТА (10)** в соответствующее ХРАНИЛИЩЕ, а затем извлекая его оттуда, вы получаете ожидаемый целостный объект. Если вы измените экземпляр АГРЕГАТА, извлеченный из ХРАНИЛИЩА, то эти изменения будут сохранены. Если вы удалите экземпляр из ХРАНИЛИЩА, то больше не сможете его извлечь.

> Для каждого типа объектов, к которым требуется глобальный доступ, введите объект-посредник, который может создать иллюзию, что все объекты такого типа объединены в коллекцию и находятся в оперативной памяти. Наладьте доступ через хорошо известный глобальный интерфейс... Реализуйте методы, которые будут выбирать объекты по заданным критериям и возвращать полностью сгенерированные и инициализированные объекты или коллекции объектов с атрибутами, подходящими под критерии... Реализуйте ХРАНИЛИЩА только для АГРЕГАТОВ... [Эванс, с. 145.]

Все эти объекты, напоминающие коллекции, требуют постоянного хранения. Каждый тип АГРЕГАТА, предполагающий постоянное хранение, имеет свое ХРАНИЛИЩЕ. В принципе, между типом АГРЕГАТА и ХРАНИЛИЩЕМ существует отношение "один-к-одному". Однако иногда, когда несколько типов АГРЕГАТА совместно используют одну и ту же иерархию объектов, эти типы могут иметь общее хранилище. В главе обсуждаются оба подхода.

Назначение главы

- Изучить ХРАНИЛИЩА, ориентированные на коллекции и на механизмы постоянного хранения, и понять, когда и почему используется тот или иной вид ХРАНИЛИЩ.
- Продемонстрировать реализацию ХРАНИЛИЩ в системах Hibernate, TopLink, Coherence и MongoDB.
- Показать, почему в интерфейсе ХРАНИЛИЩА может понадобиться дополнительная поведенческая функция.
- Продемонстрировать роль транзакций в использовании ХРАНИЛИЩ.
- Показать проблемы, связанные с проектированием ХРАНИЛИЩ для иерархий типов.
- Показать фундаментальные различия между ХРАНИЛИЩАМИ и ОБЪЕКТАМИ ДОСТУПА К ДАННЫМ (DATA ACCESS OBJECTS).
- Продемонстрировать тестирование ХРАНИЛИЩ и разные способы проверки их эксплуатации.

Строго говоря, только АГРЕГАТЫ имеют ХРАНИЛИЩА. Если вы не используете АГРЕГАТЫ в конкретном **ОГРАНИЧЕННОМКОНТЕКСТЕ (2)**, то шаблон ХРАНИЛИЩЕ может не принести большой пользы. Если вы извлекаете и используете **СУЩНОСТИ (5)** непосредственно и произвольным образом, не прорабатывая транзакционные границы АГРЕГАТОВ, то, возможно, вам следует избегать ХРАНИЛИЩ. Однако те, кого больше интересует техническая сторона дела, а не принципы DDD, иногда выбирают ХРАНИЛИЩА, а не ОБЪЕКТЫ ДОСТУПА К ДАННЫМ. В то же время остальные полагают, что при работе с механизмом постоянного хранения целесообразнее непосредственно использовать шаблон СЕАНС (SESSION) или ЕДИНИЦА РАБОТЫ (UNIT OF WORK) [P of EAA]. Это не значит, что вы обязаны избегать использования АГРЕГАТОВ. Как раз наоборот. Впрочем, это лишь возможность, которой кто-то воспользуется, а кто-то нет.

По моим оценкам, есть два вида проектов ХРАНИЛИЩ — *ориентированные на имитацию коллекции* (collection-oriented) и *ориентированные на механизм постоянного хранения* (persistence-oriented). Иногда полезно ориентироваться на имитацию коллекции, а иногда — на механизм постоянного хранения. Сначала мы обсудим, когда использовать и как создать ХРАНИЛИЩЕ, ориентированное на имитацию коллекции, а затем перейдем к изучению ХРАНИЛИЩ, ориентированных на механизм постоянного хранения.

Хранилища, ориентированные на имитацию коллекции

Проекты, ориентированные на имитацию коллекции, можно назвать традиционными, потому что они основаны на идеях, представленных в исходном шаблоне DDD. Они очень точно имитируют коллекцию, моделируя по крайней мере часть ее стандартного интерфейса. Интерфейс ХРАНИЛИЩА, разработанный в рамках этого шаблона, ничем не выдает существования базового механизма постоянного хранения, избегая любого понятия, связанного с сохранением или постоянным хранением данных в памяти.

Поскольку этот подход к проектированию требует наличия определенных возможностей базового механизма постоянного хранения, он может оказаться неудобным. Если ваш механизм постоянного хранения препятствует проектированию, ориентированному на имитацию коллекции, переходите к следующему подразделу. В этом подразделе описаны ситуации, в которых проекты, ориентированные на имитацию коллекций, работают лучше всего. Для этого нужно заложить определенный фундамент.

Посмотрите, как работает стандартная коллекция. В языках Java, C# и большинстве других объектно-ориентированных языков программирования объекты добавляются в коллекции и остаются там, пока не будут удалены. Нет никакой необходимости делать что-то специальное, чтобы заставить коллекцию распознавать изменения в объектах, которые она содержит. Достаточно запросить у коллекции ссылку на конкретный объект, а затем попросить, чтобы этот объект применил к себе действие, которое изменяет его собственное состояние. Этот объект все еще хранится в коллекции, но теперь его состояние отличается от состояния до модификации.

Рассмотрим эту процедуру подробнее на нескольких примерах. Возьмем стандартный интерфейс `java.util.Collection`.

```
package java.util;

public interface Collection ... {
    public boolean add(Object o);
    public boolean addAll(Collection c);
    public boolean remove(Object o);
    public boolean removeAll(Collection c);
    ...
}
```

Если вы хотите добавить объект в коллекцию, вызывайте функцию `add()`. Если вы хотите удалить этот объект, передайте его ссылку функции `remove()`. В следующем тесте предполагается, что только что созданная коллекция может содержать экземпляры класса `Calendar`.

```
assertTrue(calendarCollection.add(calendar));

assertEquals(1, calendarCollection.size());

assertTrue(calendarCollection.remove(calendar));

assertEquals(0, calendarCollection.size());
```

Все это довольно просто. ХРАНИЛИЩЕ имитирует специальный вид коллекций — `java.util.Set` и ее реализацию `java.util.HashSet`. Каждый объект, добавляемый в экземпляр класса `Set`, должен быть уникальным. Попытка добавить объект, который уже содержится в коллекции `Set`, завершится неудачей. Таким образом, необходимость добавления одного и того же объекта дважды никогда не возникает. Добавление объекта во второй раз означало бы, что вы пытаетесь сохранить изменения, которые объект произвел самостоятельно. Следующие тестовые утверждения доказывают, что добавление одного и того же объекта не приводит ни к каким последствиям, как позитивным, так и негативным.

```
Set<Calendar> calendarSet = new HashSet<Calendar>();

assertTrue(calendarSet.add(calendar));

assertEquals(1, calendarSet.size());

assertFalse(calendarSet.add(calendar));

assertEquals(1, calendarSet.size());
```

Все эти утверждения выполняются, потому что один и тот же экземпляр класса `Calendar` добавляется дважды и вторая попытка добавить объект не изменяет состояние экземпляра класса `Set`. Это относится к любому ХРАНИЛИЩУ, ориентированному на коллекции. Если добавить экземпляр АГРЕГАТА `calendar` в экземпляр класса `CalendarRepository`, имитирующий коллекцию, то добавление экземпляра `calendar` второй раз не вызовет негативных последствий. Каждый АГРЕГАТ имеет уникальный глобальный идентификатор, ассоциированный с **КОРНЕВОЙ СУЩНОСТЬЮ (ROOT ENTITY) (5, 10)**. Именно этот уникальный идентификатор позволяет ХРАНИЛИЩУ типа `Set` предотвращать добавление одних и тех же экземпляров АГРЕГАТА больше одного раза.

Важно понимать, какой именно вид коллекции — `Set` — должно имитировать ХРАНИЛИЩЕ. Независимо от конкретной реализации и конкретного механизма постоянного хранения нельзя разрешать добавление одного и того же объекта дважды.

Другой ключевой вывод — вы не должны "повторно сохранять" измененные объекты, уже сохраненные в ХРАНИЛИЩЕ. Подумайте еще раз о модификации

объекта, уже хранящегося в коллекции. Это действительно просто. Достаточно извлечь из коллекции ссылку на объект, который вы хотите модифицировать, а затем попросить объект выполнить поведенческую функцию, изменяющую его состояние, вызвав командный метод.

Выводы по ХРАНИЛИЩАМ, ориентированным на имитацию коллекции

ХРАНИЛИЩЕ должно имитировать коллекцию `Set`. Какой бы ни была базовая реализация конкретного механизма постоянного хранения, вы не должны допускать повторной вставки экземпляра одного и того же объекта. Кроме того, при извлечении объектов из ХРАНИЛИЩА и их модификации нет необходимости повторно сохранять их в ХРАНИЛИЩЕ.

Для иллюстрации создадим подкласс стандартного класса `java.util.HashSet` и создадим метод для нового типа, позволяющий находить конкретный экземпляр объекта по уникальному идентификатору. Дадим этому подклассу имя, которое будет идентифицировать его в ХРАНИЛИЩЕ, но на самом деле это обычный встроенный класс `HashSet`.

```
public class CalendarRepository extends HashSet {
    private Set<CalendarId, Calendar> calendars;

    public CalendarRepository() {
       this.calendars = new HashSet<CalendarId, Calendar>();
    }

    public void add(Calendar aCalendar) {
       this.calendars.add(aCalendar.calendarId(), aCalendar);
    }

    public Calendar findCalendar(CalendarId aCalendarId) {
       return this.calendars.get(aCalendarId);
    }
}
```

Обычно подкласс `HashSet` не используется для создания типичного ХРАНИЛИЩА. Здесь мы его рассматриваем просто как образец. Итак, вернемся к примеру. Теперь мы можем добавить экземпляр класса `Calendar` в специализированную коллекцию `Set`, а затем найти этот экземпляр и попросить его модифицировать себя.

```
CalendarId calendarId = new CalendarId(...);
Calendar calendar =
    new Calendar(calendarId, "Project Calendar", ...);
CalendarRepository calendarRepository = new CalendarRepository();
calendarRepository.add(calendar);
```

```
// позднее...

Calendar calendarToRename =
    calendarRepository.findCalendar(calendarId);

calendarToRename.rename("CollabOvation Project Calendar");

// еще позднее...

Calendar calendarThatWasRenamed =
   calendarRepository.findCalendar(calendarId);

assertEquals("CollabOvation Project Calendar",
   calendarThatWasRenamed.name());
```

Отметим, что экземпляр класса `Calendar`, на который ссылается переменная `calendarToRename`, сам изменяет свое имя по команде. Позднее, после переименования, это имя можно снова изменить. Для этого нет необходимости просить подкласс `CalendarRepository` класса `HashSet` сохранить изменения в экземпляре класса `Calendar`, потому что в этом нет смысла. В классе `CalendarRepository` нет метода `save()`, потому что в нем нет необходимости. Нет никаких причин сохранять изменения в экземпляре класса `Calendar`, на который ссылается переменная `calendarToRename`, потому что коллекция по-прежнему содержит ссылки на модифицируемый объект и все изменения происходят непосредственно в объекте.

Подведем итог. Традиционное ХРАНИЛИЩЕ, ориентированное на имитацию коллекции, действительно напоминает коллекцию тем, что через открытый интерфейс клиент не получает никакой информации о механизме постоянного хранения. Следовательно, наша цель — спроектировать и реализовать ХРАНИЛИЩЕ, ориентированное на имитацию коллекции, которое имело бы свойства класса `HashSet`, но при этом имело бы механизм постоянного хранения данных.

Для этого необходимо иметь определенные возможности базового механизма постоянного хранения. Этот механизм должен как-то поддерживать возможность неявно отслеживать изменения каждого из хранящихся объектов, которыми он управляет. Эту возможность можно реализовать разными способами. Укажем два из них.

1. **НЕЯВНОЕ КОПИРОВАНИЕ ПРИ ЧТЕНИИ (IMPLICIT COPY-ON-READ)** [Keith & Stafford]. Механизм постоянного хранения неявно копирует каждый объект хранения при чтении из базы данных и сравнивает его закрытую копию с клиентской при фиксации транзакции. Когда вы просите механизм постоянного хранения считать объект из базы данных, он делает это и немедленно создает копию всего объекта (за исключением частей, предназначенных для отложенной загрузки, которые могут быть загружены и

скопированы позднее). При фиксации транзакции, созданной с помощью механизма постоянного хранения, он проверяет наличие модификаций в загруженных (или заново подсоединенных) и скопированных объектах, сравнивая их. Все объекты с обнаруженными изменениями сбрасываются в базу данных.

2. **НЕЯВНОЕ КОПИРОВАНИЕ ПРИ ЗАПИСИ (IMPLICIT COPY-ON-WRITE)** [Keith & Stafford]. Механизм постоянного хранения управляет всеми загруженными объектами хранения с помощью прокси-объекта. При загрузке каждого объекта из базы данных создается тонкий прокси-объект, который управляется клиентом. Клиенты невольно вызывают поведенческую функцию из прокси-объекта, которая отражается в реальном объекте. Когда прокси-объект впервые получает вызов метода, он создает копию управляемого объекта. Прокси-объект отслеживает изменения состояния управляемого объекта и отмечает их. После фиксации транзакции, созданной механизмом постоянного хранения, она проверяет помеченные объекты и сбрасывает их в базу данных.

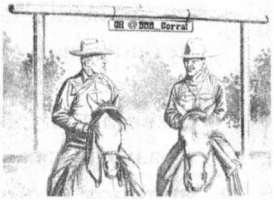
Преимущества этих подходов и различия между ними могут варьироваться. Если ваша система испытывает негативные последствия от применения одного из этих подходов, следует задуматься. Разумеется, вы можете просто выбрать свой привычный подход, стремясь избавиться от лишней работы, но это может быть не самым безопасным решением.

Общее преимущество обоих подходов заключается в том, что изменения объекта постоянного хранения отслеживаются неявно и от клиента не требуется специальных знаний или вмешательства в работу механизма постоянного хранения. Итак, использование механизма постоянного хранения, такого как Hibernate, *позволяет развернуть традиционное* ХРАНИЛИЩЕ, *ориентированное на имитацию коллекции.*

С другой стороны, это возможно, даже если вы считаете использование механизмов постоянного хранения, допускающих неявное отслеживание изменений путем копирования, таких как Hibernate, нежелательным или неприемлемым. Если вы работаете в очень высокопроизводительной среде и в каждый момент времени храните в памяти очень много объектов, такие механизмы могут значительно повысить нагрузку как на память, так и на систему выполнения. Следует тщательно взвесить все аргументы "за" и "против". Существует множество предметных областей, в которых работает механизм Hibernate, поэтому не считайте мои предостережения запретом. Использование любых инструментов сопряжено с компромиссами.

Ковбойская логика

LB: Когда у моей собаки заводятся глисты, ветеринар прописывает ей какие-то репозитории.[1]



Все вышесказанное может натолкнуть вас на мысль использовать более эффективный инструмент отображения объектно-реляционного отображения, способный поддерживать ХРАНИЛИЩА, ориентированные на имитацию коллекции. Одними из таких инструментов является система TopLink компании Oracle и ее ближайший родственник — EclipseLink. Система TopLink реализует шаблон ЕДИНИЦА РАБОТЫ, который в чем-то совпадает с шаблоном СЕАНС, реализуемым в механизме Hibernate. Однако шаблон ЕДИНИЦА РАБОТЫ в системе TopLink не подразумевает неявного копирования при чтении. Вместо этого система выполняет **ЯВНОЕ КОПИРОВАНИЕ ПЕРЕД ЗАПИСЬЮ (EXPLICIT COPY-BEFORE-WRITE)** [Keith & Stafford]. Здесь термин *явное* означает, что клиент должен информировать механизм ЕДИНИЦА РАБОТЫ о том, что произошли изменения. Это дает механизму ЕДИНИЦ РАБОТЫ возможность клонировать заданный объект предметной области и подготовить его к модификациям (этот процесс в дальнейшем будет называться *редактированием*). Ключевым моментом является то, что система TopLink потребляет память только при необходимости.

Реализация с помощью системы Hibernate

В любом из двух подходов создание ХРАНИЛИЩА происходит в два этапа. Сначала необходимо определить открытый интерфейс и по крайней мере одну реализацию.

В частности, в рамках первого подхода сначала необходимо определить интерфейс, имитирующий коллекцию. На втором этапе создается реализация, предназначенная для работы с базовым механизмом хранения, например Hibernate. Интерфейс, как и коллекция, часто содержит общие методы, которые можно обнаружить в следующем примере.

```
package com.saasovation.collaboration.domain.model.calendar;

public interface CalendarEntryRepository {
    public void add(CalendarEntry aCalendarEntry);
    public void addAll(
            Collection<CalendarEntry> aCalendarEntryCollection);
```

[1] Шутка основана на созвучии слов "репозиторий" (хранилище) и "суппозиторий" (ректальная свеча). — *Примеч. ред.*

```
    public void remove(CalendarEntry aCalendarEntry);
    public void removeAll(
           Collection<CalendarEntry> aCalendarEntryCollection);

}
```

Определение интерфейса следует поместить в тот же **МОДУЛЬ (9)**, в котором находится тип содержащего его АГРЕГАТА. В данном случае интерфейс `CalendarEntryRepository` находится в том же МОДУЛЕ (пакете Java), что и класс `CalendarEntry`. Как будет показано ниже, класс реализации размещается в отдельном пакете.

Интерфейс `CalendarEntryRepository` содержит методы, которые можно встретить в коллекциях, например в стандартной коллекции `java.util.Collection`. Новый экземпляр класса `CalendarEntry` можно добавить в данное ХРАНИЛИЩЕ с помощью метода `add()`. Несколько новых экземпляров можно добавить с помощью метода `addAll()`. После того как экземпляры будут добавлены, они будут постоянно храниться в базе данных и впоследствии извлекаться оттуда. Антиподами этих методов являются методы `remove()` и `removeAll()`, позволяющие удалять из коллекции один или сразу несколько экземпляров.

Лично я не люблю методы, возвращающие булевы значения, как принято в полноценных коллекциях. Это объясняется тем, что в некоторых ситуациях возврат значения `true` после выполнения операции добавления не гарантирует успеха. Значение `true` может относиться к фиксации транзакции в базе данных. Таким образом, в случае ХРАНИЛИЩА ключевое слово `void` может точнее отражать тип возвращаемого значения.

Бывают ситуации, в которых нельзя добавлять и/или удалять несколько экземпляров АГРЕГАТА в рамках одной транзакции. В таком случае не включайте в интерфейс методы `addAll()` и `removeAll()`. Впрочем, эти методы добавлены только для удобства. Клиент всегда может организовать цикл, в котором несколько раз вызываются методы `add()` или `remove()`, самостоятельно перемещаясь по коллекции. Таким образом, исключение методов `addAll()` и `removeAll()` является символическим жестом, не влияющим на проект, если только вы не хотели обнаружить ситуации, в которых добавление и удаление нескольких объектов выполняется в рамках одной транзакции. В таких ситуациях, скорее всего, придется создавать экземпляр ХРАНИЛИЩА для каждой транзакции, что потенциально может оказаться дорогостоящим решением. Я не хотел бы углубляться в дискуссию на эту тему.

Возможно, что экземпляры некоторых типов АГРЕГАТА в обычных условиях никогда не должны удаляться из ХРАНИЛИЩА. Это может быть вызвано желанием сохранить экземпляр надолго, даже если он больше не используется приложением, например для ссылок на него и/или для накопления исторических данных.

Кроме того, некоторые объекты трудно или даже невозможно удалить. С точки зрения базы данных удаление некоторых объектов может оказаться нежелательным, необоснованным или даже недопустимым. В некоторых случаях можно просто пометить экземпляр АГРЕГАТА как *отключенный*, *неиспользуемый* или даже *логически удаленный* (в зависимости от предметной области). В таких ситуациях можно не включать в открытый интерфейс ХРАНИЛИЩА никаких методов удаления или же реализовать методы удаления так, чтобы они делали экземпляры АГРЕГАТА недоступными. В качестве альтернативы можно предотвращать полное удаление объектов, предоставляя клиентам возможность внимательно просматривать код, чтобы убедиться в том, что он не содержит методов удаления. Решений может быть много, но проще всего отключить удаление вообще. Помимо всего прочего, любые методы открытого интерфейса по умолчанию считаются доступными. Если метод удаления открыт для доступа, но при этом экземпляр АГРЕГАТА может быть удален логически, значит, проектировщики предпочли реализовать логическое, а не физическое удаление.

Другой важной частью интерфейса ХРАНИЛИЩА является определение методов поиска.

```
public interface CalendarEntryRepository {
    ...
    public CalendarEntry calendarEntryOfId(
            Tenant aTenant,
            CalendarEntryId aCalendarEntryId);

    public Collection<CalendarEntry> calendarEntriesOfCalendar(
            Tenant aTenant,
            CalendarId aCalendarId);

    public Collection<CalendarEntry> overlappingCalendarEntries(
            Tenant aTenant,
            CalendarId aCalendarId,
            TimeSpan aTimeSpan);
}
```

Определение первого метода, `calendarEntryOfId()`, позволяет находить конкретный экземпляр АГРЕГАТА типа `CalendarEntry` по уникальному идентификатору. В этом классе используется явный тип идентификатора — `CalendarEntryId`. Определение второго метода, `calendarEntriesOfCalendar()`, позволяет находить коллекцию всех экземпляров класса `CalendarEntry` для конкретного экземпляра класса `Calendar` по его уникальному идентификатору. В заключение третий метод поиска, `overlappingCalendarEntries()`, возвращает коллекцию всех экземпляров класса `CalendarEntry` для конкретного экземпляра класса `Calendar` по конкретному объекту класса `TimeSpan`.

В частности, этот метод поддерживает поиск мероприятий, запланированных на конкретный период времени по дате и времени.

В заключение можно попробовать выяснить, как экземпляру класса CalendarEntry присваивается его уникальный глобальный идентификатор. Эту функцию также удобно реализовать в ХРАНИЛИЩЕ.

```
public interface CalendarEntryRepository {
    public CalendarEntryId nextIdentity();
    ...
}
```

Любой код, выполняющий создание новых экземпляров класса CalendarEntry, получает новый экземпляр класса CalendarEntryId с помощью метода nextIdentity().

```
CalendarEntry calendarEntry =
    new CalendarEntry(tenant, calendarId,
        calendarEntryRepository.nextIdentity(),
        owner, subject, description, timeSpan, alarm,
        repetition, location, invitees);
```

Подробное описание методов создания идентификаторов, использования предметно-ориентированных и суррогатных идентификаторов, а также выбора моментов времени назначения идентификаторов приведено в главе, посвященной **СУЩНОСТЯМ (ENTITIES) (5)**.

Рассмотрим класс реализации традиционного ХРАНИЛИЩА. Есть несколько способов выбора МОДУЛЯ, в котором будет размещен класс. Некоторые специалисты предпочитают использовать МОДУЛЬ (пакет Java) непосредственно внутри МОДУЛЯ АГРЕГАТА и ХРАНИЛИЩА. В этом случае класс может выглядеть следующим образом.

```
package com.saasovation.collaboration.domain.model.calendar.impl;

public class HibernateCalendarEntryRepository
        implements CalendarEntryRepository {
    ...
}
```

Такое размещение класса позволяет управлять реализациями на УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ, но в специальном пакете, предназначенном для реализаций. В этом случае концепции предметной области четко отделены от концепций, непосредственно связанных с постоянным хранением. Такой стиль объявления интерфейсов в пакетах с длинными именами и их реализации во вложенных пакетах с именем impl широко распространен в проектах на языке Java. Однако

команда *Контекста сотрудничества* решила локализовать все классы технической реализации на УРОВНЕ ИНФРАСТРУКТУРЫ.

```
package com.saasovation.collaboration.infrastructure.persistence;

public class HibernateCalendarEntryRepository


       implements CalendarEntryRepository
{

}
```

Команда применила **ПРИНЦИП ИНВЕРСИИ ЗАВИСИМОСТИ (4)** (Dependency Inversion Principle — DIP) для того, чтобы создать иерархии инфраструктурных концепций. УРОВЕНЬ ИНФРАСТРУКТУРЫ логически размещается выше всех, поэтому ссылки идут только в одном направлении — вниз к УРОВНЮ ПРЕДМЕТНОЙ ОБЛАСТИ.

Класс `HibernateCalendarEntryRepository` — это зарегистрированный управляемый объект (bean) каркаса Spring. Он содержит конструктор без аргументов и другой инфраструктурный управляемый объект внедренной зависимости.

```
import com.saasovation.collaboration.infrastructure
       .persistence.SpringHibernateSessionProvider;

public class HibernateCalendarEntryRepository
       implements CalendarEntryRepository {

   public HibernateCalendarEntryRepository() {
       super();
    }
    ...
    private SpringHibernateSessionProvider sessionProvider;

    public void setSessionProvider(
           SpringHibernateSessionProvider aSessionProvider) {
       this.sessionProvider = aSessionProvider;
    }

    private org.hibernate.Session session() {
       return this.sessionProvider.session();
    }
}
```

Класс `SpringHibernateSessionProvider` также располагается на УРОВНЕ ИНФРАСТРУКТУРЫ в МОДУЛЕ `com.saasovation.collaboration.`

infrastructure.persistence и внедрен в каждое хранилище, основанное на механизме Hibernate. Каждый метод, использующий объект Session библиотеки Hibernate, автоматически вызывает метод session(), чтобы получить его. Метод session() использует экземпляр внедренной зависимости sessionProvider, чтобы получить экземпляр Session, ограниченный потоком выполнения (см. ниже).

Методы add(), addAll(), remove() и removeAll() реализованы следующим образом.

```
package com.saasovation.collaboration.infrastructure.persistence;

public class HibernateCalendarEntryRepository
       implements CalendarEntryRepository {
    ...
    @Override
    public void add(CalendarEntry aCalendarEntry) {
         try {
             this.session().saveOrUpdate(aCalendarEntry);
         } catch (ConstraintViolationException e) {
             throw new IllegalStateException(
                 "CalendarEntry is not unique.", e);
         }
    }

    @Override
    public void addAll(
             Collection<CalendarEntry> aCalendarEntryCollection) {
         try {
             for (CalendarEntry instance : aCalendarEntryCollection) {
                  this.session().saveOrUpdate(instance);
             }
         } catch (ConstraintViolationException e) {
             throw new IllegalStateException(
                       "CalendarEntry is not unique.", e);
         }
    }

    @Override
    public void remove(CalendarEntry aCalendarEntry) {
         this.session().delete(aCalendarEntry);
    }

    @Override
    public void removeAll(
            Collection<CalendarEntry> aCalendarEntryCollection) {
       for (CalendarEntry instance : aCalendarEntryCollection) {
           this.session().delete(instance);
       }
    }
    ...
}
```

Эти методы имеют довольно простую реализацию. Каждый метод автоматически вызывает метод `session()`, чтобы получить свой экземпляр класса `Session` каркаса Hibernate (как указано выше).

Как ни странно, методы `add()` и `addAll()` используют метод `saveOrUpdate()` класса `Session`. Он очень похож на метод добавления в экземпляр класса `Set`. Если оказывается, что один и тот же экземпляр класса `CalendarEntry` добавляется несколько раз, то поведенческий метод `saveOrUpdate()` не выполняет никаких действий. В третьей версии библиотеки Hibernate любая форма обновления не приводит ни к каким последствиям, поскольку они неявно отслеживаются методами модификации состояния объектов. Следовательно, если объекты, добавляемые указанными двумя методами, не являются совершенно новыми, то поведенческий метод ничего не делает.

Операция добавления может вызвать исключение класса `ConstraintViolationException`. Вместо непосредственной пересылки клиентам исключений механизма Hibernate они перехватываются и упаковываются в более удобный экземпляр класса `IllegalStateException`. Кроме того, можно объявить и сгенерировать предметно-ориентированные исключения. Это решение зависит от команды проектировщиков. Ключевой момент заключается в том, что мы хотели бы абстрагироваться от деталей базового механизма постоянного хранения и поэтому изолируем клиентов от всех деталей, включая исключения.

Методы `remove()` и `removeAll()` довольно простые. Им достаточно вызвать метод `Sessiondelete()`, чтобы удалить экземпляр АГРЕГАТА из базы данных. Правда, существует еще одна тонкость, связанная с удалением АГРЕГАТОВ, использующих отображение "один-к-одному" (в частности, это относится к *Контексту идентификации и доступа*). Поскольку каскадные изменения таких отношений выполнить невозможно, необходимо явным образом удалять объекты на каждой из сторон такой связи.

```
package com.saasovation.identityaccess.infrastructure.persistence;

public class HibernateUserRepository implements UserRepository {
    ...
    @Override
    public void remove(User aUser) {
        this.session().delete(aUser.person());
        this.session().delete(aUser);
    }

    @Override
    public void removeAll(Collection<User> aUserCollection) {
        for (User instance : aUserCollection) {
            this.session().delete(instance.person());
            this.session().delete(instance);
        }
    }
    ...
}
```

Сначала необходимо удалить внутренний объект класса Person, а затем — КОРЕНЬ АГРЕГАТА класса User. Если не удалить внутренний объект класса Person, то он потеряет связь со своей таблицей в базе данных. Это достаточная причина для того, чтобы избегать связей типа "один-к-одному" и вместо них использовать ограниченные вырожденные односторонние связи ("многие-к-одному"). Однако я специально решил реализовать двустороннюю связь "один-к-одному", чтобы продемонстрировать неприятности, которые она вызывает.

Отметим, что существуют разные подходы в решению таких проблем. Некоторые специалисты предпочитают зависеть от событий жизненного цикла ORM, вызывающих частичное каскадное удаление объектов. Я преднамеренно избегаю такого подхода, потому что являюсь категоричным противником механизмов постоянного хранения, управляемых АГРЕГАТАМИ, и ярым сторонником механизмов постоянного хранения, зависящих только от ХРАНИЛИЩА. Спорить на эту тему можно бесконечно. Вы должны делать осознанный выбор, но знайте, что эксперты по предметно-ориентированному проектированию осуждают использование механизмов постоянного хранения, управляемых АГРЕГАТАМИ.

Вернемся к классу HibernateCalendarEntryRepository и реализациям его методов поиска.

```
public class HibernateCalendarEntryRepository
        implements CalendarEntryRepository {
    ...
    @Override
    @SuppressWarnings("unchecked")
    public Collection<CalendarEntry> overlappingCalendarEntries(
        Tenant aTenant, CalendarId aCalendarId, TimeSpan aTimeSpan) {
        Query query =
           this.session().createQuery(
                "from CalendarEntry as _obj_ " +
                "where _obj_.tenant = :tenant and " +
                "_obj_.calendarId = :calendarId and " +
                "((_obj_.repetition.timeSpan.begins between " +
                ":tsb and :tse) or " +
                " (_obj_.repetition.timeSpan.ends between " +
                ":tsb and :tse))");

        query.setParameter("tenant", aTenant);
        query.setParameter("calendarId", aCalendarId);
        query.setParameter("tsb", aTimeSpan.begins(), Hibernate.DATE);
        query.setParameter("tse", aTimeSpan.ends(), Hibernate.DATE);

        return (Collection<CalendarEntry>) query.list();
    }

    @Override
    public CalendarEntry calendarEntryOfId(
           Tenant aTenant,
           CalendarEntryId aCalendarEntryId) {
```

```
        Query query =
            this.session().createQuery(
                 "from CalendarEntry as _obj_ " +
                 "where _obj_.tenant = ? and _obj_.calendarEntryId = ?");
                 query.setParameter(0, aTenant);
                 query.setParameter(1, aCalendarEntryId);

            return (CalendarEntry) query.uniqueResult();
    }

    @Override
    @SuppressWarnings("unchecked")
    public Collection<CalendarEntry> calendarEntriesOfCalendar(
            Tenant aTenant, CalendarId aCalendarId) {
            Query query =
                this.session().createQuery(
                     "from CalendarEntry as _obj_ " +
                     "where _obj_.tenant = ? and _obj_.calendarId = ?");

            query.setParameter(0, aTenant);
            query.setParameter(1, aCalendarId);

           return (Collection<CalendarEntry>) query.list();
    }
    ...
}
```

Каждый из этих методов поиска создает экземпляр класса `Query` с помощью своего метода `Session`. В стиле, характерном для запросов Hibernate, команда использует язык HQL для описания критерия поиска и загружает объекты-параметры. Затем происходит выполнение запроса, относящегося к одному или нескольким объектам, указанным в списке. Метод `overlappingCalendarEntries()` выполняет более сложный поиск, в котором необходимо найти все экземпляры класса `CalendarEntry`, перекрывающиеся с конкретным диапазоном дат или временных моментов, или с экземпляром `TimeSpan`.

В заключение рассмотрим реализацию метода `nextIdentity()`.

```
public class HibernateCalendarEntryRepository
       implements CalendarEntryRepository {
   ...
   public CalendarEntryId nextIdentity() {
       return new CalendarEntryId(
              UUID.randomUUID().toString().toUpperCase());
   }
    ...
}
```

Эта конкретная реализация не использует механизм постоянного хранения или базу данных для генерации уникального идентификатора. Вместо этого она использует относительно быстрый и очень надежный генератор `UUID`.

Реализация с помощью системы TopLink

В системе TopLink есть и СЕАНС, и ЕДИНИЦА РАБОТЫ. Этим она отличается от каркаса Hibernate, в котором СЕАНС одновременно является и ЕДИНИЦЕЙ РАБОТЫ.[2] Рассмотрим возможность использования ЕДИНИЦЫ РАБОТЫ отдельно от СЕАНСА, а затем разберемся, как применить их для реализации хранилища.

Если не стремиться извлечь выгоду из абстракции ХРАНИЛИЩА, то систему TopLink можно было бы использовать следующим образом.

```
Calendar calendar = session.readObject(...);

UnitOfWork unitOfWork = session.acquireUnitOfWork();

Calendar calendarToRename = unitOfWork.registerObject(calendar);

calendarToRename.rename("CollabOvation Project Calendar");

unitOfWork.commit();
```

Класс `UnitOfWork` обеспечивает более эффективное использование памяти и процессора, поскольку мы вынуждены явно сообщать ему, какой объект хотим модифицировать. Только в этот момент создается клон, или редактируемая копия, АГРЕГАТА. Как показано выше, метод `registerObject()` возвращает клон экземпляра исходного объекта класса `Calendar`. Именно этот клонированный объект, на который указывает ссылка `calendarToRename`, подлежит редактированию/модификации. Система TopLink может отслеживать изменения, происходящие с объектом, который вы собираетесь модифицировать. При вызове методов `commit()` из класса `UnitOfWork` все модифицированные объекты сохраняются в базе данных.[3]

Добавление новых объектов в ХРАНИЛИЩЕ TopLink можно значительно упростить:

```
...
public void add(Calendar aCalendar) {

this.unitOfWork().registerNewObject(aCalendar);
}
...
```

[2] Я не пытаюсь оценить, какая из систем лучше — TopLink или Hibernate. Система TopLink имеет очень долгую и успешную историю, которая началась задолго до того, как компания Oracle приобрела ее в результате банкротства компании WebGain с последующей "распродажей". *Top* — это аббревиатура от "The Object People", названия компании, которая первой начала разрабатывать этот инструмент, успешно доказывающий свою полезность уже два десятилетия. Я просто использую эти системы для иллюстрации.

[3] Это значит, что данная единица работы не вложена в родительскую единицу работы. В противном случае изменения из загруженной единицы работы смешались бы с изменениями родительской единицы работы. В базе данных фиксируется последнее изменение.

Использование метода registerNewObject() означает, что экземпляр aCalendar является новым. Если бы метод add() был вызван для уже существующего объекта aCalendar, возникла бы ошибка. Здесь можно было бы просто использовать обычный метод registerObject(), похожий на метод saveOrUpdate() из библиотеки Hibernate (см. выше). Оба эти варианта обеспечивают работоспособный интерфейс, ориентированный на имитацию коллекции.

И все же нам нужен способ запроса клона на случай, если мы захотим модифицировать уже существующий АГРЕГАТ. Для этого необходимо найти удобный способ регистрации экземпляра АГРЕГАТА с помощью объекта класса UnitOfWork. До сих пор в нашем обсуждении мы не использовали интерфейс ХРАНИЛИЩА, потому что мы пытались имитировать коллекцию Set и стремились избежать влияния механизма постоянного хранения на интерфейс. Впрочем, мы могли бы решить задачу, не прибегая к механизму постоянного хранения. Рассмотрим один из двух подходов.

```
public Calendar editingCopy(Calendar aCalendar);
// или
public void useEditingMode();
```

В первом варианте метод editingCopy() предписывает экземпляру класса UnitOfWork зарегистрировать заданный экземпляр класса Calendar, создает его клон и возвращает его.

```
...
public Calendar editingCopy(Calendar aCalendar) {

return (Calendar) this.unitOfWork().registerObject(aCalendar);
}
...
```

Именно так и работает метод registerObject(). Разумеется, этот подход нежелателен, но он вполне понятен и не подвержен влиянию механизма постоянного хранения.

Второй подход подразумевает перевод ХРАНИЛИЩА в режим редактирования с помощью метода useEditingMode(). После этого все последующие методы поиска будут автоматически регистрировать все запрашиваемые объекты с помощью экземпляра класса UnitOfWork и возвращать их клоны. В этом случае ХРАНИЛИЩЕ в той или иной степени используется при модификации АГРЕГАТА. Иначе говоря, не важно, как используется ХРАНИЛИЩЕ — только для чтения или для чтения с модификацией. Кроме того, этот вариант отражает использование ХРАНИЛИЩА АГРЕГАТОВ с четко очерченными границами, способствующими успеху транзакций.

Существуют и другие способы проектирования хранилища, ориентированного на имитацию коллекции для системы TopLink, но эта тема выходит за рамки нашей книги.

ХРАНИЛИЩА, ориентированные на механизм постоянного хранения

В ситуациях, когда стиль, ориентированный на имитацию коллекции, не работает, необходимо применять ХРАНИЛИЩЕ, предназначенное для постоянного хранения данных. В этом случае ваш механизм постоянного хранения не обнаруживает и не отслеживает изменения объекта явно или неявно. Именно в таких ситуациях применяется **ФАБРИКА ДАННЫХ (4)**, развернутая в оперативной памяти, или какое-нибудь NoSQL-хранилище типа "ключ–значение". Каждый раз, когда создается новый экземпляр АГРЕГАТА или происходит изменение существующего экземпляра, вы будете обязаны сохранять его в хранилище данных с помощью метода `save()` или другого аналогичного метода ХРАНИЛИЩА.

Есть и другое соображение по поводу выбора подхода, ориентированного на имитацию постоянного хранения, даже если вы используете объектно-реляционный механизм отображения, поддерживающий подход, ориентированный на имитацию коллекции. Что произошло бы, если бы вы спроектировали ХРАНИЛИЩА, ориентированные на имитацию коллекции, а затем решили заменить свою реляционную базу данных хранилищем типа "ключ–значение"? Это сильно отразилось бы на вашем ПРИКЛАДНОМ УРОВНЕ, поскольку он должен быть перестроен на использование метода `save()` везде, где происходит изменение АГРЕГАТА. Вы также захотели бы избавить свои ХРАНИЛИЩА от методов `add()` и `addAll()`, поскольку они больше не нужны. Если замена вашего механизма постоянного хранения в будущем весьма вероятна, возможно, лучше всего было бы спроектировать более гибкий интерфейс. В этом случае ваш текущий объектно-реляционный механизм отображения может пропустить необходимый вызов метода `save()`, который вы можете перехватить лишь позднее, когда больше не будет поддержки ЕДИНИЦЫ РАБОТЫ.[4] Зато шаблон ХРАНИЛИЩЕ позволит полностью заменить ваш механизм постоянного хранения с минимальным влиянием на ваше приложение.

[4] Вы можете создать тесты **ПРИКЛАДНОЙ СЛУЖБЫ (14)**, учитывающие сохранение изменений при необходимости. Можно также создать реализацию хранилища в оперативной памяти (см. дальнейший основной текст главы), которая проверяла бы необходимость сохранения изменений.

Выводы по ХРАНИЛИЩАМ, ориентированным на имитацию механизмов постоянного хранения

Мы должны явным образом записывать новые и измененные объекты в хранилище с помощью метода `put()`, заменяя все значения, ранее связанные с заданным ключом. Использование таких хранилищ данных значительно упрощает запись и чтение АГРЕГАТОВ. По этой причине их иногда называют ХРАНИЛИЩАМИ АГРЕГАТОВ (AGGREGATE STORES) или АГРЕГАТНО-ОРИЕНТИРОВАННЫМИ БАЗАМИ ДАННЫХ (AGGREGATE-ORIENTED DATABASES).

При использовании фабрики данных, развернутой в оперативной памяти, например GemFire или Oracle Coherence, хранилище представляет собой реализацию класса `Map`, имитирующего класс `java.util.HashMap`, в котором каждый отображаемый элемент рассматривается как *запись*. Аналогично при использовании хранилища NoSQL, такого как MongoDB или Riak, механизм постоянного хранения объектов создает иллюзию коллекции, а не таблиц, строк и столбцов. В таких хранилищах хранятся пары "ключ–значение". Они представляют собой эффективные хранилища типа `Map`, но в качестве основной среды для постоянного хранения данных используют диск, а не оперативную память.

Несмотря на то что оба этих вида механизмов постоянного хранения отдаленно напоминают коллекцию `Map`, мы вынуждены применять метод `put()` как для новых, так и для измененных объектов в хранилище, заменяя значение, ранее связанное с заданным ключом. Это происходит даже тогда, когда измененный объект с логической точки зрения представляет собой тот же самый объект, который уже был сохранен, потому что ЕДИНИЦА РАБОТЫ обычно не отслеживает изменения и не поддерживает демаркацию транзакций для управления атомарными записями. Вместо этого каждый вызов методов `put()` и `putAll()` представляет собой отдельную логическую транзакцию.

Использование любого из указанных видов баз данных значительно упрощает чтение и запись АГРЕГАТОВ. Например, посмотрим, насколько просто добавить объект класса `Product` (*Контекст управления гибким проектированием*) в grid-систему управления данными Coherence, а затем прочитать его.

```
cache.put(product.productId(), product);
// позже ...
product = cache.get(productId);
```

Здесь экземпляр класса `Product` автоматически сериализуется в объект класса `Map` с помощью стандартных средств сериализации языка Java. Простота этого интерфейса обманчива. В предметных областях, требующих высокой производительности, необходимо сделать намного больше. Если в системе не зарегистрирован специальный провайдер пользовательской сериализации, то система

Coherence применяет средства сериализации языка Java. Использование стандартных средств сериализации языка Java — не лучшее решение. Они требуют дополнительных байтов для представления каждого объекта и работают относительно медленно.[5] Нет смысла покупать высокопроизводительную ФАБРИКУ ДАННЫХ, а затем ограничивать количество кешируемых ею объектов и тормозить из-за медленной сериализации. Таким образом, следует помнить, что при использовании ФАБРИКИ ДАННЫХ, например, в систему включается механизм распределения данных. Это часто приводит к включению в модель предметной области дополнительного механизма пользовательской или хотя бы специализированной сериализации. Это может привести к разным решениям, по крайней мере на уровне реализации.

Итак, использование систем GemFire или Coherence, хранилищ типа "ключ-значение" MongoDB или Riak или других механизмов NoSQL, вероятно, потребует применения быстрых и компактных инструментов для превращения АГРЕГАТОВ в сериализованную или документную форму, а затем их возврата в исходную форму. К счастью, решение этих проблем не представляет проблемы. Например, создание оптимальной сериализации для АГРЕГАТА, который сохраняется с помощью систем GemFire или Coherence, не сложнее, чем создание описания отображений для объектно-реляционного механизма отображения. Но это не так просто, как обычный вызов методов `put()` и `get()` из класса `Map`.

Далее я продемонстрирую, как можно создать ХРАНИЛИЩЕ, ориентированное на имитацию механизма постоянного хранения для системы Coherence, а затем покажу несколько приемов, позволяющих сделать то же самое с помощью системы MongoDB.

Реализация с помощью системы Coherence

Как и в случае ХРАНИЛИЩА, ориентированного на имитацию коллекции, сначала определим интерфейс и его реализацию. Ниже показан интерфейс, ориентированный на имитацию механизма постоянного хранения, который определяет методы сохранения для grid-системы Oracle Coherence.

```
package com.saasovation.agilepm.domain.model.product;

import java.util.Collection;

import com.saasovation.agilepm.domain.model.tenant.Tenant;

public interface ProductRepository {
```

[5] Кроме того, этот механизм ограничивает клиентов системы Coherence только средствами языка Java, в то время как клиенты платформы .NET и языка C++ могут использовать grid-данные, если предусмотрена сериализация в формате Portable Object Format (POF).

```
    public ProductId nextIdentity();
    public Collection<Product> allProductsOfTenant(Tenant aTenant);
    public Product productOfId(Tenant aTenant, ProductId aProductId);
    public void remove(Product aProduct);
    public void removeAll(Collection<Product> aProductCollection);
    public void save(Product aProduct);
    public void saveAll(Collection<Product> aProductCollection);
}
```

Этот класс `ProductRepository` похож на класс `CalendarEntryRepository` из предыдущего раздела. Он отличается только тем, что позволяет включать экземпляры АГРЕГАТА в имитируемую коллекцию. В данном случае мы используем методы `save()` и `saveAll()`, а не методы `add()` и `addAll()`. Методы обоих стилей делают примерно одно и то же. Основная разница между ними заключается в том, как клиенты используют эти методы. Напомним, что при использовании стиля, имитирующего коллекцию, экземпляры АГРЕГАТА добавляются только при их создании. При использовании стиля, имитирующего механизм постоянного хранения, экземпляры АГРЕГАТА должны сохраняться как при их создании, так и при их модификации.

```
Product product = new Product(...);

productRepository.save(product);

// позднее ...

Product product =
productRepository.productOfId(tenantId, productId);

product.reprioritizeFrom(backlogItemId, orderOfPriority);

productRepository.save(product);
```

В остальном они различаются деталями реализации, поэтому перейдем к их исследованию. Сначала рассмотрим инфраструктуру системы Coherence, которую необходимо создать, чтобы перейти к кешу grid-системы.

```
package com.saasovation.agilepm.infrastructure.persistence;

import com.tangosol.net.CacheFactory;
import com.tangosol.net.NamedCache;

public class CoherenceProductRepository
        implements ProductRepository {
    private Map<Tenant,NamedCache> caches;

    public CoherenceProductRepository() {
        super();
```

```
        this.caches = new HashMap<Tenant,NamedCache>();
    }
    ...
    private synchronized NamedCache cache(TenantId aTenantId) {
        NamedCache cache = this.caches.get(aTenantId);

        if (cache == null) {
            cache = CacheFactory.getCache(
                    "agilepm.Product." + aTenantId.id(),
                    Product.class.getClassLoader());

            this.caches.put(aTenantId, cache);
        }

        return cache;
    }
    ...
}
```

В *Контексте управления гибким проектированием* команда решила разместить техническую реализацию ХРАНИЛИЩА на ИНФРАСТРУКТУРНОМ УРОВНЕ.

Наряду с простым конструктором без параметров существует ключевой элемент системы Coherence — класс `NamedCache`. Обратите внимание на то, что импортируемые классы `CacheFactory` и `NamedCache` так или иначе относятся к процессам создания или присоединения, а также к использованию кеша. Оба эти класса включены в пакет `com.tangosol.net`.

Закрытый метод `cache()` позволяет получить объект класса `NamedCache`. Этот метод в режиме отложенного выполнения получает кеш при первой попытке ХРАНИЛИЩА использовать его. Это в основном объясняется тем, что каждый кеш предназначен для конкретного объекта класса `Tenant`, и ХРАНИЛИЩЕ должно подождать, пока не будет вызван открытый метод, чтобы получить доступ к объекту класса `TenantId`. В системе Coherence существует множество возможностей для разработки стратегий кеширования. В нашем случае команда решила выполнять кеширование с помощью следующего пространства имен.

1. Первый уровень задается сокращенным именем ограниченного контекста: `agilepm`.
2. Второй уровень определяется простым именем АГРЕГАТА: `Product`.
3. Третий уровень задается уникальным идентификатором каждого арендатора: `TenantId`.

Этот подход имеет несколько преимуществ. Во-первых, модель каждого ОГРАНИЧЕННОГО КОНТЕКСТА, АГРЕГАТ и арендатор, работающий под управлением системы Coherence, могут настраиваться и масштабироваться по отдельности. Кроме того, каждый АГРЕГАТ полностью отделен от всех остальных, поэтому

запрос к одному арендатору не может случайно содержать объекты других арендаторов. По этой же причине в механизме постоянного хранения mySQL "расщепляются" все таблицы с идентификатором арендатора. Правда, в данном случае это происходит еще более явно. Более того, каждый раз, когда метод поиска обращается ко всем АГРЕГАТАМ для ответа на запрос конкретного арендатора, сам запрос на самом деле не нужен. Метод поиска просто обращается ко всем записям в кеше системы Coherence. Вы увидите эту оптимизацию позднее при реализации метода `allProductsOfTenant()`.

Каждый раз, когда происходит создание или присоединение объекта класса `NamedCache`, он включается в объект класса `Map`, связанный с экземпляром переменной `caches`. Это позволяет после первого использования при всех последующих обращениях быстро просматривать каждый кеш с помощью объекта класса `TenantId`.

Существует слишком много вариантов конфигураций и настроек системы Coherence. Эта тема заслуживает отдельного обсуждения, и публикации на эту тему уже появляются. Интересующиеся читатели могут обратиться к работе Алекса Сеовича (Aleks Seović) [Seović]. Вот как выглядит реализация.

```
public class CoherenceProductRepository
       implements ProductRepository {
    ...
    @Override
    public ProductId nextIdentity() {
       return new ProductId(
              java.util.UUID.randomUUID()
             .toString()
             .toUpperCase());

    }
    ...
}
```

Метод `nextIdentity()` в классе `ProductRepository` реализован так же, как в классе `CalendarEntryRepository`. Он получает идентификатор UUID и использует его для создания экземпляра класса `ProductId`, который потом возвращает.

```
public class CoherenceProductRepository
       implements ProductRepository {
    ...
    @Override
    public void save(Product aProduct) {
       this.cache(aProduct.tenantId())
              .put(this.idOf(aProduct), aProduct);
       }
```

```
    @Override
    public void saveAll(Collection<Product> aProductCollection) {
       if (!aProductCollection.isEmpty()) {
           TenantId tenantId = null;

           Map<String,Product> productsMap =
              new HashMap<String,Product>(aProductCollection.size());

           for (Product product : aProductCollection) {
              if (tenantId == null) {
                  tenantId = product.tenantId();
              }
              productsMap.put(this.idOf(product), product);
           }

           this.cache(tenantId).putAll(productsMap);
       }
    }
    ...
    private String idOf(Product aProduct) {
        return this.idOf(aProduct.productId());
    }

    private String idOf(ProductId aProductId) {
        return aProductId.id();
    }
}
```

Для того чтобы сохранить новый или модифицированный экземпляр класса `Product` в grid-системе управления данными, используется метод `save()`. Этот метод использует метод `cache()`, чтобы получить экземпляр класса `NamedCache` для экземпляра класса `TenantId` в экземпляре класса `Product`. Затем он помещает экземпляр класса `Product` в экземпляр класса `NamedCache`. Обратите внимание на использование метода `idOf()`, который имеет два варианта: один — для экземпляра класса `Product`, а другой — для экземпляра класса `ProductId`. Оба эти варианта возвращают уникальный идентификатор экземпляра класса `Product` или `ProductId` в виде экземпляра класса `String`. Итак, метод `put()` в классе `NamedCache`, реализующем класс `java.util.Map`, получает ключ в виде экземпляра класса `String` и значение в виде экземпляра класса `Product`.

Метод `saveAll()` может показаться немного сложнее, чем вы ожидали. Почему бы просто не пройтись по экземпляру `aProductCollection`, применяя метод `save()` к каждому элементу? Это можно было бы сделать. Однако в зависимости от используемого кеша системы Coherence каждый вызов метода `put()` потребовал бы запроса к сети. Следовательно, лучше собрать все экземпляры класса `Product` в простой локальный экземпляр класса `HashMap` и применить к ним метод `putAll()`. Это сокращает время сетевой задержки до минимума и является почти оптимальным решением.

```
public class CoherenceProductRepository
       implements ProductRepository {
    ...
    @Override
    public void remove(Product aProduct) {
       this.cache(aProduct.tenant()).remove(this.idOf(aProduct));
    }

    @Override
    public void removeAll(Collection<Product> aProductCollection) {
       for (Product product : aProductCollection) {
            this.remove(product);
       }
    }
    ...
}
```

Реализация метода `remove()` работает в точном соответствии с ожиданиями. Однако реализация методов `saveAll()` и `removeAll()` может показаться удивительной. В конце концов, существует ли способ удалить совокупность записей? Нет, такого способа в стандартном интерфейсе `java.util.Map` нет, а значит, его нет и в системе Coherence. Следовательно, в данном случае мы просто обходим коллекцию `aProductCollection` и применяем метод `remove()` к каждому элементу. С учетом возможных последствий удаления только части коллекции из-за сбоя системы Coherence этот способ может показаться опасным. Разумеется, следует тщательно взвесить возможные последствия применения метода `removeAll()`, но следует помнить, что основное преимущество ФАБРИК ДАННЫХ, таких как GemFire и Coherence, — это избыточность и высокая степень доступности.

В заключение рассмотрим реализации интерфейсного метода, обеспечивающие разные способы поиска экземпляров класса `Product`.

```
public class CoherenceProductRepository
       implements ProductRepository {
   ...
   @SuppressWarnings("unchecked")
   @Override
   public Collection<Product> allProductsOfTenant(Tenant aTenant) {
       Set<Map.Entry<String, Product>> entries =
           this.cache(aTenant).entrySet();

       Collection<Product> products =
           new HashSet<Product>(entries.size());

       for (Map.Entry<String, Product> entry : entries) {
           products.add(entry.getValue());
       }
```

```
        return products;
    }

    @Override
    public Product productOfId(Tenant aTenant, ProductId aProductId) {
        return (Product) this.cache(aTenant).get(this.idOf(aProductId));
    }
    ...
}
```

Метод `productOfId()` должен лишь вызвать базовый метод `get()` из объекта класса `NamedCache`, возвращая запрошенный идентификатор экземпляра класса `Product`.

Метод `allProductsOfTenant()` уже упоминался выше. Вместо применения сложного процесса фильтрации данных, существующего в системе Coherence, достаточно всего лишь запросить у grid-системы управления данными все экземпляры класса `Product`, находящиеся в конкретном экземпляре класса `NamedCache`. Поскольку каждый кеш связан с отдельным арендатором, каждый экземпляр АГРЕГАТА, находящийся в кеше, удовлетворяет запрос.

Все это заключено в оболочку класса `CoherenceProductRepository`. Эта реализация показывает, как создать абстрактный интерфейс с помощью системы Coherence в виде клиента, записывающего данные в grid-кеш, чтобы найти их там позднее. Мы не касались конфигурирования и настройки системы Coherence, а также создания индексов для каждого кеша или проектирования компактного высокопроизводительного инструмента сериализации каждого объекта предметной области. Все эти вопросы не относятся к ХРАНИЛИЩАМ. Они подробно освещены в работе [Seović].

Реализация с помощью системы MongoDB

Как и любая другая реализация ХРАНИЛИЩА, реализация с помощью системы MongoDB имеет определенные базовые принципы. По существу, реализация MongoDB похожа на версию системы Coherence. Ниже приведен общий список требуемых нам средств.

1. Средства сериализации экземпляров АГРЕГАТА в формате MongoDB, а также средства десериализации из этого формата в восстановленный экземпляр АГРЕГАТА. Система MongoDB использует специальную версию формата JSON под названием BSON, т.е. бинарный формат JSON.
2. Уникальный идентификатор, генерируемый системой MongoDB и присвоенный АГРЕГАТУ.
3. Ссылка на узел/кластер системы MongoDB.

4. Уникальная коллекция, в которой хранится каждый тип АГРЕГАТА. Все экземпляры каждого типа АГРЕГАТА должны храниться в виде сериализованных документов (пар "ключ–значение") в собственной коллекции.

Рассмотрим реализацию ХРАНИЛИЩА шаг за шагом. Поскольку мы снова используем класс `ProductRepository`, можно сравнить эту реализацию с реализацией, полученной с помощью системы Coherence (см. предыдущий раздел).

```
public class MongoProductRepository
       extends MongoRepository<Product>
       implements ProductRepository {

    public MongoProductRepository() {
       super();

       this.serializer(new BSONSerializer<Product>(Product.class));
    }
    ...
}
```

Для сериализации и десериализации всех экземпляров класса `Product` эта реализация использует экземпляр класса `BSONSerializer` (который на самом деле относится к суперклассу `MongoRepository`). Я не хочу вдаваться в подробности устройства класса `BSONSerializer`. Это специальный класс системы MongoDB, который создает экземпляры класса `DBObject` из экземпляров класса `Product` (и агрегатов любого другого типа), и наоборот. Этот класс приводится вместе с другими образцами кода.

О классе `BSONSerializer` следует помнить несколько вещей. Основная часть сериализации и десериализации выполняется с помощью прямого доступа к полям. Это освобождает ваши объекты предметной области от необходимости реализовывать методы получатели и установщики (`get-` и `set`-методы) JavaBean, что помогает избежать **АНЕМИЧНОЙ МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ** [Fowler, Anemic]. Поскольку мы не хотим использовать методы для доступа к полям, придется в какой-то момент выполнить переход от одной версии типа АГРЕГАТА к другой. Для этого можно определить отображение каждого поля, замещаемое при десериализации.

```
public class MongoProductRepository
       extends MongoRepository<Product>
       implements ProductRepository {

    public MongoProductRepository() {
       super();

       this.serializer(new BSONSerializer<Product>(Product.class));
```

```
        Map<String, String> overrides = new HashMap<String, String>();
        overrides.put("description", "summary");
        this.serializer().registerOverrideMappings(overrides);
    }
    ...
}
```

В этом примере мы предполагаем, что предыдущая версия класса `Product` имеет поле с именем `description`. В последующей версии это поле будет называться `summary`. Для решения этой проблемы мы могли бы выполнить сценарий обхода всех коллекций системы MongoDB, используемый для сохранения экземпляров класса `Product` для каждого арендатора. Однако для этого пришлось бы долго и нудно выполнять множество операций, что совершенно непрактично. В качестве альтернативы мы можем просто попросить экземпляр класса `BSONSerializer` отобразить каждое поле в формате BSON экземпляра класса `Product` с именем `description` в поле с именем `summary`. Затем, при сериализации экземпляра класса `Product` обратно в вид `DBObject` и сохранении в коллекции MongoDB, новая сериализация будет содержать поле с именем `summary`, а не `description`. Разумеется, это значит, что все экземпляры класса `Product` никогда не считываются и не записываются обратно в хранилище с устаревшим именем поля `description`. Следует тщательно взвесить все аргументы за и против такой отложенной миграции.

Далее, нам нужен метод, с помощью которого система MongoDB генерировала бы уникальный идентификатор для каждого экземпляра АГРЕГАТА.

```
public class MongoProductRepository
        extends MongoRepository<Product>
        implements ProductRepository {
    ...
    public ProductId nextIdentity() {
        return new ProductId(new ObjectId().toString());
    }
    ...
}
```

Мы по-прежнему используем метод `nextIdentity()`, но в этой реализации мы инициализируем экземпляр класса `ProductId` значением типа `String` нового экземпляра класса `ObjectId`. Основная причина заключается в том, что мы хотим, чтобы система MongoDB использовала тот же самый уникальный идентификатор, который хранится в самом экземпляре АГРЕГАТА. Таким образом, когда мы сериализуем экземпляр класса `Product` (или экземпляр другого типа в другой реализации ХРАНИЛИЩА), мы можем попросить экземпляр класса `BSONSerializer` отобразить этот идентификатор в специальный ключ MongoDB `_id`.

```
public class BSONSerializer<T> {
    ...
    public DBObject serialize(T anObject) {
         DBObject serialization = this.toDBObject(anObject);

         return serialization;
    }

    public DBObject serialize(String aKey, T anObject) {
         DBObject serialization = this.serialize(anObject);

         serialization.put("_id", new ObjectId(aKey));

         return serialization;
    }
    ...
}
```

Первый метод `serialize()` не поддерживает такое отображение `_id`, предоставляя клиентам возможность запоминать соответствующие идентификаторы или нет. Далее рассмотрим реализацию метода `save()`.

```
public class MongoProductRepository
       extends MongoRepository<Product>
       implements ProductRepository {
   ...
   @Override
   public void save(Product aProduct) {
       this.databaseCollection(
           this.collectionName(aProduct.tenantId()))
           .save(this.serialize(aProduct));
   }
   ...
}
```

Аналогично реализации интерфейса ХРАНИЛИЩА с помощью системы Coherence мы получаем коллекцию, связанную с арендатором, в которой хранятся экземпляры класса `Product`, соответствующие заданному экземпляру класса `TenantId`. Это позволяет создать объект класса `DBCollection` из класса `DB`. Для того чтобы получить объект класса `DBCollection`, необходимо внести в абстрактный базовый класс `MongoRepository` следующие изменения.

```
public abstract class MongoRepository<T> {
    ...
    protected DBCollection databaseCollection(
          String aDatabaseName,
          String aCollectionName) {
             return MongoDatabaseProvider
                    .database(aDatabaseName)
```

```
                .getCollection(aCollectionName);
    }
    ...
}
```

Мы используем экземпляр класса `MongoDatabaseProvider` для того, чтобы установить связь с экземпляром базы данных, который возвращает объект класса `DB`. Из полученного объекта класса `DB` мы запрашиваем экземпляр класса `DBCollection`. Как видно в конкретной реализации ХРАНИЛИЩА, имя коллекции представляет собой комбинацию текста `"product"` и полного идентификатора арендатора. В *Контексте управления гибким проектированием* используется специальная база данных с именем `agilepm`, очень похожим на имя кеша в системе Coherence.

```
public class MongoProductRepository
        extends MongoRepository<Product>
    implements ProductRepository {
    ...
    protected String collectionName(TenantId aTenantId) {
        return "product" + aTenantId.id();
    }

    protected String databaseName() {
        return "agilepm";
    }
    ...
}
```

Аналогично классу `SpringHibernateSessionProvider`, описанному выше, класс `MongoDatabaseProvider` представляет собой средство для извлечения экземпляра класса `DB` для приложения.

Тот же самый экземпляр класса `DBCollection` используется для вызова метода `save()` и поиска экземпляров класса `Product`.

```
public class MongoProductRepository
    extends MongoRepository<Product>
    implements ProductRepository {
    ...
     @Override
     public Collection<Product> allProductsOfTenant(
            TenantId aTenantId) {
        Collection<Product> products = new ArrayList<Product>();

        DBCursor cursor =
            this.databaseCollection(
                    this.databaseName(),
                    this.collectionName(aTenantId)).find();
```

```
            while (cursor.hasNext()) {
                DBObject dbObject = cursor.next();

                Product product = this.deserialize(dbObject);

                products.add(product);
            }

            return products;
        }

        @Override
        public Product productOfId(
               TenantId aTenantId, ProductId aProductId) {
            Product product = null;

            BasicDBObject query = new BasicDBObject();

            query.put("productId",
                  new BasicDBObject("id", aProductId.id()));

            DBCursor cursor =
                  this.databaseCollection(
                              this.databaseName(),
                              this.collectionName(aTenantId)).find(query);

            if (cursor.hasNext()) {
                product = this.deserialize(cursor.next());
            }

            return product;
    }
    ...
}
```

Реализация класса `allProductsOfTenant()` снова очень похожа на реализацию, полученную с помощью системы Coherence. Мы просто просим экземпляр класса `DBCollection`, зависящий от арендатора, применить метод `find()` ко всем экземплярам. Что касается метода `productOfId()`, то на этот раз мы передаем методу `find()` из класса `DBCollection` объект класса `DBObject`, описывающий конкретный экземпляр класса `Product`, который необходимо найти. В обоих версиях метода поиска мы используем возвращенный объект класса `DBCursor`, чтобы получить все экземпляры или только первый соответственно.

Дополнительные поведенческие функции

Иногда возникает необходимость включить в интерфейс ХРАНИЛИЩА дополнительную функцию, которая отличается от функций, описанных в предыдущих

разделах. Например, удобно было бы иметь функцию, возвращающую количество всех экземпляров в коллекции агрегатов. Допустим, что эта функция называется *count*. Однако, поскольку ХРАНИЛИЩЕ должно как можно точнее имитировать коллекцию, можно было бы рассмотреть следующий метод.

```
public interface CalendarEntryRepository {
    ...
    public int size();
}
```

Метод `size()` делает именно то, что подразумевает стандартный интерфейс `java.util.Collection`. При работе с библиотекой Hibernate эта реализация могла бы выглядеть следующим образом.

```
public class HibernateCalendarEntryRepository
       implements CalendarEntryRepository {
    ...
    public int size() {
       Query query =
          this.session().createQuery(
               "select count(*) from CalendarEntry");

      int size = ((Integer) query.uniqueResult()).intValue();

      return size;
   }
}
```

В хранилище данных (базе данных или grid-системе) можно выполнять и другие вычисления, необходимые для удовлетворения насущных нефункциональных требований. Например, такая ситуация складывается, если перемещение данных из хранилища в место их обработки в соответствии с бизнес-логикой является слишком медленным процессом. В таком случае целесообразно включить код в саму базу данных. Это можно осуществить с помощью процедур, хранимых в базе данных или обработчиков записей, доступных в системе Coherence. Однако такие реализации обычно лучше всего оставлять под управлением **СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ (7)**, поскольку именно они используются для хранения операций, специфичных для предметной области и не имеющих состояния.

Время от времени удобно запрашивать части АГРЕГАТОВ из ХРАНИЛИЩА, не обращаясь к самому КОРНЮ. Это происходит, например, если АГРЕГАТ содержит большую коллекцию СУЩНОСТЕЙ какого-то типа, и вы хотите получить доступ только к экземплярам, которые соответствуют определенному критерию. Конечно, это было бы целесообразным, только если бы АГРЕГАТ допускал такой доступ с помощью навигации через КОРЕНЬ. Нет смысла разрабатывать ХРАНИЛИЩЕ,

чтобы обеспечить доступ к частям, к которым нет доступа через КОРЕНЬ АГРЕГАТА. Это нарушило бы контракт АГРЕГАТА. Я также полагаю, что вы не стали бы разрабатывать такое хранилище, просто чтобы угодить клиенту. Считаю, что принятие такого решения должно основываться, в первую очередь, на соображениях производительности при условии, что навигация через КОРЕНЬ не станет узким местом. Методы, обеспечивающие такую оптимизацию доступа, должны иметь те же характеристики, что и другие методы поиска, ранее описанные в этой главе, возвращая при этом части, содержащиеся в ХРАНИЛИЩЕ, а не КОРНЕВЫЕ СУЩНОСТИ. Напоминаю: будьте осторожны!

Существует еще одна причина, которая могла бы заставить вас разрабатывать специальные методы поиска. Некоторые сценарии использования вашей системы могут не соответствовать точным контурам единственного типа АГРЕГАТА при визуализации представлений данных предметной области. Они могут выходить за пределы типов, возможно, просто образуя определенные части одного или нескольких АГРЕГАТОВ. В таких ситуациях вы могли бы не выбрать целые АГРЕГАТЫ разных типов в рамках единственной транзакции, чтобы затем программно поместить их в единственный контейнер и предоставить его клиенту. Вместо этого вы могли бы использовать то, что вызывают *сценарием использования оптимального запроса* (use case optimal query). В нем вы определили бы сложный запрос к механизму постоянного хранения данных, динамично помещая результаты в **ОБЪЕКТ-ЗНАЧЕНИЕ (6)**, специально предназначенный для данного сценария использования.

То, что в некоторых случаях ХРАНИЛИЩЕ возвращает ОБЪЕКТ-ЗНАЧЕНИЕ, а не экземпляр АГРЕГАТА, не должно вас удивлять. АГРЕГАТ, имеющий метод `size()`, возвращает очень простое значение в виде целого числа, равного количеству всех экземпляров АГРЕГАТА, которые в нем содержатся. Сценарий использования оптимального запроса просто расширяет эту концепцию, чтобы вернуть более сложное ЗНАЧЕНИЕ, удовлетворяющее более сложным требованиям клиентов.

Если окажется, что вы должны создать много методов поиска, поддерживающих сценарий использования оптимального запроса к нескольким ХРАНИЛИЩАМ, то, вероятно, вы написали плохой код. Прежде всего, эта ситуация может быть свидетельством того, что вы неправильно очертили границы АГРЕГАТА и пропустили возможность определить один или несколько АГРЕГАТОВ другого типа. Эту ошибку можно назвать так: *"ХРАНИЛИЩЕ маскирует недостатки проектирования АГРЕГАТА"*.

А что делать, если вы оказались в такой ситуации и ваш анализ показывает, что границы агрегата определены правильно? Это может указывать на то, что вам необходимо рассмотреть возможность использования шаблона **CQRS (4)**.

Управление транзакциями

Модель предметной области и содержащий ее УРОВЕНЬ ПРЕДМЕТНОЙ ОБЛАСТИ — неправильное место для управления транзакциями.[6] Операции, связанные с моделью, обычно слишком мелкомодульные, чтобы самостоятельно управлять транзакциями, и не должны зависеть от них. Если механизмы управления транзакциями нельзя включать в эту модель, то куда их поместить?

Как правило, управление транзакциями со стороны механизма постоянного хранения данных в модели предметной области включается в **ПРИКЛАДНОЙ УРОВЕНЬ (14)**.[7] Обычно мы создаем **ФАСАД (FACADE)** [Гамма и др.] для каждого из основных сценариев использования, группируя их по приложениями или системам. ФАСАД содержит крупномодульные бизнес-методы, обычно по одному на каждый поток сценария использования (который может быть единственном в этом сценарии). Каждый такой бизнес-метод координирует задачу в соответствии со сценарием использования. Когда бизнес-метод ФАСАДА вызывается на **УРОВНЕ ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА (14)**, человеком или системой, бизнес-метод начинает транзакцию, а затем действует как клиент модели предметной области. Успешно завершив взаимодействие с моделью предметной области, бизнес-метод ФАСАДА фиксирует начатую им транзакцию. Если возникает ошибка или исключение, препятствующее завершению задачи сценария использования, другой управляющий бизнес-метод выполняет откат транзакции.

Транзакцией можно управлять декларативно или непосредственно из кода. Независимо от того являются ли ваши транзакции декларативными или управляются пользователем, следует сделать следующее.

```
public class SomeApplicationServiceFacade {
    ...
    public void doSomeUseCaseTask() {
        Transaction transaction = null;

        try {
            transaction = this.session().beginTransaction();

            // используем модель предметной области...

            transaction.commit();
```

[6] Обратите внимание на то, что в некоторых механизмах постоянного хранения данных управление транзакциями либо просто не предусмотрено, либо работает не так, как это принято для транзакций ACID, типичных для реляционных баз данных. Это характерно и для системы Coherence, и для многих хранилищ NoSQL, поэтому излагаемый здесь материал к этим механизмам хранения данных не относится.

[7] Прикладной уровень управляет и другими аспектами, например безопасностью, но мы это здесь обсуждать не будем.

```
        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
        }
    }
}
```

Для того чтобы записать изменения модели предметной области, произошедшие в результате транзакции, реализации ХРАНИЛИЩА должны иметь доступ к СЕАНСУ или ЕДИНИЦЕ РАБОТЫ, связанным с транзакцией, которую начал ПРИКЛАДНОЙ УРОВЕНЬ. В этом случае модификации, сделанные на УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ, будут правильно зафиксированы в соответствующей базе данных или будет выполнен откат.

Существует так много возможных вариантов решения этой задачи, что я не буду описывать все возможности. Отмечу лишь, что промышленные контейнеры Java и контейнеры с инверсией управления, такие как Spring, предоставляют средства для выполнения всех описанных выше операций, которые обычно не вызывают затруднений. Вопрос лишь в выборе средства, подходящего для вашей среды. Например, покажем, как сделать это с помощью каркаса Spring.

```
<tx:annotation-driven transaction-manager="transactionManager"/>

<bean
    id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="configLocation">
        <value>classpath:hibernate.cfg.xml</value>
    </property>
</bean>

<bean
    id="sessionProvider"
    class="com.saasovation.identityaccess.infrastructure
          .persistence.SpringHibernateSessionProvider"
    autowire="byName">
</bean>

<bean
    id="transactionManager"
    class="org.springframework.orm.hibernate3
          .HibernateTransactionManager">
    <property name="sessionFactory">
       <ref bean="sessionFactory"/>
    </property>
</bean>

<bean
```

```
    id="abstractTransactionalServiceProxy"
    abstract="true"
    class="org.springframework.transaction.interceptor
           .TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="transactionAttributes">
       <props>
          <prop key="*">PROPAGATION_REQUIRED</prop>
       </props>
    </property>
</bean>
```

Сконфигурированный пакет `sessionFactory` предоставляет средства для получения объекта класса `Session` от каркаса Hibernate. Пакет `sessionProvider` используется для того, чтобы связать объект класса `Session`, полученный из пакета `sessionFactory`, с текущим объектом класса `Thread`, описывающим поток выполнения. Пакет `sessionProvider` может использоваться ХРАНИЛИЩАМИ, основанными на каркасе Hibernate, когда им понадобится связать экземпляр класса `Session` с объектом класса `Thread`, в котором выполняется работа. В пакете `transactionManager` используется пакет `sessionFactory` для получения транзакций каркаса Hibernate и управления ими. Оставшийся пакет `abstractTransactionalServiceProxy` используется как прокси-объект для объявления транзакционных пакетов с помощью конфигурации каркаса Spring. Первое объявление позволяет объявлять транзакции с помощью аннотаций Java. Это может оказаться намного удобнее, чем конфигурация.

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Объединив все это в одно целое, мы можем теперь объявить конкретный транзакционный бизнес-метод ФАСАДА с помощью простой аннотации.

```
public class SomeApplicationServiceFacade {
    ...
    @Transactional
    public void doSomeUseCaseTask() {

    // используем модель предметной области...
    }
}
```

По сравнению с предыдущим примером управления транзакцией это позволяет скрыть содержимое бизнес-метода и сосредоточиться на самой координации задачи. Благодаря этой аннотации при вызове бизнес-метода каркас Spring

автоматически начинает транзакцию, а когда метод завершается, либо фиксируется транзакция, либо выполняется откат.

Рассмотрим исходный код пакета `sessionProvider` в *Контексте идентификации и доступа*.

```
package com.saasovation.identityaccess.infrastructure.persistence;

import org.hibernate.Session;
import org.hibernate.SessionFactory;

public class SpringHibernateSessionProvider {

    private static final ThreadLocal<Session> sessionHolder =
            new ThreadLocal<Session>();

    private SessionFactory sessionFactory;

    public SpringHibernateSessionProvider() {
        super();
    }

    public Session session() {
        Session threadBoundsession = sessionHolder.get();
        if (threadBoundsession == null) {
            threadBoundsession = sessionFactory.openSession();
            sessionHolder.set(threadBoundsession);
        }
        return threadBoundsession;
    }

    public void setSessionFactory(SessionFactory aSessionFactory) {
        this.sessionFactory = aSessionFactory;
    }
}
```

Поскольку пакет Spring `sessionProvider` содержит строку `autowire="byName"`, при создании экземпляра пакета в качестве синглтона для внедрения экземпляра класса `sessionFactory` вызывается метод `setSessionFactory()`. Чтобы не возвращаться к коду, приведенному ранее, снова приведем код ХРАНИЛИЩА, реализованного на основе каркаса Hibernate.

```
package com.saasovation.identityaccess.infrastructure.persistence;

public class HibernateUserRepository
       implements UserRepository {

    @Override
    public void add(User aUser) {
        try {
            this.session().saveOrUpdate(aUser);
```

```
        } catch (ConstraintViolationException e) {
            throw new IllegalStateException("User is not unique.", e);
        }
    }
    ...
    private SpringHibernateSessionProvider sessionProvider;

    public void setSessionProvider(
            SpringHibernateSessionProvider aSessionProvider) {
        this.sessionProvider = aSessionProvider;
    }

      private org.hibernate.Session session() {
        return this.sessionProvider.session();
    }
}
```

Этот фрагмент исходного кода взят из класса `HibernateUserRepository` в *Контексте идентификации и доступа*. Этот класс представляет собой пакет каркаса Spring, который автоматически связывается по имени, т.е. его метод `setSessionProvider()` автоматически вызывается при создании экземпляра, так что он получает ссылку на пакет `sessionProvider`, представляющий собой экземпляр класса `SpringHibernateSessionProvider`. При вызове метода `add()` (или любого другого метода, обеспечивающего постоянное хранение данных) он запрашивает объект класса `Session` с помощью своего метода `session()`. В свою очередь, метод `session()` использует внедренный пакет `sessionProvider` для получения экземпляра класса `Session`, связанного с потоком.

Несмотря на то что выше я продемонстрировал управление транзакциями с помощью каркаса Hibernate, все указанные принципы относятся и к другим механизмам постоянного хранения, таким как TopLink, JPA и др. Работая с любым из этих механизмов, вы должны найти способ обеспечить доступ к СЕАНСУ, ЕДИНИЦЕ РАБОТЫ и ТРАНЗАКЦИИ, которыми управляет ПРИКЛАДНОЙ УРОВЕНЬ. Если инъекция зависимости возможна, то она работает хорошо. Если же она недоступна, то используются другие способы связывания, вплоть до ручного связывания объектов с текущим потоком выполнения.

Предупреждение

Считаю необходимым предостеречь читателей от чрезмерного увлечения транзакциями в модели предметной области. АГРЕГАТЫ должны быть спроектированы в точном соответствии с границами согласованности. Не злоупотребляйте возможностью фиксации модификаций во многих АГРЕГАТАХ в рамках одной транзакции просто потому, что это возможно в тестовом модуле среды. Если вы не проявите осторожности, то, что хорошо работает на этапе разработки

и проектирования, может выйти из строя на этапе производства вследствие проблем с параллельным выполнением. При необходимости перечитайте главу, посвященную **АГРЕГАТАМ (10)**. В ней описано, как точно определить границы согласованности, чтобы обеспечить успешное выполнение транзакции

Иерархии типов

Если для разработки предметной модели используется объектно-ориентированный язык, может возникнуть соблазн применить наследование для создания иерархии типов. Мы могли бы сначала поддаться соблазну и сконцентрировать в базовом классе стандартное состояние и поведение, а затем расширить его на подклассы. Почему бы и нет? Это кажется идеальным способом избежать повторения.

Кроме того, для создания АГРЕГАТОВ, имеющих общую родословную, но отличающихся от остальных отдельным ХРАНИЛИЩЕМ, можно применить наследование к АГРЕГАТАМ, имеющих общую родословную и использующим одно и то же ХРАНИЛИЩЕ. В этом разделе мы не будем обсуждать ситуацию, в которой все типы АГРЕГАТОВ в одной и той же модели предметной области расширяют **УРОВЕНЬ СУПЕРТИПА (LAYER SUPERTYPE)** [Fowler, P of EAA], чтобы обеспечить состояние и/или поведение, общее для всей предметной области.[8]

Вместо этого рассмотрим создание относительно небольшого количества типов АГРЕГАТОВ, расширяющих общий проблемно-ориентированный суперкласс. Они разрабатываются, чтобы сформировать иерархию тесно связанных типов, у которых есть взаимозаменяемые, полиморфные характеристики. Эти виды иерархий используют единственное ХРАНИЛИЩЕ, чтобы сохранять и извлекать экземпляры отдельных типов, потому что клиенты должны использовать экземпляры взаимозаменяемо и клиенты крайне редко должны знать об определенном подклассе, с которым они имеют дело в любой момент времени, что соответствует **ПРИНЦИПУ ЗАМЕЩЕНИЯ ЛИСКОВ** (Liskov Substitution Principle — LSP) [Liskov].

Вот что я имею в виду. Допустим, что ваш бизнес использует внешние компании, чтобы обеспечить различные виды услуг, и вы должны моделировать отношения. Вы принимаете решение разработать общий абстрактный базовый класс `ServiceProvider`, но по некоторой серьезной причине вы должны разделить различные конкретные типы услуг, потому что услуги, которые предоставляет каждая компания, имеют универсальный характер, но в то же время отчетливо различаются. У вас могли бы быть классы `WarbleServiceProvider` и

[8] Преимущества уровня супертипа рассматриваются в главах, посвященных СУЩНОСТЯМ (5) и ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6).

WonkleServiceProvider. Вы разрабатываете эти типы так, чтобы иметь возможность запланировать запрос на обслуживание универсальным способом.

```
// клиент модели предметной области
serviceProviderRepository.providerOf(id)
      .scheduleService(date, description);
```

В этом контексте очевидно, что создание проблемно-ориентированных иерархий типов АГРЕГАТОВ, вероятно, ограничит их полезность во многих предметных областях. Разберемся в причинах. Как было показано ранее, в большинстве случаев общее ХРАНИЛИЩЕ будет содержать методы поиска, получающие экземпляры любого из подклассов. Это означает, что метод будет возвращать экземпляры общего суперкласса, в данном случае ServiceProvider, а не экземпляры определенных подклассов, WarbleServiceProvider и WonkleServiceProvider. Представьте себе, что произошло бы, если бы методы поиска возвращали экземпляры конкретных типов. Клиенты должны были бы знать, какие идентификаторы или другие дескриптивные атрибуты АГРЕГАТОВ относятся к экземплярам конкретного типа. Иначе это могло бы привести к неправильному поиску или исключению ClassCastException, которое возникает, когда возвращается экземпляр неправильного типа. Даже если бы вам удалось правильно разработать поиск экземпляров корректных типов, клиенты должны были бы также знать, какие подклассы могут выполнять конкретные операции, учитывая, что разработка АГРЕГАТОВ не могла полностью соответствовать принципу LSP.

При решении первой проблемы сегрегации по идентификаторам может показаться, что вы способны безопасно распознавать экземпляры, кодируя информацию о типе АГРЕГАТА в виде дискриминатора в классе уникального идентификатора. Это правда, но при этом возникают две дополнительные проблемы. Во-первых, клиент должен взять на себя ответственность за разрешение и отображение идентификаторов в типы. Во-вторых, возникает проблема связывания клиентов с разными операциями с помощью типа. Это приводит к следующим зависимостям клиента от типа.

```
// клиент модели предметной области

if (id.identifiesWarble()) {
    serviceProviderRepository.warbleOf(id)
       .scheduleWarbleService(date, warbleDescription);
} else if (id.identifiesWonkle()) {
    serviceProviderRepository.wonkleOf(id)
        .scheduleWonkleService(date, wonkleDescription);
} ...
```

Если этот вид взаимодействия становится нормой, а не исключением, это свидетельствует о плохом коде. К счастью, если преимущества, полученные от создания иерархии, достаточно велики, то редкое одноразовое использование такого взаимодействия может быть компромиссом, заслуживающим внимания. Однако в данном искусственном примере, вероятно, достаточно было бы применить более продуманный проект типа `ServiceDescription` и внутреннюю реализацию метода `schedule-Service()`. Иначе, я полагаю, стоило бы подумать, а не могли бы мы получить преимущества от использования наследования при присвоении каждому типу отдельного ХРАНИЛИЩА? Если требуется лишь несколько таких конкретных подклассов, может быть, лучше создать отдельные ХРАНИЛИЩА. Если же количество конкретных подклассов становится значительным, причем большинство из них может использоваться полностью взаимозаменяемо (по принципу LSP), то целесообразно использовать общее ХРАНИЛИЩЕ.

В большинстве случаев таких ситуаций можно полностью избежать, разрабатывая дескриптивную информацию о типе как свойство АГРЕГАТА (а не идентификатора). Просмотрите описание СТАНДАРТНЫХ ТИПОВ в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**. Таким образом, один тип АГРЕГАТА мог бы внутренне реализовать разное поведение на основе явно определенного СТАНДАРТНОГО ТИПА. Используя явный СТАНДАРТНЫЙ ТИП, мы могли иметь один конкретный АГРЕГАТ `ServiceProvider` и разработать его метод `schedule-Service()`, выполняющий диспетчеризацию на основе его типа. Для того чтобы экранировать клиентов от решений на основе типа, мы гарантируем, что информация о нем наружу не просочится. Вместо этого метод `scheduleService()` и другие методы класса `ServiceProvider` должным образом включают проблемно-ориентированные решения, продемонстрированные ниже.

```
public class ServiceProvider {
    private ServiceType type;
    ...
    public void scheduleService(
           Date aDate,
           ServiceDescription aDescription) {
       if (type.isWarble()) {
           this.scheduleWarbleService(aDate, aDescription);
       } else if (type.isWonkle()) {
           this.scheduleWonkleService(aDate, aDescription);
       } else {
           this.scheduleCommonService(aDate, aDescription);
       }
    }
    ...
}
```

Если внутренняя диспетчеризация становится запутанной, мы можем всегда разработать для нее меньшую иерархию. Фактически сам СТАНДАРТНЫЙ ТИП мог быть разработан как **СОСТОЯНИЕ (STATE)** [Гамма и др.], если вам нравится этот подход. В этом случае специализированное поведение реализовало бы различные типы. Это, конечно, также означало бы, что существует единственный класс `ServiceProviderRepository`, выполняющий сохранение разных типов в одном ХРАНИЛИЩЕ и использующий их с помощью общей функции.

В этой ситуации можно было бы также обойтись использованием ролевых интерфейсов. Мы могли бы разработать интерфейс `SchedulableService`, реализующий несколько типов АГРЕГАТА. Посмотрите дискуссию о ролях и обязанностях в главе, посвященной **СУЩНОСТЯМ (5)**. Даже если используется наследование, полиморфное поведение АГРЕГАТА чаще всего может быть тщательно разработано таким образом, что клиенты не сталкивались ни с какими особыми случаями.

Хранилище и объект доступа к данным

Иногда идею ХРАНИЛИЩА считают синонимичной ОБЪЕКТУ ДОСТУПА К ДАННЫМ (Data Access Object — DAO). Обе идеи описывают абстракцию механизма постоянного хранения данных. Это правда. Однако объектно-реляционный инструмент отображения также обеспечивает абстракцию механизма постоянного хранения данных, но это ни ХРАНИЛИЩЕ, ни ОБЪЕКТ ДОСТУПА К ДАННЫМ. Таким образом, мы не можем назвать именем DAO любую абстракцию механизма постоянного хранения данных. Мы должны скорее определить, реализуется ли образец. Вместо этого мы должны определить, реализован ли шаблон DAO.

Я думаю, что, в принципе, различия между ХРАНИЛИЩАМИ и DAO существуют. В основном объект DAO выражается в терминах таблиц базы данных, предоставляя им интерфейсы операций CRUD. Мартин Фаулер в работе [Фаулер, P EAA] отделяет использование подобных объектов DAO от тех, которые используются моделью предметной области. Он идентифицирует **ТАБЛИЧНЫЙ МОДУЛЬ (TABLE MODULE)**, **ТАБЛИЧНЫЙ ШЛЮЗ ДАННЫХ (TABLE DATA GATEWAY)** и **АКТИВНУЮ ЗАПИСЬ (ACTIVE RECORD)** как шаблоны, которые обычно используются в приложениях **СЦЕНАРИЯ ТРАНЗАКЦИИ (TRANSACTION SCRIPT)**. По этой причине объекты DAO и связанные с ними шаблоны, как правило, служат обертками вокруг таблиц базы данных. С другой стороны, ХРАНИЛИЩЕ и **ПРЕОБРАЗОВАТЕЛЬ ДАННЫХ (DATA MAPPER)**, имея объектное сродство, обычно используются моделью предметной области.

Так как DAO и связанные шаблоны можно использовать для выполнения мелкомодульных операций CRUD над данными, которые иначе считались бы

частями АГРЕГАТА, его не следует применять вместе с моделью предметной области. При нормальных условиях желательно, чтобы АГРЕГАТ сам управлял своей бизнес-логикой и другими внутренними компонентами и не допускал к ним всех остальных.

Выше указывалось, что иногда хранимая процедура или процессор записей grid-данных играет важную роль в удовлетворении некоторого нефункционального требования. В зависимости от предметной области это может быть скорее правилом, чем исключением. Однако, если такое системное нефункциональное требование не является жизненно необходимым, я рекомендую избегать его. Хранение и реализация бизнес-логики в хранилище данных во многом противоречат принципам DDD. Я пришел к заключению, что использование ФУНКЦИИ ФАБРИКИ ДАННЫХ/ПРОЦЕССОРА ЗАПИСЕЙ на самом деле не противоречит целям предметно-ориентированного моделирования. Реализация ФУНКЦИИ/ПРОЦЕССОРА ЗАПИСЕЙ может быть написана на языке Java, например, и должна соответствовать **ЕДИНОМУ ЯЗЫКУ (1)** и целям предметной области. Единственное отличие от базовой модели заключается в том, где выполняются ФУНКЦИЯ/ПРОЦЕССОР ЗАПИСЕЙ, которые не противоречат DDD. С другой стороны, излишнее использование хранимых процедур потенциально очень сильно противоречит принципам DDD, потому что язык программирования обычно плохо понимается командой моделирования, а реализации обычно "безопасно" отделены от их представления. Это полная противоположность тому, для чего предназначен подход DDD.

Можно представлять себе ХРАНИЛИЩЕ как ОБЪЕКТ ДОСТУПА К ДАННЫМ в общем смысле. Главное — помнить, что хранилища по возможности следует проектировать как ориентированные на имитацию коллекции, а не на доступ к данным. Это позволит сосредоточиться на модели предметной области, а не на данных и операциях CRUD, которые могут выполняться за кулисами для управления механизмом постоянного хранения данных.

Тестирование хранилищ

Существуют два подхода к тестированию хранилищ. Вы обязаны тестировать сами ХРАНИЛИЩА для того, чтобы убедиться, что они работают правильно. Кроме того, вы должны тестировать код, использующий ХРАНИЛИЩА для хранения АГРЕГАТОВ, которые ХРАНИЛИЩА создают и находят. Для тестирования первого вида необходимо проверять код окончательной программы. В противном случае вы не узнаете, работает ли ваша программа. При тестировании второго типа следует либо проверять окончательную программу, либо использовать реализации, размещенные в оперативной памяти. Я обсуждаю тестирование окончательной программы, а оперативные тесты мы обсудим немного позднее.

Рассмотрим тесты для реализации класса `ProductRepository` на основе системы Coherence.

```
public class CoherenceProductRepositoryTest extends DomainTest {

    private ProductRepository productRepository;
    private TenantId tenantId;

    public CoherenceProductRepositoryTest() {
        super();
    }
    ...
    @Override
    protected void setUp() throws Exception {
        this.setProductRepository(new CoherenceProductRepository());
        this.tenantId = new TenantId("01234567");
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        Collection<Product> products =
            this.productRepository()
                .allProductsOfTenant(tenantId);

        this.productRepository().removeAll(products);
    }

    protected ProductRepository productRepository() {
        return this.productRepository;
    }

    protected void setProductRepository(
        ProductRepository aProductRepository) {
           this.productRepository = aProductRepository;
    }
}
```

Существуют общие операции настройки тестов и очистки после каждого теста. Для того чтобы настроить тест, создадим экземпляр класса `CoherenceProductRepository`, а затем поддельный экземпляр класса `TenantId`.

Для очистки после тестирования мы удаляем все экземпляры класса `Product`, которые моли быть добавлены в соответствующий кеш при выполнении каждого теста. При работе с системой Coherence эта очистка очень важна. Если не удалить все кешированные экземпляры, то они будут существовать во время выполнения следующих тестов, что может привести к нарушению некоторых предположений, например переполнению счетчика сохраняемых экземпляров.

Перейдем к тестировании функций ХРАНИЛИЩА.

```
public class CoherenceProductRepositoryTest extends DomainTest {
    ...
    public void testSaveAndFindOneProduct() throws Exception {

        Product product =
            new Product(
                    tenantId,
                    this.productRepository().nextIdentity(),
                    "My Product",
                    "This is the description of my product.");

        this.productRepository().save(product);

        Product readProduct =
            this.productRepository()
                 .productOfId(tenantId, product.productId());

            assertNotNull(readProduct);
            assertEquals(readProduct.tenantId(), tenantId);
            assertEquals(readProduct.productId(), product.productId());
            assertEquals(readProduct.name(), product.name());
            assertEquals(readProduct.description(), product.description());
        }
        ...
}
```

Как утверждает имя тестового метода, мы сохраняем единственный экземпляр класса `Product` и пытаемся найти его. Первая задача состоит в том, чтобы создать экземпляр класса `Product`, а затем сохранить его в ХРАНИЛИЩЕ. Если инфраструктура не выдала никаких исключений, мы можем думать, что продукт был сохранен правильно. Однако есть только один способ убедиться в этом наверняка: найти этот экземпляр и сравнить его с оригиналом. Для того чтобы найти экземпляр, мы передаем его глобальный уникальный идентификатор методу `productOfId()`. Если экземпляр был найден, то можно утверждать, что раз операторы проверки его глобального идентификатора, атрибуты `tenantId`, `productId` и `name` возвращают значение, отличное от `null`, то его описание совпадает с описанием сохраненного экземпляра.

Перейдем к тестированию сохранения и поиска нескольких экземпляров.

```
public class CoherenceProductRepositoryTest extends DomainTest {
    ...
    public void testSaveAndFindMultipleProducts() throws Exception {

        Product product1 =
            new Product(
                    tenantId,
                    this.productRepository().nextIdentity(),
                    "My Product 1",
                    "This is the description of my first product.");
```

```
        Product product2 =
            new Product(
                    tenantId,
                    this.productRepository().nextIdentity(),
                    "My Product 2",
                    "This is the description of my second product.");

        Product product3 =
            new Product(
                     tenantId,
                     this.productRepository().nextIdentity(),
                     "My Product 3",
                     "This is the description of my third product.");

        this.productRepository()
            .saveAll(Arrays.asList(product1, product2, product3));

        assertNotNull(this.productRepository()
            .productOfId(tenant, product1.productId()));
        assertNotNull(this.productRepository()
            .productOfId(tenant, product2.productId()));
        assertNotNull(this.productRepository()
            .productOfId(tenant, product3.productId()));

        Collection<Product> allProducts =
             this.productRepository().allProductsOfTenant(tenant);

        assertEquals(allProducts.size(), 3);
    }
    ...
}
```

Сначала мы создаем три экземпляра класса `Product`, а затем сохраняем их все сразу с помощью метода `saveAll()`. Далее мы снова используем метод `productOfId()` для поиска индивидуальных экземпляров. Если все три экземпляра не равны `null`, мы убеждаемся, что они были правильно сохранены.

Ковбойская логика

AJ: Моя сестра рассказала мне, что ее муж попросил ее распродать все, что есть в его магазине, когда он умрет. Сестра спросила его, почему. Он ответил, что не хочет, чтобы после его смерти в его магазине хозяйничал какой-то болван, когда она повторно выйдет замуж. Сестра ответила ему, чтобы он не беспокоился, потому что она не собирается выходить замуж за другого болвана.

Остался один метод ХРАНИЛИЩА `allProductsOfTenant()`, который мы еще не тестировали. Если кеш ХРАНИЛИЩА во время начала тестирования был совершенно пустым, мы должны были бы успешно извлечь из ХРАНИЛИЩА три экземпляра класса `Product`. Попытаемся их найти. Возвращаемый объект класса

Collection никогда не должен быть равным null, даже если вы не нашли то, что искали. По этой причине последний этап тестирования заключается в проверке полного количества ожидаемых экземпляров класса Product. Оно должно быть равно трем.

Теперь у нас есть тест, который демонстрирует, что клиенты могут использовать ХРАНИЛИЩЕ, и делают это правильно, поэтому мы можем перейти к вопросу оптимизации тестирования клиентов, использующих ХРАНИЛИЩА.

Тестирование реализаций в оперативной памяти

Очень трудно настроить полноценное тестирование ХРАНИЛИЩА как механизма постоянного хранения данных. Кроме того, такой тест работает очень медленно, поэтому следует попытаться применить другой подход. Вы можете столкнуться с непредвиденными обстоятельствами уже на ранних стадиях моделирования предметной области. Это может произойти потому, например, что ваши механизмы постоянного хранения, включая схему базы данных, пока недоступны. Когда вы оказываетесь в таких ситуациях, лучше всего создать вариант ХРАНИЛИЩА в оперативной памяти.

Создание варианта ХРАНИЛИЩА в оперативной памяти может быть довольно простым, но может и вызывать определенные проблемы. Достаточно просто создать экземпляр класса HashMap для вашего интерфейса. Выполнить методы put() для внесения записей и remove() для их удаления из экземпляра класса Map нетрудно. Мы просто используем глобальный уникальный идентификатор каждого экземпляра АГРЕГАТА в качестве ключа. Значением служит сам экземпляр АГРЕГАТА. Методы add(), save() и remove() довольно тривиальные. Полная реализация класса ProductRepository выглядит очень просто.

```
package com.saasovation.agilepm.domain.model.product.impl;

public class InMemoryProductRepository implements ProductRepository {

    private Map<ProductId,Product> store;

    public InMemoryProductRepository() {
       super();
       this.store = new HashMap<ProductId,Product>();
    }

    @Override
    public Collection<Product> allProductsOfTenant(Tenant aTenant) {
        Set<Product> entries = new HashSet<Product>();

        for (Product product : this.store.values()) {
            if (product.tenant().equals(aTenant)) {
                entries.add(product);
```

```
            }
        }

        return entries;
    }

    @Override
    public ProductId nextIdentity() {
        return new ProductId(java.util.UUID.randomUUID()
                    .toString().toUpperCase());
    }

    @Override
    public Product productOfId(Tenant aTenant, ProductId aProductId) {
        Product product = this.store.get(aProductId);

        if (product != null) {
            if (!product.tenant().equals(aTenant)) {
                product = null;
            }
        }

        return product;
    }

    @Override
    public void remove(Product aProduct) {
        this.store.remove(aProduct.productId());
    }

    @Override
        public void removeAll(Collection<Product> aProductCollection) {
            for (Product product : aProductCollection) {
                this.remove(product);
            }
        }

    @Override
    public void save(Product aProduct) {
        this.store.put(aProduct.productId(), aProduct);
    }

    @Override
    public void saveAll(Collection<Product> aProductCollection) {
        for (Product product : aProductCollection) {
             this.save(product);
        }
    }
}
```

На самом деле у метода `productOfId()` существует только один частный случай. Для правильной реализации этого метода поиска после сравнения экземпляра класса `Product` с заданным экземпляром класса `ProductId` мы должны

также убедиться, что атрибут TenantId в классе Product совпадает с параметром типа Tenant. Если это не так, экземпляру класса Product присваивается значение null.

Мы можем создать почти идентичную копию класса CoherenceProductRepositoryTest с именем InMemoryProductRepositoryTest для проверки в оперативной памяти. Для этого необходимо внести только одно изменение в метод setUp().

```
public class InMemoryProductRepositoryTest extends TestCase {
    ...
    @Override
    protected void setUp() throws Exception {
        this.setProductRepository(new InMemoryProductRepository());
        this.tenantId = new TenantId("01234567");

        super.setUp();
    }
    ...
}
```

Просто создайте экземпляр класса InMemoryProductRepository, а не реализацию, основанную на каркасе Coherence. В остальном тестовые методы совершенно идентичны.

Намного более сложные проблемы связаны с более изощренными методами поиска, в которых труднее преобразовать параметры, описывающие критерий. Если критерии и логика преобразования их параметров становятся слишком сложными, необходимо найти выход из этой ситуации. Возможно, для этого придется заранее заполнить ХРАНИЛИЩЕ экземплярами, соответствующими критериям поиска, а сам метод заставить возвращать только эти экземпляры. Заполнить хранилище можно с помощью тестового метода setUp().

Другое преимущество реализаций ХРАНИЛИЩА в оперативной памяти проявляется в ситуации, когда необходимо проверить работу метода save() с интерфейсом, ориентированным на механизм постоянного хранения. Например, можно сделать так, чтобы метод save() подсчитывал количество вызовов. После запуска каждого теста можно проверить, не превысило ли количество вызовов заранее заданный порог, установленный клиентом данного ХРАНИЛИЩА. Этот подход можно использовать и для тестирования ПРИКЛАДНЫХ СЛУЖБ, которые обязаны явным образом выполнять метод save(), для сохранения изменений в АГРЕГАТЕ.

Резюме

В этой главе мы рассмотрели реализацию ХРАНИЛИЩ.

- Вы изучили ХРАНИЛИЩА, ориентированные на коллекции и на механизмы постоянного хранения, и узнали, почему используется тот или иной вид ХРАНИЛИЩ.
- Вы увидели реализацию ХРАНИЛИЩ с использованием систем Hibernate, TopLink, Coherence и MongoDB.
- Вы узнали, почему в интерфейсе ХРАНИЛИЩА может понадобиться дополнительная поведенческая функция.
- Вы рассмотрели роль транзакций в использовании ХРАНИЛИЩ.
- Вы ознакомились с проблемами, связанными с проектированием ХРАНИЛИЩ для иерархий типов.
- Вы увидели фундаментальные различия между ХРАНИЛИЩАМИ и ОБЪЕКТАМИ ДОСТУПА К ДАННЫМ.
- Вы узнали, как проводить тестирование ХРАНИЛИЩ, и освоили способы проверки их эксплуатации.

Теперь соберемся с духом и приступим к изучению интеграции ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Глава 13

Интеграция ограниченных контекстов

Установление ментальных связей —
самый важный инструмент обучения,
суть интеллекта человека;
он позволяет налаживать отношения;
выходить за установленные рамки;
видеть шаблоны, отношения и контекст.

Мэрилин Фергюсон

В любом значительном проекте всегда есть несколько **ОГРАНИЧЕННЫХ КОНТЕКСТОВ (2)**, причем некоторые из них необходимо интегрировать. С помощью **КАРТ КОНТЕКСТОВ (3)** мы обсудили отношения, которые обычно существуют между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, и рассмотрели некоторые способы, которыми этими отношениями можно правильно управлять согласно принципам DDD. Если вы недостаточно хорошо изучили **ПРЕДМЕТНЫЕ ОБЛАСТИ (2)**, **ПОДОБЛАСТИ (2)**, ОГРАНИЧЕННЫЕ КОНТЕКСТЫ или КАРТЫ КОНТЕКСТОВ, вам следует вернуться к этим темам, поскольку материал, представленный здесь, основывается на этих фундаментальных понятиях.

Как говорилось ранее, существуют две основные формы КАРТ КОНТЕКСТОВ. Одна форма — простая графическая схема, которая используется для иллюстрации видов отношений, существующих между любыми двумя или более ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Вторая и намного более конкретная форма — код, в котором фактически реализованы эти отношения. Именно их мы и рассмотрим в этой главе.

Назначение главы

- Рассмотреть основы интеграции и выработать соответствующие принципы интеграции систем в среде распределенных вычислений.
- Описать подход к интеграции многочисленных КОНТЕКСТОВ на основе ресурсов RESTful и обсудить его преимущества и недостатки.
- Научиться интегрировать системы с помощью сообщений.
- Изучить проблемы, с которыми связано дублирование информации в разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ, научиться решать или избегать их.
- Исследовать примеры, обогащающие опыт проектирования.

Основы интеграции

Есть несколько очевидных способов интеграции двух ОГРАНИЧЕННЫХ КОНТЕКСТОВ с помощью кода.

Один из таких прямых подходов состоит в том, что ОГРАНИЧЕННЫЙ КОНТЕКСТ открывает свой прикладной программный интерфейс (API), а другой ОГРАНИЧЕННЫЙ КОНТЕКСТ использует этот API через дистанционный вызов процедур (RPC). Интерфейс API можно сделать доступным, используя протокол SOAP или просто отправляя XML-запросы и получая ответы по протоколу HTTP (не путать с REST). Фактически существует несколько способов создать удаленно доступный интерфейс API. Это один из наиболее популярных способов интеграции. Поскольку этот подход поддерживает стиль, основанный на вызовах процедур, он понятен программистам, привыкшим вызывать процедуры и методы, как и большинство из нас.

Второй прямой способ интеграции ОГРАНИЧЕННЫХ КОНТЕКСТОВ основан на использовании механизма обмена сообщениями. Каждая из взаимодействующих систем посылает и принимает сообщения с помощью очереди сообщений или механизма **ИЗДАТЕЛЬ–ПОДПИСЧИК** [Гамма и др.]. Конечно, шлюзы обмена сообщениями вполне можно считать интерфейсами API, но мы можем расширить их сферу применения, если будем просто называть их интерфейсами служб. Есть большое количество методов интеграции, основанных на использовании механизма обмена сообщениями. Многие из них рассмотрены в работе [Hohpe & Woolf].

Третий способ интеграции ОГРАНИЧЕННЫХ КОНТЕКСТОВ основан на использовании протокола RESTful HTTP. Иногда его считают разновидностью подхода RPC (Remote Procedure Calls), но на самом деле это не так. У него есть похожие свойства, потому что одна система выполняет запрос в другую систему, но эти запросы не выполняются с помощью процедур, которым передаются параметры. Как

указано в главе, посвященной **АРХИТЕКТУРЕ (4)**, REST — это средство обмена и модификации ресурсов, которые однозначно определяются с помощью уникального идентификатора URI. На каждом ресурсе могут выполняться разные операции. Выполнение методов, прежде всего GET, PUT, POST и DELETE, обеспечивает протокол RESTful HTTP. Хотя может показаться, что он поддерживает только операции CRUD, достаточно небольшого воображения, чтобы отнести операции, имеющие явно выраженную цель, к одной из четырех категорий методов. Например, метод GET может использоваться для дифференцирования различных видов операций запроса по категориям, а метод PUT может использоваться для инкапсуляции командной операции, которая выполняется на **АГРЕГАТЕ (10)**.

Разумеется, это не значит, что существует всего три способа интеграции приложений. Можно, например, использовать интеграцию с помощью файлов или общей базы данных, но из-за этих методов вы можете раньше времени почувствовать себя старым.

Ковбойская логика

AJ: Лучше опустите седло пониже. Этот конь строптивый и может заставить вас почувствовать себя старым раньше времени.

Несмотря на то что я выделил три распространенных способа, которые используются для интеграции ОГРАНИЧЕННЫХ КОНТЕКСТОВ, в этой главе мы будем фактически придерживаться только двух из них. Мы сосредоточимся главным образом на интеграции с помощью механизмов обмена сообщениями, а также покажем, как использовать протокол RESTful HTTP. Мы не станем рассматривать примеры использования RPC, потому что вы можете легко создать интерфейс API, который мог бы заменить другие два подхода. Кроме того, подход RPC не обладает требуемой гибкостью, в то время как наша цель состоит в том, чтобы поддерживать автономные службы (или автономные приложения). Отказавшая система, обеспечивающая обычный интерфейс API, основанный на подходе RPC, помешала бы зависимым системам успешно выполнить свои операции.

Это подводит нас к теме огромной важности, которая требует внимания каждого разработчика, выполняющего интеграцию.

Распределенные системы совершенно другие

Если разработчики, не знакомые с принципами работы распределенных систем, игнорируют свойственную им сложность, то неизбежно возникают проблемы. Это особенно относится к использованию RPC, потому что неопытные

разработчики, плохо знающие работу распределенных систем, обычно предполагают, что любой дистанционный вызов процедуры ничем не отличается от вызова локальной процедуры. Такие предположения могут вызвать каскадный отказ любого количества систем, когда всего одна система или один из ее компонентов становится недоступным, даже временно. Таким образом, все разработчики, работающие с распределенными системами, должны следовать приведенным ниже ПРИНЦИПАМ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ, иначе их ждет крах.

- Сеть не надежна.
- Всегда существует задержка при выполнении операций, причем, иногда, значительная.
- Полоса пропускания не бесконечна.
- Не следует считать, что сеть безопасна.
- Топология сети изменяется.
- Информация и политики контролируются многими администраторами.
- Передача данных по сети связана с расходами.
- Сеть неоднородна.

Эти принципы специально сформулированы отдельно от принципов, приведенных в работе "Fallacies of Distributed Computing" [Deutsch]. Я называю их принципами, чтобы подчеркнуть проблемы, которые необходимо решить, и сложность, которую следует учитывать, чтобы не делать глупых ошибок.

Обмен информацией через границы систем

В большинстве случаев, когда мы нуждаемся во внешней системе, предоставляющей услугу для нашей собственной системы, мы должны передавать соответствующей службе некую информацию. Используемые нами службы должны возвращать ответы. Таким образом, мы нуждаемся в надежном способе передачи данных между системами. Эти данные должны передаваться между разными системами по структуре, которую могут использовать все участники. Большинство из нас приняло бы решение использовать для этого некоторый стандартный способ.

Данные, которые отсылаются как параметры или сообщения, образуют всего лишь машиночитаемые структуры, которые могут быть сгенерированы в одном из многих форматов. Мы должны также создать некую форму контракта между системами, обменивающимися данными, и, возможно, даже механизмы для лексического разбора или интерпретации этих структур, чтобы их можно было обрабатывать.

Есть несколько способов генерировать структуры, используемые для обмена информацией между системами. В одной из технических реализаций просто используются средства языка программирования для сериализации объектов в двоичный формат и десериализации их на стороне потребителя. Этот подход оправдывает себя, если все системы поддерживают одни и те же языковые средства, а сериализация фактически является совместимой или взаимозаменяемой в разных аппаратных архитектурах. Для этого также необходимо в каждой системе, использующей конкретный тип объектов, поддерживать все интерфейсы и классы объектов, которые используются всеми системами.

Другой подход к созданию взаимозаменяемых информационных структур состоит в использовании стандартного промежуточного формата. Например, можно использовать форматы XML, JSON или специализированный формат, такой как Protocol Buffers. У каждого из этих подходов есть преимущества и недостатки, к которым, в частности, относятся выразительность и лаконичность, скорость преобразований типов, поддержка гибкости между версиями объектов и простота использования. Некоторые из них могут оказаться дорогостоящими с учетом ПРИНЦИПОВ РАСПРЕДЕЛЕННЫХ ВЫЧИСЛЕНИЙ, приведенных выше (например, "Передача данных по сети связана с расходами").

При использовании промежуточного формата у вас все еще может появиться желание поддерживать все интерфейсы и классы объектов, которые используются разными системами, и поместить данные, представленные в промежуточном формате, в безопасные с точки зрения типов объекты. Благодаря этому вы можете использовать объекты в системе-получателе точно так же, как и в системе-отправителе.

Разумеется, поддержка этих интерфейсов и классов также представляет определенную сложность, и это обычно означает, что система-получатель должна быть перекомпилирована, чтобы обеспечить совместимость с последними версиями определений классов и интерфейсов. Есть также опасность свободно использовать внешние объекты в системе-получателе так, будто они являются внутренними. Это противоречит стратегическим принципам проектирования DDD, которые мы так упорно отстаивали. Некоторые читатели могут подумать, что можно подстраховаться, объявив этот подход **ОБЩИМ ЯДРОМ (3)**. Однако знайте, что удобство совместного использования объектов в разных системах может привести вас на скользкую дорожку. И все же, независимо от сложности и потенциальной опасности таких искаженных моделей, многие полагают, что строгий контроль типов, предоставляемый этой тактикой, является подходящим компромиссом с точки зрения требуемой сложности.

Впрочем, я часто встречаю людей, которые возражают против этого по разным причинам, и они выражают желание применять более простой и безопасный

подход, но не могут совсем игнорировать безопасность типов. Давайте рассмотрим этот подход.

Что было бы, если бы мы могли так определить контракт между системами, создающими взаимозаменяемые информационные структуры с одной стороны и получающими их с другой, чтобы потребители могли уверенно использовать данные, не выполняя их десериализацию в объектные экземпляры конкретных классов? Мы можем определить такой надежный контракт, используя подход, основанный на стандартах, который фактически формирует **ЕДИНЫЙ ЯЗЫК (3)**. Один из таких стандартных подходов должен определить пользовательский тип носителя информации или его семантический эквивалент. Независимо от того, есть ли у вас серьезное основание регистрировать такой тип носителя информации, как описано в RFC 4288, это реальная спецификация, которую необходимо учитывать. Спецификация определяет обязывающий договор между производителями информации и ее получателями, и предлагает надежное средство для обмена, без совместного использования двоичных файлов классов и интерфейсов.

Как всегда, этот подход требует принятия определенных компромиссов. Вы не будете в состоянии перемещаться по объектам с помощью методов доступа к свойствам, как могли бы, если бы у вас были интерфейсы/классы для каждого объекта и была обеспечена безопасность соответствующих типов. Вы также испытали бы недостаток поддержки интегрированной среды разработки, например автодополнения кода. Это не такой уж большой недостаток. Далее, у вас не было бы функциональной поддержки функций/методов, которую могло бы предоставить наличие класса СОБЫТИЙ. Однако я считаю отсутствие операционных функций/методов для СОБЫТИЙ не недостатком, а скорее защитой. Получающий ОГРАНИЧЕННЫЙ КОНТЕКСТ должен интересоваться только свойствами данных и никогда не должен пытаться использовать функциональные возможности, являющиеся частью другой модели. **ПОРТЫ И АДАПТЕРЫ (4)** получающей системы должны экранировать ее модель предметной области от любых таких зависимостей и вместо этого должны передавать необходимые данные о СОБЫТИИ как соответствующие параметры, типы которых определены только в ее собственном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Все необходимые вычисления (или обработка) должны быть выполнены в создающем ОГРАНИЧЕННОМ КОНТЕКСТЕ и представлены как дополнение атрибутов данных о СОБЫТИИ.

Рассмотрим пример. Компания SaaSOvation желает разработать механизм обмена информацией между разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Для этого она будет использовать ресурсы RESTful, а также службы, обменивающиеся сообщениями, содержащими **СОБЫТИЯ (8)**. Один из ресурсов RESTful называется *уведомлением*, поэтому сообщения, содержащие СОБЫТИЯ, посылаются подписчикам как объекты класса `Notification`. Иначе говоря, в обоих случаях объект

класса Notification содержит СОБЫТИЕ, которое форматируется в виде отдельной структуры. Пользовательская спецификация носителя для уведомлений и СОБЫТИЙ должна содержать информацию о контракте, который содержит следующие пункты.

- Тип: Notification, формат: JSON
- notificationId: уникальный идентификатор в виде длинного целого числа
- typeName: тип текстового уведомления String, пример имени com.saasovation.agilepm.domain.model.product.backlogItem.BacklogItemCommitted
- version: версия уведомления в виде целого числа
- occurredOn: дата/время, когда произошло СОБЫТИЕ, содержащееся в уведомлении
- event: детали формата JSON; см. конкретные типы СОБЫТИЯ

Использование полного имени класса typeName (включая имя пакета) позволяет подписчикам точно дифференцировать разные типы Notification. Спецификация уведомления обязательна для всех спецификаций СОБЫТИЙ. Для примера рассмотрим СОБЫТИЕ с именем BacklogItemCommitted.

- Тип СОБЫТИЯ: com.saasovation.agilepm.domain.model.product.backlogItem.BacklogItemCommitted
- eventVersion: целочисленный номер версии СОБЫТИЯ, соответствующий атрибуту Notificationversion
- occurredOn: дата/время, когда произошло СОБЫТИЕ, соответствующее атрибуту NotificationoccurredOn
- backlogItemId: BacklogItemId, содержит текстовый атрибут идентификатора
- committedToSprintId: SprintId, содержит текстовый атрибут идентификатора
- tenantId: TenantId, содержит текстовый атрибут идентификатора
- Детали СОБЫТИЯ: см. конкретные типы СОБЫТИЯ

Разумеется, мы могли бы привести детали СОБЫТИЯ для каждого типа СОБЫТИЯ. Для класса Notification и всех указанных типов СОБЫТИЙ мы можем безопасно использовать объект класса NotificationReader, продемонстрированный в следующем тесте.

```
DomainEvent domainEvent = new TestableDomainEvent(100, "testing");

Notification notification = new Notification(1, domainEvent);

NotificationSerializer serializer =
    NotificationSerializer.instance();

String serializedNotification = serializer.serialize(notification);

NotificationReader reader =
   new NotificationReader(serializedNotification);

assertEquals(1L, reader.notificationId());
assertEquals("1", reader.notificationIdAsString());
assertEquals(domainEvent.occurredOn(), reader.occurredOn());
assertEquals(notification.typeName(), reader.typeName());
assertEquals(notification.version(), reader.version());
assertEquals(domainEvent.eventVersion(), reader.version());
```

Этот тест показывает, как объект класса `NotificationReader` может предоставить стандартные и безопасные с точки зрения типов компоненты для каждого сериализованного объекта класса `Notification`.

Следующий тест демонстрирует, как эти специальные части каждой детали СОБЫТИЯ можно прочитать из объекта класса `Notification`. Навигация по объектам СОБЫТИЯ осуществляется с помощью синтаксиса, напоминающего язык XPath или свойства, разделяемые точками. В качестве альтернативы можно использовать имена, разделенные запятыми (аргументы переменной длины из языка Java). Каждый из этих атрибутов можно прочитать как значение типа `String` или одного из элементарных типов (`int`, `long`, `boolean`, `double` и т.д.).

```
TestableNavigableDomainEvent domainEvent =
    new TestableNavigableDomainEvent(100, "testing");

Notification notification = new Notification(1, domainEvent);

NotificationSerializer serializer = NotificationSerializer.instance();

String serializedNotification = serializer.serialize(notification);

NotificationReader reader =
    new NotificationReader(serializedNotification);

assertEquals("" + domainEvent.eventVersion(),
 reader.eventStringValue("eventVersion"));
assertEquals("" + domainEvent.eventVersion(),
    reader.eventStringValue("/eventVersion"));
assertEquals(domainEvent.eventVersion(),
    reader.eventIntegerValue("eventVersion").intValue());
assertEquals(domainEvent.eventVersion(),
    reader.eventIntegerValue("/eventVersion").intValue());
```

```
assertEquals("" + domainEvent.nestedEvent().eventVersion(),
    reader.eventStringValue("nestedEvent", "eventVersion"));
assertEquals("" + domainEvent.nestedEvent().eventVersion(),
    reader.eventStringValue("/nestedEvent/eventVersion"));
assertEquals(domainEvent.nestedEvent().eventVersion(),
    reader.eventIntegerValue("nestedEvent", "eventVersion").intValue());
assertEquals(domainEvent.nestedEvent().eventVersion(),
    reader.eventIntegerValue("/nestedEvent/eventVersion").intValue());

assertEquals("" + domainEvent.nestedEvent().id(),
    reader.eventStringValue("nestedEvent", "id"));
assertEquals("" + domainEvent.nestedEvent().id(),
    reader.eventStringValue("/nestedEvent/id"));
assertEquals(domainEvent.nestedEvent().id(),
    reader.eventLongValue("nestedEvent", "id").longValue());
assertEquals(domainEvent.nestedEvent().id(),
    reader.eventLongValue("/nestedEvent/id").longValue());

assertEquals("" + domainEvent.nestedEvent().name(),
    reader.eventStringValue("nestedEvent", "name"));
assertEquals("" + domainEvent.nestedEvent().name(),
    reader.eventStringValue("/nestedEvent/name"));

assertEquals("" + domainEvent.nestedEvent().occurredOn().getTime(),
    reader.eventStringValue("nestedEvent", "occurredOn"));
assertEquals("" + domainEvent.nestedEvent().occurredOn().getTime(),
    reader.eventStringValue("/nestedEvent/occurredOn"));
assertEquals(domainEvent.nestedEvent().occurredOn(),
    reader.eventDateValue("nestedEvent", "occurredOn"));
assertEquals(domainEvent.nestedEvent().occurredOn(),
    reader.eventDateValue("/nestedEvent/occurredOn"));
assertEquals("" + domainEvent.occurredOn().getTime(),
    reader.eventStringValue("occurredOn"));
assertEquals("" + domainEvent.occurredOn().getTime(),
    reader.eventStringValue("/occurredOn"));
assertEquals(domainEvent.occurredOn(),
    reader.eventDateValue("occurredOn"));
assertEquals(domainEvent.occurredOn(),
    reader.eventDateValue("/occurredOn"));
```

Объект класса `TestableNavigableDomainEvent` содержит объект класса `TestableDomainEvent`, позволяющий тестировать навигацию к более глубоко расположенным атрибутам. Атрибуты считываются с помощью синтаксиса, похожего на язык XPath. Кроме того, осуществляется проверка чтения значений атрибута, имеющих разные типы.

Поскольку экземпляры класса `Notification` и СОБЫТИЯ всегда имеют номер версии, существует возможность использовать это для чтения специальных атрибутов конкретной версии. Потребители, настроенные на определенную версию, могут при необходимости найти соответствующие части. Однако, помимо этого, потребители могут получить любой экземпляр класса `Notification`, содержащий СОБЫТИЕ, так, будто номер его версии равен единице.

Таким образом, если мы тщательно учтем, как спроектирован каждый тип СОБЫТИЯ, то сможем защитить большинство потребителей от несовместимости, если все они ждут первую версию данного СОБЫТИЯ. Такие потребители никогда не должны изменяться или перекомпилироваться при изменении СОБЫТИЯ. Однако вам все равно необходимо думать о совместимости версий и правильно планировать переход к новым версиям, чтобы не навредить большинству потребителей. Иногда этого невозможно достигнуть, но во многих случаях это вполне реально.

У этого подхода есть одно дополнительное преимущество: СОБЫТИЯ могут содержать больше информации, чем обычные элементарные атрибуты и строки. События могут также безопасно содержать экземпляры более сложных **ОБЪЕКТОВ-ЗНАЧЕНИЙ (6)**, что особенно важно, когда типы ЗНАЧЕНИЙ остаются постоянными. Конечно, это относится и к классам `BacklogItemId`, `SprintId` и `TenantId`, как показано в следующем коде, в котором используется свойство навигации с разделяющими точками.

```
NotificationReader reader =
    new NotificationReader(backlogItemCommittedNotification);

String backlogItemId = reader.eventStringValue("backlogItemId.id"));

String sprintId = reader.eventStringValue("sprintId.id"));

String tenantId = reader.eventStringValue("tenantId.id"));
```

Тот факт, что любые сохраненные экземпляры ЗНАЧЕНИЙ встроены в структуру, позволяет СОБЫТИЯМ быть не только неизменными, но и зафиксированными навечно. Новые версии типов ОБЪЕКТОВ-ЗНАЧЕНИЙ, содержавших СОБЫТИЯ, не повлияют на возможность прочитать более старые версии этих ЗНАЧЕНИЙ из существовавших ранее экземпляров класса `Notification`. Конечно, если версии СОБЫТИЙ изменяются значительно и часто и контакт с этими изменениями становится неудобным для потребителей, использующих объект класса `NotificationReader`, язык Protocol Buffers намного упростил бы работу.

Помните, что все это — лишь возможность упростить десериализацию без поддержки типов СОБЫТИЯ и зависимостей. Одни считают этот стиль элегантным и свободным, а другие — рискованным, неприемлемым или даже опасным. Широко распространен противоположный подход, основанный на поддержке интерфейсов там, где используются сериализованные объекты. Здесь я привел лишь информацию к размышлению.

Ковбойская логика

LB: Ты знаешь, Джей, когда ковбой становится слишком старым, чтобы показывать плохие примеры, он начинает давать хорошие советы.

Вполне возможно, что каждый подход — поддержка классов для обмена сериализованными объектами и определение контракта для типа носителя — имеет свои преимущества на различных этапах проекта. Например, в зависимости от количества команд разработчиков, ОГРАНИЧЕННЫХ КОНТЕКСТОВ, интенсивности изменений и других факторов, можно было бы совместно использовать классы и интерфейсы во время запуска проекта, а на производственном этапе использовать более изолированный, пользовательский контракт для типа носителя. На практике все зависит от количества и состава команд разработчиков. Иногда команда не в состоянии отказаться от того подхода, который она выбрала с самого начала, и не торопится развернуться на 180 градусов.

Для того чтобы сделать наши рабочие примеры простыми и понятными, в оставшейся части главы я повсюду использую класс `NotificationReader`. Использовать ли пользовательский контракт для типа носителя или класс `NotificationReader` в ваших ОГРАНИЧЕННЫХ КОНТЕКСТАХ — дело ваше.

Интеграция с помощью ресурсов RESTful

Схема, в которой ОГРАНИЧЕННЫЙ КОНТЕКСТ предоставляет широкий набор ресурсов RESTful с помощью идентификаторов URI, является **СЛУЖБОЙ С ОТКРЫТЫМ ПРОТОКОЛОМ (3)**.

> Определите протокол, который предоставил бы доступ к вашей подсистеме как к набору СЛУЖБ (SERVICES). Сделайте протокол открытым, чтобы все желающие интегрироваться смогли бы им воспользоваться. Совершенствуйте и расширяйте протокол для учета новых интеграционных требований, кроме тех случаев, когда у отдельных групп возникают особо экзотические запросы [Эванс, с. 327].

Методы HTTP — `GET`, `PUT`, `POST` и `DELETE` — в сочетании с ресурсами, которыми они оперируют, можно легко представить как открытые службы. Протокол HTTP и архитектура REST очевидным образом формируют открытый протокол, позволяющий всем интегрироваться с подсистемой. Возможность создавать практически неограниченное количество ресурсов, каждый из которых имеет

уникальный идентификатор, представленный в виде URI, позволяет протоколу удовлетворять потребности в интеграции по мере необходимости. Это универсальный способ, позволяющий клиентам интегрироваться с вашим ОГРАНИЧЕННЫМ КОНТЕКСТОМ.

Впрочем, поскольку поставщик службы RESTful должен непосредственно взаимодействовать с ресурсом, которым он управляет, этот стиль не разрешает клиентам быть абсолютно автономными. Если ОГРАНИЧЕННЫЙ КОНТЕКСТ, основанный на архитектуре REST, по каким-то причинам станет недоступным, зависимые клиентские ОГРАНИЧЕННЫЕ КОНТЕКСТЫ будут не способны выполнять необходимые операции интеграции на протяжении всего времени простоя.

Тем не менее мы можем в некоторой степени преодолеть это ограничение, уменьшая зависимость от ресурсов RESTful. Даже если стиль RESTful (или RPC) — ваше единственное средство интеграции, вы можете создать иллюзию временного разъединения с помощью таймеров или обмена сообщениями в вашей собственной системе. Таким образом, ваша система обратится к любой удаленной системе, только когда сработает таймер или будет получено сообщение. Если удаленная система недоступна, то порог таймера может быть сброшен, а если используется обмен сообщениями, то брокер может получить сообщение об отказе и выполнить повторную отправку. Естественно, чтобы сделать системы слабо связанными, вашей команде придется затратить больше времени, но это цена, которую, вероятно, придется заплатить за обеспечение автономии.

Когда команде SaaSOvation, разрабатывающей *Контекст Идентификации и Доступа*, потребовалось предоставить интеграторам способ использования их ОГРАНИЧЕННОГО КОНТЕКСТА, они решили, что протокол RESTful HTTP будет одним из лучших способов открыть их систему для интеграции без непосредственного раскрытия структурных и поведенческих деталей их модели предметной области. Для них это означало разработку ряда ресурсов RESTful, которые должны были обеспечить представление концепций идентификации и доступа для каждого арендатора.

Такой проект должен был обеспечить интеграцию ОГРАНИЧЕННЫХ КОНТЕКСТОВ для получения ресурсов с помощью операции GET. Эти ресурсы должны содержать идентификатор пользователя и группы, а также информацию о роли и полномочиях доступа для идентификаторов указанных типов. Например, если клиенту интеграции необходимо знать, может ли пользователь указанного арендатора играть конкретную роль доступа, этот клиент должен выполнить операцию GET и получить ресурс с помощью следующего формата идентификатора URI.

```
/tenants/{tenantId}/users/{username}/inRole/{role}
```

Если пользователь арендатора играет указанную роль, то представление ресурса включается в ответ с кодом успеха 200. Если же такого пользователя нет или он не может играть указанную роль, то ответ содержит код 204, означающий отсутствие требуемого содержимого. Это простая схема протокола RESTful HTTP.

Посмотрим, как команда откроет ресурсы доступа и как клиенты интеграции смогут использовать их в терминах **ЕДИНОГО ЯЗЫКА (1)** в своем ОГРАНИЧЕННОМ КОНТЕКСТЕ.

Реализация ресурса RESTful

Решив придерживаться принципов архитектуры REST в одном из своих ОГРАНИЧЕННЫХ КОНТЕКСТОВ, команда SaaSOvation усвоила несколько важных уроков. Посмотрим, как им это удалось.

Работая над *Контекстом Идентификации и Доступа*, команда SaaSOvation размышляла о том, как предоставить интеграторам СЛУЖБУ С ОТКРЫТЫМ ПРОТОКОЛОМ. Разработчики рассматривали возможность просто представить их модель предметной области в виде ряда связанных ресурсов RESTful. Это означало бы позволить клиентам HTTP получать уникальный ресурс арендатора с помощью операции GET и перебирать пользователей, группы и роли. Было ли это хорошей идеей? Сначала это казалось естественным. В конце концов, это обеспечивало бы клиентам максимальную гибкость. Клиенты могли знать все о модели предметной области и просто принимать решения в собственном ОГРАНИЧЕННОМ КОНТЕКСТЕ.

Какой шаблон КАРТЫ КОНТЕКСТА DDD лучше всего описывает этот подход к проектированию? На самом деле это не СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ. В зависимости от размера совместно используемой модели можно было бы применить шаблон ОБЩЕЕ ЯДРО или **КОНФОРМИСТ (CONFORMIST) (3)**. Публикация ОБЩЕГО ЯДРА или установление отношения КОНФОРМИСТ привело бы к тесной интеграции пользователей с применяемой моделью предметной области. Этих видов отношений по возможности нужно избегать, так как они противоречат большинству фундаментальных целей DDD.

Лучше было бы избежать представления модели своим клиентам. Разработчики стали думать о сценариях использования (или пользовательских историях), необходимых интеграторам. Это полностью соответствовало следующей части определения СЛУЖБЫ С ОТКРЫТЫМ ПРОТОКОЛОМ: "Совершенствуйте и расширяйте протокол для учета новых интеграционных требований". Это означает, что вы обеспечиваете только то, в чем интеграторы нуждаются в настоящее время, и вы понимаете эти потребности, только рассматривая весь диапазон сценариев использования.

Команда последовала этому совету и поняла, например, что на самом деле интеграторов интересует, играет ли заданный пользователь конкретную роль.

Экранирование интеграторов от деталей модели предметной области повышает их производительность и облегчает поддержку их зависимых ОГРАНИЧЕННЫХ КОНТЕКСТОВ. С точки зрения проектирования это значит, что их RESTful ресурс `User` может иметь следующую схему.

```
@Path("/tenants/{tenantId}/users")
public class UserResource {
   ...
   @GET
   @Path("{username}/inRole/{role}")
   @Produces({ OvationsMediaType.ID_OVATION_TYPE })
   public Response getUserInRole(
          @PathParam("tenantId") String aTenantId,
          @PathParam("username") String aUsername,
          @PathParam("role") String aRoleName) {

      Response response = null;

      User user = null;

      try {
          user = this.accessService().userInRole(
                        aTenantId, aUsername, aRoleName);
      } catch (Exception e) {
         // неудача
      }

      if (user != null) {
          response = this.userInRoleResponse(user, aRoleName);
      } else {
          response = Response.noContent().build();
      }

      return response;
    }
    ...
}
```

В **ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЕ (4)**, или архитектуре ПОРТЫ И АДАПТЕРЫ, класс `UserResource` является JAX-RS реализацией АДАПТЕРА ПОРТА RESTful HTTP. Потребитель создает запрос в виде

```
GET /tenants/{tenantId}/users/{username}/inRole/{role}
```

АДАПТЕР делегирует запрос объекту класса `AccessService`, представляющем собой **ПРИКЛАДНУЮ СЛУЖБУ (APPLICATION SERVICE) (14)**, которая обеспечивает интерфейс API во внутреннем шестиугольнике. Класс `AccessService` является непосредственным клиентом модели предметной области и

управляет задачей сценария использования и транзакции. Эта задача включает в себя функцию поиска, выясняющую, существует ли объект класса `User` вообще, а если существует, то играет ли он указанную роль.

```
package com.saasovation.identityaccess.application;
...
public class AccessService ... {
    ...
    @Transactional(readOnly=true)
    public User userInRole(
            String aTenantId,
            String aUsername,
            String aRoleName) {

        User userInRole = null;

        TenantId tenantId = new TenantId(new TenantId(aTenantId));

        User user =
            DomainRegistry
                .userRepository()
                .userWithUsername(tenantId, aUsername);

        if (user != null) {
            Role role =
                DomainRegistry
                    .roleRepository()
                    .roleNamed(tenantId, aRoleName);

            if (role != null) {
                GroupMemberService groupMemberService =
                     DomainRegistry.groupMemberService();

                if (role.isInRole(user, groupMemberService)) {
                    userInRole = user;
                }
            }
         }
         return userInRole;
    }
    ...
}
```

ПРИКЛАДНАЯ СЛУЖБА находит объект класса `User` и АГРЕГАТ, играющий роль, заданную объектом класса `Role`. При вызове метода запроса `isInRole()` из класса `Role` ему передается экземпляр класса `GroupMemberService`. Это не ПРИКЛАДНАЯ СЛУЖБА, а **СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN SERVICE) (7)**, помогающая классу `Role` выполнять некоторые проверки и запросы, специфичные для предметной области, за которые сам класс `Role` ответственности не несет.

Объект класса Response из объекта класса UserResource образуется из найденного объекта класса User и имени заданной роли с помощью одного из пользовательских типов носителя информации.

```
package com.saasovation.common.media;

public class OvationsMediaType {
    public static final String COLLAB_OVATION_TYPE =
    "application/vnd.saasovation.collabovation+json";

public static final String ID_OVATION_TYPE =
    "application/vnd.saasovation.idovation+json";

public static final String PROJECT_OVATION_TYPE =
    "application/vnd.saasovation.projectovation+json";
    ...
}
```

Если пользователь играет указанную роль, то АДАПТЕР UserResource создает ответ по протоколу HTTP в виде представления JSON.

```
HTTP/1.1 200 OK
Content-Type: application/vnd.saasovation.idovation+json
...
{
    "role":"Author","username":"zoe",
    "tenantId":"A94A8298-43B8-4DA0-9917-13FFF9E116ED",
    "firstName":"Zoe","lastName":"Doe",
    "emailAddress":"zoe@saasovation.com"
}
```

Как будет показано далее, заказчик интеграции, потребляющий данный ресурс RESTful, может преобразовать его в конкретный объект предметной области, необходимый в его ОГРАНИЧЕННОМ КОНТЕКСТЕ.

Реализация клиента в архитектуре REST с помощью ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ

Несмотря на то что представление JSON, созданное *Контекстом Идентификации и Доступа*, довольно полезно для клиентских интеграторов с точки зрения принципов DDD, оно не будет использоваться так, как в клиентском ОГРАНИЧЕННОМ КОНТЕКСТЕ. Как указано в предыдущих главах, если потребителем является *Контекст Сотрудничества*, команда интересуется проблемно-ориентированными, а не простыми ролями пользователей. Тот факт, что в какой-то другой модели есть ряд объектов класса User, которые могут быть присвоены одной или более ролям, моделируемым объектами класса Role, к вопросам сотрудничества не относится.

Как же заставить представление пользователя, играющего некую роль, служить конкретным целям сотрудничества? Давайте еще раз взглянем на КАРТУ КОНТЕКСТОВ, изображенную на рис. 13.1. Важные части АДАПТЕРА `UserResource` были показаны в предыдущем подразделе. Осталось разработать интерфейсы и классы `CollaboratorService`, `UserInRoleAdapter` и `CollaboratorTranslator` для *Контекста Сотрудничества*. Существует также класс `HttpClient`, но он предоставляется реализацией JAX-RS в виде классов `ClientRequest` и `ClientResponse`.

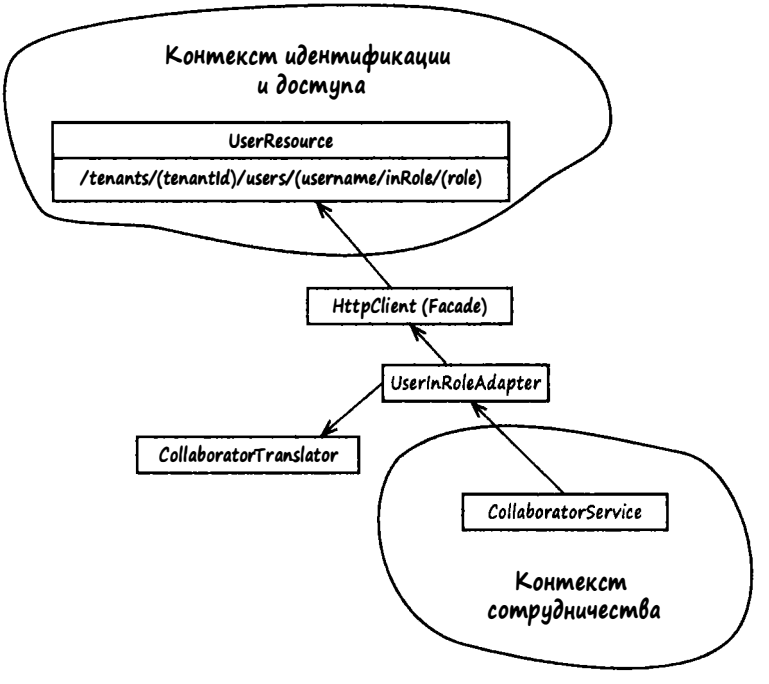


Рис. 13.1. СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ в Контексте идентификации и доступа и ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ в Контексте сотрудничества ПРЕДМЕТНОЙ ОБЛАСТИ, используемые для интеграции этих двух контекстов

Классы `CollaboratorService`, `UserInRoleAdapter` и `CollaboratorTranslator` образуют **ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (3)**, с помощью которого *Контекст сотрудничества* будет взаимодействовать с *Контекстом идентификации и доступа* и преобразовывать представление пользователя, играющего роль, в ОБЪЕКТ-ЗНАЧЕНИЕ для конкретного класса `Collaborator`.

Рассмотрим интерфейс `CollaboratorService`, обеспечивающий простые операции на ПРЕДОХРАНИТЕЛЬНОМ УРОВНЕ.

```
public interface CollaboratorService {
    public Author authorFrom(Tenant aTenant, String anIdentity);
    public Creator creatorFrom(Tenant aTenant, String anIdentity);
    public Moderator moderatorFrom(Tenant aTenant, String anIdentity);
    public Owner ownerFrom(Tenant aTenant, String anIdentity);
    public Participant participantFrom(

    Tenant aTenant, String anIdentity);
}
```

С точки зрения клиентов интерфейс `CollaboratorService` полностью абстрагируется от сложности доступа к удаленной системе и последующих переводов с ОБЩЕДОСТУПНОГО ЯЗЫКА в объекты, соответствующие локальному ЕДИНОМУ ЯЗЫКУ. В данном случае мы действительно используем ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС [Фаулер, P EAA] и класс реализации, потому что реализация носит технический характер и не должна находиться на УРОВНЕ ПРЕДМЕТНОЙ ОБЛАСТИ.

Все эти **ФАБРИКИ (11)** очень похожи друг на друга. Все они создают подкласс абстрактного типа ЗНАЧЕНИЙ `Collaborator`, но только если пользователь внутри объекта класса `aTenant`, имеющий идентификатор `anIdentity`, играет какую-то роль в классах `Author`, `Creator`, `Moderator`, `Owner` и `Participant`, имеющую отношение к безопасности. Поскольку они настолько похожи, рассмотрим реализацию только одного метода, `authorFrom()`.

```
package com.saasovation.collaboration.infrastructure.services;

import com.saasovation.collaboration.domain.model.collaborator.Author;
...

public class TranslatingCollaboratorService
       implements CollaboratorService {
    ...
    @Override
    public Author authorFrom(Tenant aTenant, String anIdentity) {
         Author author =
             this.userInRoleAdapter
                 .toCollaborator(
                        aTenant,
                        anIdentity,
                        "Author",
                        Author.class);

        return author;
    }
   ...
}
```

Отметим в первую очередь, что класс `TranslatingCollaboratorService` является частью **МОДУЛЯ (9)** ИНФРАСТРУКТУРЫ. Мы создаем ВЫДЕЛЕННЫЙ ИНТЕРФЕЙС во внутреннем шестиугольнике как часть модели предметной области. Впрочем, реализация носит технический характер и расположена вне ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ, которая включает в себя ПОРТ И АДАПТЕРЫ.

Как часть технической реализации, ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ обычно имеет специализированный **АДАПТЕР** [Гамма и др.] и транслятор. На рис. 13.1 видно, что в данном случае АДАПТЕРОМ является класс `UserInRoleAdapter`, а транслятором — класс `CollaboratorTranslator`. Специализированный класс `UserInRoleAdapter` на ПРЕДОХРАНИТЕЛЬНОМ УРОВНЕ обеспечивает обращение к удаленной системе и отправку запроса на ресурс пользователя, играющего указанную роль.

```
package com.saasovation.collaboration.infrastructure.services;

import org.jboss.resteasy.client.ClientRequest;
import org.jboss.resteasy.client.ClientResponse;
...
public class UserInRoleAdapter {
    ...
    public <T extends Collaborator> T toCollaborator(
            Tenant aTenant,
            String anIdentity,
            String aRoleName,
            Class<T> aCollaboratorClass) {

        T collaborator = null;

        try {
            ClientRequest request =
                  this.buildRequest(aTenant, anIdentity, aRoleName);

            ClientResponse<String> response =
                  request.get(String.class);

            if (response.getStatus() == 200) {
                collaborator =
                    new CollaboratorTranslator()
                            .toCollaboratorFromRepresentation(
                                  response.getEntity(),
                                  aCollaboratorClass);

            } else if (response.getStatus() != 204) {
                throw new IllegalStateException(
                      "There was a problem requesting the user: "
                      + anIdentity
                      + " in role: "
                      + aRoleName
                      + " with resulting status: "
                      + response.getStatus()); }
        } catch (Throwable t) {
               throw new IllegalStateException(
```

```
                    "Failed because: " + t.getMessage(), t);
        }

        return collaborator;
    }
    ...
}
```

Если ответ на запрос GET оказался успешным (код 200), значит, объект класса `UserInRoleAdapter` получил ресурс пользователя, играющего указанную роль, который теперь можно преобразовать в подкласс класса `Collaborator`.

```
package com.saasovation.collaboration.infrastructure.services;

import java.lang.reflect.Constructor;
import com.saasovation.common.media.RepresentationReader;
...
public class CollaboratorTranslator {
    public CollaboratorTranslator() {
        super();
    }

    public <T extends Collaborator> T toCollaboratorFromRepresentation(
            String aUserInRoleRepresentation,
            Class<T> aCollaboratorClass)
    throws Exception {

        RepresentationReader reader =
              new RepresentationReader(aUserInRoleRepresentation);

        String username = reader.stringValue("username");
        String firstName = reader.stringValue("firstName");
        String lastName = reader.stringValue("lastName");
        String emailAddress = reader.stringValue("emailAddress");

        T collaborator =
              this.newCollaborator(
                      username,
                      firstName,
                      lastName,
                      emailAddress,
                      aCollaboratorClass);

        return collaborator;
    }

    private <T extends Collaborator> T newCollaborator(
        String aUsername,
        String aFirstName,
        String aLastName,
        String aEmailAddress,
        Class<T> aCollaboratorClass)
    throws Exception {
```

```
        Constructor<T> ctor =
            aCollaboratorClass.getConstructor(
                  String.class, String.class, String.class);

        T collaborator =
            ctor.newInstance(
                  aUsername,
                  (aFirstName + " " + aLastName).trim(),
                  aEmailAddress);

        return collaborator;
    }
}
```

Этот транслятор получает текст представления пользователя, играющего роль, в виде объекта класса `String`, а также объект класса `Class`, который должен использоваться для создания объекта подкласса класса `Collaborator`. Сначала объект класса `RepresentationReader`, похожий на объект класса `NotificationReader`, рассмотренного ранее, используется для чтения четырех атрибутов из представления в формате JSON. Это вполне надежный способ, потому что пользовательский тип носителя информации в компании SaaSOvation создает обязательный контракт между производителями и потребителями. После того как транслятор получит необходимые значения класса `String`, он использует их для создания экземпляра ОБЪЕКТА-ЗНАЧЕНИЯ класса `Collaborator` и, в нашем случае, объекта класса `Author`.

```
package com.saasovation.collaboration.domain.model.collaborator;

public final class Author
       extends Collaborator {

   public Author(
          String anIdentity,
          String aName,
          String anEmailAddress) {
      super(anIdentity, aName, anEmailAddress);
    }
    ...
}
```

Поддержка синхронизации экземпляров ЗНАЧЕНИЙ класса `Collaborator` с *Контекстом идентификации и доступа* не требует никаких усилий. Они не изменяются и могут лишь полностью заменяться, но не модифицироваться. Покажем, как ПРИКЛАДНАЯ СЛУЖБА получает объект класса `Author`, а затем передает его объекту класса `Forum` для создания нового экземпляра класса `Discussion` (т.е. передает информацию об авторе в форум для начала новой дискуссии).

```
package com.saasovation.collaboration.application;
...
public class ForumService ... {
    ...
    @Transactional
    public Discussion startDiscussion(
            String aTenantId,
            String aForumId,
            String anAuthorId,
            String aSubject) {

        Tenant tenant = new Tenant(aTenantId);
        ForumId forumId = new ForumId(aForumId);

        Forum forum = this.forum(tenant, forumId);

        if (forum == null) {
            throw new IllegalStateException("Forum does not exist.");
        }

        Author author =
            this.collaboratorService.authorFrom(
                    tenant, anAuthorId);

        Discussion newDiscussion =
            forum.startDiscussion(
                    this.forumNavigationService(),
                    author,
                    aSubject);

        this.discussionRepository.add(newDiscussion);

        return newDiscussion;
    }
    ...
}
```

Если имя (или адрес) электронной почты, хранящееся в объекте класса `Collaborator`, изменится в *Контексте идентификации и доступа*, то оно не будет автоматически воспроизведено в *Контексте сотрудничества*. Такие изменения происходят редко, поэтому команда может придерживаться описанной выше простой схемы и не пытаться поддерживать синхронизацию объектов из удаленного КОНТЕКСТА с объектами, находящимися в локальном КОНТЕКСТЕ. Впрочем, как будет показано далее, *Контекст управления гибким проектированием* преследует другие цели.

Существуют и другие способы реализации ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ, например с помощью **ХРАНИЛИЩА (12)**. Однако, поскольку ХРАНИЛИЩА обычно используются для постоянного хранения и восстановления АГРЕГАТОВ, создание ОБЪЕКТОВ-ЗНАЧЕНИЙ с их помощью кажется неуместным. Если бы мы

хотели, чтобы на ПРЕДОХРАНИТЕЛЬНОМ УРОВНЕ создавались АГРЕГАТЫ, то ХРАНИЛИЩА были бы более естественным инструментом.

Интеграция с помощью сообщений

Подход к интеграции, основанный на обмене сообщениями, может обеспечить системе более высокую степень независимости. Пока инфраструктура обмена сообщениями сохраняет работоспособность, сообщения могут отправляться и передаваться, даже если какая-то из систем недоступна.

Один из способов, с помощью которых подход DDD может усилить автономность систем, основан на использовании СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ. Когда в одной из систем происходит нечто важное, она публикует соответствующее СОБЫТИЕ. В каждой системе может происходить много разных явлений, поэтому для записи информации о каждом из них создается отдельное уникальное СОБЫТИЕ. Когда СОБЫТИЯ возникают, они публикуются для сведения заинтересованных сторон посредством механизма обмена сообщениями. Это всего лишь общее схематическое описание. Если вы обошли вниманием эту тему в ранних главах, то сначала перечитайте главы, посвященные **АРХИТЕКТУРЕ (4)**, **СОБЫТИЯМ ПРЕДМЕТНОЙ ОБЛАСТИ (8)** и **АГРЕГАТАМ (10)**.

Информирование о владельцах продукта и членах команды

Контекст управления гибким проектированием предназначен для управления группой владельцев продукции Scrum и членами команды каждого арендатора, подписавшегося на эту услугу. В любой момент времени владелец продукта может создать новый продукт, а затем назначить в команду новых членов. Как же Scrum-приложение, управляющее проектом, может узнать, кто и какую роль играет? Это происходит благодаря тому, что данное приложение работает не автономно.

На самом деле *Контекст управления гибким проектированием* позволяет управлять ролями *Контексту идентификации и доступа*. Это вполне естественное и правильное решение. В данной системе для каждого арендатора, подписавшегося на услугу Scrum, будут созданы два экземпляра класса `Role`: `ScrumProductOwner` и `ScrumTeamMember`. Кроме того, каждому экземпляру класса `User`, участвующему в работе системы, будет присвоена одна из указанных ролей. Рассмотрим метод ПРИКЛАДНОЙ СЛУЖБЫ в *Контексте идентификации и доступа*, управляющий распределением ролей среди пользователей.

```
package com.saasovation.identityaccess.application;
...
public class AccessService ... {
```

```
    ...
    @Transactional
    public void assignUserToRole(AssignUserToRoleCommand aCommand) {

        TenantId tenantId =
                new TenantId(aCommand.getTenantId());

        User user =
                this.userRepository
                    .userWithUsername(
                         tenantId,
                         aCommand.getUsername());

        if (user != null) {
            Role role =
                   this.roleRepository
                       .roleNamed(
                            tenantId,
                            aCommand.getRoleName());

            if (role != null) {
                role.assignUser(user);
            }

        }
    }
    ...
}
```

Отлично, но как это поможет *Контексту управления гибким проектированием* узнать, кто играет роль `ScrumTeamMember` или `ScrumProductOwner`? Вот как это происходит. Когда метод `assignUser()` класса `Role` завершает свое выполнение, он публикует СОБЫТИЕ.

```
package com.saasovation.identityaccess.domain.model.access;
...
public class Role extends Entity {
    ...
    public void assignUser(User aUser) {

    if (aUser == null) {
        throw new NullPointerException("User must not be null.");
    }
    if (!this.tenantId().equals(aUser.tenantId())) {
        throw new IllegalArgumentException(
              "Wrong tenant for this user.");
    }

    this.group().addUser(aUser);

    DomainEventPublisher
        .instance()
```

```
        .publish(new UserAssignedToRole(
             this.tenantId(),
             this.name(),
             aUser.username(),
             aUser.person().name().firstName(),
             aUser.person().name().lastName(),
             aUser.person().emailAddress()));
    }
    ...
}
```

В конце концов событие `UserAssignedToRole`, дополненное свойствами экземпляра класса `User`, задающими имя и электронный адрес пользователя, доставляется всем заинтересованным сторонам. Когда *Контекст управления гибким проектированием* получает СОБЫТИЕ, он использует его для того, чтобы убедиться, что в модели созданы новые экземпляры классов `TeamMember` или `ProductOwner`. Это не слишком сложный сценарий использования. Впрочем, существует намного больше деталей, которыми необходимо управлять, чем кажется на первый взгляд. Давайте разберемся в них.

Оказывается, прослушивание уведомлений от системы RabbitMQ сопряжено с аспектами повторного использования. У нас уже есть простая объектно-ориентированная библиотека, упрощающая использование Java-клиента. Теперь мы добавим еще один простой класс, значительно упрощающий создание пользователя очереди сообщений.

```
package com.saasovation.common.port.adapter.messaging.rabbitmq;
...
public abstract class ExchangeListener {

   private MessageConsumer messageConsumer;
   private Queue queue;
   public ExchangeListener() {
       super();

       this.attachToQueue();
       this.registerConsumer();
   }

   protected abstract String exchangeName();

   protected abstract void filteredDispatch(
       String aType, String aTextMessage);

   protected abstract String[] listensToEvents();

   protected String queueName() {
       return this.getClass().getSimpleName();
   }
```

```
    private void attachToQueue() {
        Exchange exchange =

            Exchange.fanOutInstance(
                   ConnectionSettings.instance(),
                   this.exchangeName(),
                   true);

        this.queue =

            Queue.individualExchangeSubscriberInstance(
                   exchange,
                   this.exchangeName() + "." + this.queueName());

    }

     private Queue queue() {
        return this.queue;
     }

     private void registerConsumer() {
        this.messageConsumer =
             MessageConsumer.instance(this.queue(), false);

        this.messageConsumer.receiveOnly(
            this.listensToEvents(),
                 new MessageListener(MessageListener.Type.TEXT) {

                 @Override
                 public void handleMessage(
                        String aType,
                        String aMessageId,
                        Date aTimestamp,
                        String aTextMessage,
                        long aDeliveryTag,
                        boolean isRedelivery)
                 throws Exception {
                     filteredDispatch(aType, aTextMessage);
                 }
            });
        }
}
```

ExchangeListener — это абстрактный базовый класс, который повторно используется конкретным подписчиком. Для того чтобы расширить его, в конкретном классе нужно добавить лишь немного кода. Во-первых, он гарантирует, что будет вызван конструктор по умолчанию из базового класса. Впрочем, это произойдет в любом случае. После этого остается лишь реализовать три абстрактных метода, два из которых очень простые: `exchangeName()`, `filteredDispatch()` и `listensToEvents()`.

Для реализации метода `exchangeName()` достаточно вернуть имя сообщения, уведомление о котором получает подписчик, в виде экземпляра класса `String`. Для реализации абстрактного метода `listensToEvents()` необходимо вернуть массив `String[]`, содержащий типы уведомлений, которые вы желаете получать. Многие подписчики будут получать уведомления только одного типа. Для них массив должен содержать только один элемент. Последний метод, `filteredDispatch()`, — самый сложный из трех, потому что он предназначен для обработки основной массы полученных сообщений. Для того чтобы увидеть, как он работает, рассмотрим подписчик уведомлений о событии `UserAssignedToRole`.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class TeamMemberEnablerListener extends ExchangeListener {

    @Autowired
    private TeamService teamService;

    public TeamMemberEnablerListener() {
        super();
    }

    @Override
    protected String exchangeName() {
        return Exchanges.IDENTITY_ACCESS_EXCHANGE_NAME;
    }

    @Override
    protected void filteredDispatch(
              String aType,
              String aTextMessage) {
        NotificationReader reader =
              new NotificationReader(aTextMessage);

        String roleName = reader.eventStringValue("roleName");

        if (!roleName.equals("ScrumProductOwner") &&
            !roleName.equals("ScrumTeamMember")) {
            return;
        }

       String emailAddress = reader.eventStringValue("emailAddress");
       String firstName = reader.eventStringValue("firstName");
       String lastName = reader.eventStringValue("lastName");
       String tenantId = reader.eventStringValue("tenantId.id");
       String username = reader.eventStringValue("username");
       Date occurredOn = reader.occurredOn();

       if (roleName.equals("ScrumProductOwner")) {
           this.teamService.enableProductOwner(
                new EnableProductOwnerCommand(
```

```
                tenantId,
                username,
                firstName,
                lastName,
                emailAddress,
                occurredOn));
    } else {
        this.teamService.enableTeamMember(
                new EnableTeamMemberCommand(
                    tenantId,
                    username,
                    firstName,
                    lastName,
                    emailAddress,
                    occurredOn));
        }
    }

    @Override
    protected String[] listensToEvents() {
        return new String[] {
        "com.saasovation.identityaccess.domain.model.
               access.UserAssignedToRole" }; } }
```

Конструктор по умолчанию класса `ExchangeListener` вызывается правильно, метод `exchangeName()` возвращает имя сообщения, опубликованного *Контекстом идентификации и доступа*, а метод `listensToEvents()` возвращает одноэлементный массив с полностью определенным именем класса СОБЫТИЯ `UserAssignedToRole`. Обратите внимание на то, что издатели и подписчики должны использовать полностью определенные имена классов, содержащие имена МОДУЛЯ и класса. Это позволяет предотвратить неоднозначность, которая может возникнуть из-за одноименных СОБЫТИЙ из разных ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

Основные операции содержатся в классе `filteredDispatch()`. Как следует из его имени, он фильтрует уведомление перед тем, как направить его в интерфейс ПРИКЛАДНОЙ СЛУЖБЫ, игнорируя все уведомления типа `UserAssignedToRole`, которые не порождают СОБЫТИЯ о ролях с именами `ScrumProductOwner` и `ScrumTeamMember`. С другой стороны, если роли относятся к полученному СОБЫТИЮ, то метод получает детали уведомления класса `UserAssignedToRole` и передает его ПРИКЛАДНОЙ СЛУЖБЕ с именем `TeamService`. Методы СЛУЖБЫ `enableProductOwner()` и `enableTeamMember()` получают командный объект типа `EnableProductOwnerCommand` или `EnableTeamMemberCommand` соответственно.

На первый взгляд, может показаться, что член команды должен создаваться просто в результате появления одного из этих событий. Однако, поскольку связь между экземплярами классов `User` и `Role` может то устанавливаться, то

разрываться, вполне возможно, что член команды, представленный объектом класса `User` в полученном уведомлении, уже существует. Вот как класс `TeamService` обрабатывает эту ситуацию.

```
package com.saasovation.agilepm.application;
...
public class TeamService ... {

    @Autowired
    private ProductOwnerRepository productOwnerRepository;

    @Autowired
    private TeamMemberRepository teamMemberRepository;
    ...

    @Transactional
    public void enableProductOwner(
                EnableProductOwnerCommand aCommand) {
        TenantId tenantId = new TenantId(aCommand.getTenantId());

        ProductOwner productOwner =
            this.productOwnerRepository.productOwnerOfIdentity(
                    tenantId,
                    aCommand.getUsername());

        if (productOwner != null) {
            productOwner.enable(aCommand.getOccurredOn());
        } else {
            productOwner =
                new ProductOwner(
                        tenantId,
                        aCommand.getUsername(),
                        aCommand.getFirstName(),
                        aCommand.getLastName(),
                        aCommand.getEmailAddress(),
                        aCommand.getOccurredOn());

            this.productOwnerRepository.add(productOwner);
        }
    }
}
```

Например, метод СЛУЖБЫ `enableProductOwner()` учитывает возможность, что конкретный объект класса `ProductOwner` уже существует. Если он уже существует, то его не нужно создавать снова, поэтому мы передаем его соответствующей команде. Если же объект класса `ProductOwner` еще не существует, мы создаем новый экземпляр АГРЕГАТА и добавляем его в ХРАНИЛИЩЕ. Поскольку объект класса `TeamMember` обрабатывается точно так же, метод `enableTeamMember()` реализуется аналогично.

Можете ли вы принять на себя ответственность

Все это выглядит прекрасно и достаточно просто. Мы спроектировали классы АГРЕГАТОВ `ProductOwner` и `TeamMember`, содержащие информацию об экземпляре класса `User` из внешнего ОГРАНИЧЕННОГО КОНТЕКСТА. Но понимаете ли вы, какую ответственность подразумевает такой проект АГРЕГАТОВ?

Напомним, что команда решила создать в *Контексте сотрудничества* неизменяемые ОБЪЕКТЫ-ЗНАЧЕНИЯ, которые будут содержать аналогичную информацию (см. раздел "Реализация клиента в архитектуре REST с помощью ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ"). Поскольку значения не изменяются, команда не должна беспокоиться об обновлении совместно используемой информации. Разумеется, побочным эффектом этого обстоятельства является то, что при обновлении части этой информации *Контекст сотрудничества* никогда не будет обновлять соответствующие объекты, которые были созданы ранее. Таким образом, команда УПРАВЛЕНИЯ ГИБКИМ ПРОЕКТИРОВАНИЕМ столкнется с противоположной проблемой.

Однако обновление АГРЕГАТОВ требует решения нескольких задач. Почему? Нельзя ли просто подписаться на дополнительные уведомления о СОБЫТИЯХ, отражающие изменения в экземплярах класса `User`, соответствующих экземплярам классов `ProductOwner` и `TeamMember`? Да, мы действительно можем и должны так сделать. Однако тот факт, что мы используем инфраструктуру обмена сообщениями, создает больше сложностей, чем может показаться.

Например, что произойдет, если менеджер в *Контексте идентификации и доступа* случайно отменит назначение Джо Джонсона на роль `ScrumTeamMember`? Да, мы получим уведомление об этом факте и поэтому сможем использовать объект класса `TeamService` для того, чтобы отключить объект класса `TeamMember`, соответствующий Джо Джонсону. Теперь представьте себе, что через несколько секунд менеджер понял, что это действие было ошибочным и что нужно было снять с роли не Джо Джонсона, а Джо Джонса. После этого *Контекст управления гибким проектированием* получит соответствующее уведомление и все будут счастливы (за исключением, может быть, Джо Джонса). А все *ли действительно* хорошо?

Возможно, мы неправильно представляем себе этот сценарий использования. Мы предполагаем, что получаем уведомления в том порядке, в котором они возникают в *Контексте идентификации и доступа*. На самом деле ситуация может оказаться не такой благополучной. Что произойдет, если уведомление о Джо Джонсоне будет получено в следующем порядке: сначала — `UserAssignedToRole`, а затем — `UserUnassignedFromRole`? Что случится, если объект класса `TeamMember`, соответствующий Джо Джонсону, "зависнет" в неисправном состоянии, и кому-то придется восстанавливать данные в базе данных *Контекста управления*

гибким проектированием или менеджер будет вынужден прибегать к уловкам, чтобы восстановить правильную роль Джо. По иронии судьбы все, что может случиться, рано или поздно происходит. Итак, как же это предотвратить?

Рассмотрим поближе командные объекты, которые мы передаем как параметры интерфейсам прикладного программирования `TeamService`. Например, рассмотрим команды `EnableTeamMemberCommand` и `DisableTeamMemberCommand`. Каждая из них должна получать объект класса `Date`, а именно — `occurredOn`. Фактически все наши командные объекты разработаны именно так. Мы будем использовать значения из объекта `occurredOn` для проверки того, что наши АГРЕГАТЫ `ProductOwner` и `TeamMember` обрабатывают командные операции в правильном порядке. Возвращаясь к проблематичному сценарию использования, описанному выше, мы видим, что произойдет, если мы учтем возможность того, что объект класса `UserUnassignedFromRole` поступит после объекта класса `UserAssignedToRole`, даже если на самом деле они поступают в обратном порядке.

```
package com.saasovation.agilepm.application;
...
public class TeamService ... {
    ...
    @Transactional
    public void disableTeamMember(DisableTeamMemberCommand aCommand) {
        TenantId tenantId = new TenantId(aCommand.getTenantId());

        TeamMember teamMember =
                this.teamMemberRepository.teamMemberOfIdentity(
                        tenantId,
                        aCommand.getUsername());

        if (teamMember != null) {
            teamMember.disable(aCommand.getOccurredOn());
        }
    }
}
```

Отметим, что, передавая управление командному методу `disable()` класса `TeamMember`, мы обязаны передать ему значение `occurredOn` из командного объекта. Объект класса `TeamMember` использует его для проверки того, что отмена роли будет выполнена только тогда, когда это действительно должно произойти.

```
package com.saasovation.agilepm.domain.model.team;
...
public abstract class Member extends Entity {
    ...
    private MemberChangeTracker changeTracker;
```

```
    ...
    public void disable(Date asOfDate) {
        if (this.changeTracker().canToggleEnabling(asOfDate)) {
            this.setEnabled(false);
            this.setChangeTracker(
                 this.changeTracker().enablingOn(asOfDate));
        }
    }

    public void enable(Date asOfDate) {
        if (this.changeTracker().canToggleEnabling(asOfDate)) {
            this.setEnabled(true);
            this.setChangeTracker(
                 this.changeTracker().enablingOn(asOfDate));
        }
    }
    ...
}
```

Обратите внимание на то, что поведение АГРЕГАТА описывается абстрактным базовым классом `Member`. Методы `disable()` и `enable()` запрашивают объект `changeTracker`, чтобы выяснить, следует ли применять указанную операцию в соответствии с параметром `asOfDate` (значением `occurredOn`). ОБЪЕКТ-ЗНАЧЕНИЕ `MemberChangeTracker` отслеживает выполнение недавних операций и отвечает на запрос.

```
package com.saasovation.agilepm.domain.model.team;
...
public final class MemberChangeTracker implements Serializable {
    private Date emailAddressChangedOn;
    private Date enablingOn;
    private Date nameChangedOn;
    ...
    public boolean canToggleEnabling(Date asOfDate) {
      return this.enablingOn().before(asOfDate);
    }
    ...

public MemberChangeTracker enablingOn(Date asOfDate) {
    return new MemberChangeTracker(
           asOfDate,
           this.nameChangedOn(),
           this.emailAddressChangedOn());
    }
    ...
}
```

Если операция разрешена и уже выполнена, то метод `enablingOn()` возвращает замещающий объект класса `MemberChangeTracker`. Поскольку мы допускаем, что изменения объектов классов `PersonNameChanged` и `Person-`

ContactInformationChanged могут происходить в неправильном порядке, нам нужны функции emailAddressChangedOn и nameChangedOn. Фактически здесь предусмотрена дополнительная проверка изменения электронного адреса. Вполне возможно, что СОБЫТИЯ класса PersonContactInformationChanged содержат информацию об изменении телефонного номера или почтового адреса, а не менее распространенного адреса электронной почты.

```
package com.saasovation.agilepm.domain.model.team;
...
public abstract class Member extends Entity {
    ...
    public void changeEmailAddress(
        String anEmailAddress,
               Date asOfDate) {

        if (this.changeTracker().canChangeEmailAddress(asOfDate) &&
            !this.emailAddress().equals(anEmailAddress)) {
            this.setEmailAddress(anEmailAddress);
            this.setChangeTracker(
                 this.changeTracker().emailAddressChangedOn(asOfDate));
        }
    }
    ...
}
```

Здесь мы проверяем, действительно ли изменился адрес электронной почты. Если нет, то проверять изменения не требуется, и в этом случае мы игнорируем реальные изменения, которые внесло в электронный адрес внеочередное СОБЫТИЕ.

Кроме того, объект класса MemberChangeTracker обеспечивает идемпотентность командных операций подкласса Member, предотвращая неоднократную доставку одного и того же уведомления через инфраструктуру обмена сообщениями.

Может показаться, что включение класса MemberChangeTracker в проект АГРЕГАТА является ошибкой и что он не имеет ничего общего с ЕДИНЫМ ЯЗЫКОМ и командами, работающими по методологии Scrum. Это правда. Однако класс MemberChangeTracker остается невидимым за пределами АГРЕГАТА. Он представляет собой деталь реализации, и клиенты ничего не знают о его существовании. Клиенты знают лишь о том, что они должны передать значение occurredOn, когда происходит модификация. Более того, это именно такая деталь реализации, о которой писал Пэт Хелланд (Pat Helland), когда описывал управление отношениями партнерства в масштабируемых распределенных системах, обладающих свойством конечной согласованности. Особое внимание в статье [Helland] следует обратить на раздел 5 "Activities: Coping with Messy Messages".

Итак, вернемся к вопросу об ответственности...

Несмотря на то что мы рассмотрели самый простой пример поддержки изменений для дублирования информации, исходящей из внешнего ограниченного контекста, вопрос об ответственности далеко не такой простой, по крайней мере, если вы используете механизм обмена сообщениями, допускающий нарушение порядка поступления сообщений и их неоднократную доставку.[1] Далее, когда мы выясним все возможные операции в *Контексте идентификации и доступа*, которые могут повлиять лишь на небольшой набор атрибутов, поддерживаемых в классе `Member`, может прозвучать тревожный звонок.

- `PersonContactInformationChanged`
- `PersonNameChanged`
- `UserAssignedToRole`
- `UserUnassignedFromRole`

Затем выяснится, что существует лишь несколько других СОБЫТИЙ, на которые необходимо реагировать:

- `UserEnablementChanged`
- `TenantActivated`
- `TenantDeactivated`

Все эти факты означают, что при малейшей возможности следует минимизировать или даже полностью исключить дублирование информации между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Бывают ситуации, в которых невозможно полностью избежать дублирования информации. Например, соглашения об уровне предоставления услуги (SLA — Service Level Agreement) могут запрещать беспрепятственное получение данных из удаленных источников. Это одна из причин, по которым команда должна была хранить имя и адрес его электронной почты локально. Однако, стремясь сократить объем внешней информации, мы принимаем на себя ответственность за максимально эффективное выполнение своей работы. Такой подход называется минималистской интеграцией.

Конечно, невозможно избежать дублирования идентификаторов пользователей и арендаторов. Дублирование идентификаторов в ОГРАНИЧЕННЫХ КОНТЕКСТАХ в принципе является необходимым. Это один из основных способов,

[1] В этом случае способ получения сообщений, основанный на архитектуре RESTful, обладает явным преимуществом, потому что уведомления гарантированно доставляются в том порядке, в котором они были добавлены в ХРАНИЛИЩЕ СОБЫТИЙ (EVENT STORE) (4, ПРИЛОЖЕНИЕ). Все уведомления, от первого до последнего, поступающие по разным причинам, гарантированно обрабатываются в одном и том же порядке.

с помощью которых можно вообще выполнить интеграцию ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Кроме того, совместное использование идентификаторов совершенно безопасно, потому что они не изменяются. Мы можем даже отключать и логически удалять АГРЕГАТЫ, чтобы гарантировать, что объекты, на которые мы ссылаемся, никогда не исчезают, как это мы уже делали с классами `Tenant`, `User`, `ProductOwner` и `TeamMember`.

Это предупреждение не означает, что СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ нельзя дополнять свойствами, относящимися к передаче информации. Разумеется, СОБЫТИЯ должны содержать достаточно информации, чтобы потребители знали, какие действия они должны предпринять в ответ на поступившие сообщения. В то же время данные, содержащиеся в СОБЫТИЯХ, можно использовать для выполнения вычислений и изменения состояния получающих внешних ОГРАНИЧЕННЫХ КОНТЕКСТОВ, не сохраняя эти данные и предполагая, что ответственность за синхронизацию с официальным состоянием несет система записей.

Длительные процессы, или Как избежать ответственности

Если в предыдущем разделе мы уподобились ответственным взрослым, то этот раздел можно сравнить с попыткой вернуться в детство. Как вы знаете, взрослые должны принимать на себя все виды ответственности. Родители должны покупать автомобили, обслуживать их, платить за бензин и ремонт. Подростки же просто хотят использовать автомобиль своих родителей, а не платить за его содержание. У подростков нет желания платить автомобильный налог, оплачивать бензин, работу механика или страховку. Они просто позволяют своим родителям заботиться о реалиях, оставляя себе право на развлечения.

В этом разделе мы собираемся развлечься с **ДЛИТЕЛЬНЫМИ ПРОЦЕССАМИ (LONG-RUNNING PROCESSES) (4)**, отказываясь признавать какую бы то ни было ответственность за дублирование информации из других ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Мы просто позволим системе записей работать с ее собственной информацией после того, как внешний ОГРАНИЧЕННЫЙ КОНТЕКСТ создаст и сохранит необходимые нам данные.

В главе, посвященной **КАРТАМ КОНТЕКСТОВ (3)**, описан сценарий использования *Создание продукта*.

Предусловие: существует возможность сотрудничества (средство куплено).

1. Пользователь предоставляет описательную информацию о продукте.
2. Пользователь выражает желание обсудить ее с командой.
3. Пользователь посылает запрос на создание указанного продукта.
4. Система создает продукт, а также форум и обсуждение.

С этого места мы начинаем развлекаться и снимаем с себя ответственность за передачу информации по сети.

В главе, посвященной **КАРТАМ КОНТЕКСТОВ (3)**, команда предложила подход к интеграции двух ОГРАНИЧЕННЫХ КОНТЕКСТОВ, основанный на архитектуре RESTful. Однако в итоге команда остановилась на решении, использующем систему передачи сообщений.

Кроме того, легко заметить, что предложенная концепция, изначально включенная в ЕДИНЫЙ ЯЗЫК в виде класса `Discussion` (см. главу 3), была уточнена. Команда управления гибким проектированием выявила необходимость различать типы **обсуждений**, поэтому возникли два новых типа: `ProductDiscussion` и `BacklogItemDiscussion`. (В этом разделе мы рассмотрим только класс `ProductDiscussion`.) Оба ОБЪЕКТА-ЗНАЧЕНИЯ имеют одинаковое основное состояние и поведение, но различаются способами обеспечения типовой безопасности, помогая пользователям избегать неправильных **обсуждений** `Product` и `BacklogItem`. С практической точки зрения они не отличаются друг от друга. Каждый из этих типов **обсуждений** просто содержит признак доступности и идентификатор реального экземпляра АГРЕГАТА `Discussion` в *Контексте сотрудничества* для начатого **обсуждения**.

Стоит напомнить, что изначальное предложение назвать один из ОБЪЕКТОВ-ЗНАЧЕНИЙ в *Контексте управления гибким проектированием* так же, как АГРЕГАТ в *Контексте сотрудничества*, не было ошибкой. Итак, для полной ясности укажем, что имена ОБЪЕКТА-ЗНАЧЕНИЯ в классах `Discussion` и `ProductDiscussion` не изменяются и отличаются от имени АГРЕГАТА в *Контексте сотрудничества*. С точки зрения КАРТЫ КОНТЕКСТОВ было бы очень удобно оставить имена ОБЪЕКТОВ-ЗНАЧЕНИЙ прежними, потому что эти объекты различаются именно по КОНТЕКСТУ. Мы приняли решение создать два разных типа ЗНАЧЕНИЙ в *Контексте управления гибким проектированием* только для изоляции локальной модели.

Прежде чем приступить к анализу примера, сначала взглянем на ПРИКЛАДНУЮ СЛУЖБУ (API), создающую экземпляр класса `Product`.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {

    @Autowired
    private ProductRepository productRepository;

    @Autowired
    private ProductOwnerRepository productOwnerRepository;
    ...

    @Transactional
```

```
    public String newProductWithDiscussion(
                NewProductCommand aCommand) {

    return this.newProductWith(
           aCommand.getTenantId(),
           aCommand.getProductOwnerId(),
           aCommand.getName(),
           aCommand.getDescription(),
           this.requestDiscussionIfAvailable());

     }
     ...
}
```

Существует два способа создать новый экземпляр класса `Product`. Первый способ, не показанный здесь, подразумевает создание экземпляра класса `Product` без участия экземпляра класса `Discussion`, который пытается создать экземпляр класса `ProductDiscussion` и связать его с экземпляром класса `Product`. Два внутренних метода, `newProductWith()` и `requestDiscussionIfAvailable()`, здесь не показаны. Второй метод предназначен для проверки доступности надстройки CollabOvation. Если она установлена, возвращается состояние доступа `REQUESTED`; в противном случае возвращается значение `ADD_ON_NOT_ENABLED`. Метод `newProductWith()` вызывает конструктор класса `Product`, показанный ниже.

```
package com.saasovation.agilepm.domain.model.product;
...
public class Product extends ConcurrencySafeEntity {
    ...
    public Product(
           TenantId aTenantId,
           ProductId aProductId,
           ProductOwnerId aProductOwnerId,
           String aName,
           String aDescription,
           DiscussionAvailability aDiscussionAvailability) {

    this();

    this.setTenantId(aTenantId);
    this.setProductId(aProductId);
    this.setProductOwnerId(aProductOwnerId);
    this.setName(aName);
    this.setDescription(aDescription);

    this.setDiscussion(
            ProductDiscussion.fromAvailability(
                   aDiscussionAvailability));
    DomainEventPublisher
           .instance()
           .publish(new ProductCreated(
```

```
                this.tenantId(),
                this.productId(),
                this.productOwnerId(),
                this.name(),
                this.description(),
                this.discussion().availability().isRequested()));
    }
    ...
}
```

Клиент должен передать параметр класса DiscussionAvailability, который может принимать одно из следующих значений: ADD_ON_NOT_ENABLED, NOT_REQUESTED или REQUESTED. Состояние READY означает завершение. Если в результате создания объекта класса ProductDiscussion возвращается одно из первых двух состояний, значит, ассоциированного обсуждения нет, по крайней мере после создания объекта. Если возвращается третье значение, REQUESTED, создается экземпляр класса ProductDiscussion с состоянием PENDING_SETUP. В конструкторе класса Product используется ФАБРИЧНЫЙ МЕТОД ProductDiscussion.

```
package com.saasovation.agilepm.domain.model.product;
...
public final class ProductDiscussion implements Serializable {
    ...
    public static ProductDiscussion fromAvailability(
            DiscussionAvailability anAvailability) {

        if (anAvailability.isReady())  {
            throw new IllegalArgumentException(
                    "Cannot be created ready.");
        }

        DiscussionDescriptor descriptor =
                new DiscussionDescriptor(
                        DiscussionDescriptor.UNDEFINED_ID);

        return new ProductDiscussion(descriptor, anAvailability);
    }
    ...
}
```

Если запрос не находится в состоянии READY, что было бы проблемой, то мы получаем экземпляр ProductDiscussion, пребывающий в одном из трех состояний, и неопределенный дескриптор. Если запрос находится в состоянии REQUESTED, то под управлением ДЛИТЕЛЬНОГО ПРОЦЕССА будут созданы обсуждение сотрудничества и экземпляр класса Product. Как? Напомним, что в самом конце конструктор класса Product публикует СОБЫТИЕ ProductCreated.

```
package com.saasovation.agilepm.domain.model.product;
   ...
   public Product(...) {
       ...
       DomainEventPublisher
             .instance()
             .publish(new ProductCreated(
                  this.tenantId(),
                  this.productId(),
                  this.productOwnerId(),
                  this.name(),
                  this.description(),
                  this.discussion().availability().isRequested()));
   }
   ...
}
```

Если дискуссия находится в состоянии доступности REQUESTED, то последний параметр конструктора СОБЫТИЕ будет равен `true`. Это именно то, что необходимо для начала ДЛИТЕЛЬНОГО ПРОЦЕССА.

Вернемся к **СОБЫТИЯМ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**; каждый экземпляр СОБЫТИЯ, включая экземпляры типа `ProductCreated`, добавляется в ХРАНИЛИЩЕ СОБЫТИЙ, соответствующее конкретному ОГРАНИЧЕННОМУ КОНТЕКСТУ, в котором возникло СОБЫТИЕ. Затем все вновь добавленные СОБЫТИЯ направляются из ХРАНИЛИЩА СОБЫТИЙ заинтересованным сторонам с помощью механизма обмена сообщениями. Команды проектировщиков компании SaaSOvation решили для этого использовать систему RabbitMQ. Нам необходимо создать простой ДЛИТЕЛЬНЫЙ ПРОЦЕСС для управления обсуждением, а затем связать его с экземпляром класса `Product`.

Прежде чем перейти к описанию ДЛИТЕЛЬНОГО ПРОЦЕССА, рассмотрим еще один возможный способ запроса обсуждения. Что произойдет, если конкретный экземпляр класса `Product` будет создан в тот момент, когда обсуждение не было запрошено или настройка сотрудничества была инсталлирована, но еще не доступна? Позже владелец продукта решает добавить обсуждение, и надстройка становится доступной. Владелец продукта теперь может применить этот командный метод к объекту класса `Product`.

```
package com.saasovation.agilepm.domain.model.product;
...
public class Product extends ConcurrencySafeEntity {
    ...
    public void requestDiscussion(
           DiscussionAvailability aDiscussionAvailability) {

           if (!this.discussion().availability().isReady()) {
               this.setDiscussion(
                    ProductDiscussion.fromAvailability(
```

```
                        aDiscussionAvailability));

            DomainEventPublisher
                .instance()
                .publish(new ProductDiscussionRequested(
                    this.tenantId(),
                    this.productId(),
                    this.productOwnerId(),
                    this.name(),
                    this.description(),
                    this.discussion().availability().isRequested()));
        }
    }
    ...
}
```

Метод `requestDiscussion()` получает экземпляр типа `DiscussionAvailability`, потому что клиент должен передать объекту класса `Product` информацию о том, что надстройка сотрудничества инсталлирована. Конечно, в этом месте клиент имеет возможность схитрить и всегда передавать состояние `REQUESTED`, но это в конце концов может привести к неприятным последствиям, если надстройка на самом деле недоступна. Если обсуждение находится в состоянии `REQUESTED`, последний параметр конструктора СОБЫТИЯ вновь будет равен `true`, что необходимо для начала ДЛИТЕЛЬНОГО ПРОЦЕССА.

```
package com.saasovation.agilepm.domain.model.product;
...
public class ProductDiscussionRequested implements DomainEvent {
    ...
    public ProductDiscussionRequested(
            TenantId aTenantId,
            ProductId aProductId,
            ProductOwnerId aProductOwnerId,
            String aName,
            String aDescription,
            boolean isRequestingDiscussion) {
      ...
    }
    ...
}
```

Это СОБЫТИЕ имеет точно такие же свойства, что и объект класса `ProductCreated`. Благодаря этому один и тот же подписчик может обрабатывать СОБЫТИЯ обоих типов.

Может возникнуть вопрос "Целесообразно ли публиковать СОБЫТИЕ, если обсуждение не находится в состоянии `REQUESTED`?" Это целесообразно, потому что независимо от того, возможно ли выполнение запроса, он актуален, если не находится в состоянии `READY`. Именно подписчик должен определить, следует

ли что-нибудь делать в ответ на СОБЫТИЕ. Возможно, получение этого СОБЫТИЯ с параметром isRequestingDiscussion, равным false, означает проблему, или инсталляция надстройки уже выполняется, но еще не завершена. Следовательно, необходимо определенное вмешательство. Например, процесс может послать администратору группы сообщение по электронной почте.

Классы, используемые для управления ДЛИТЕЛЬНЫМ ПРОЦЕССОМ в *Контексте управления гибким проектированием*, аналогичны классам, которые использовались для управления созданием и поддержкой АГРЕГАТОВ ProductOwner и TeamMember (см. предыдущий раздел). Все подписчики, представленные здесь, связываются с помощью системы Spring, т.е. их экземпляры создаются в тот момент, когда создается прикладной контекст Spring в данном ОГРАНИЧЕННОМ КОНТЕКСТЕ. Первый подписчик регистрируется для получения двух видов уведомлений: AGILEPM_EXCHANGE_NAME, а также ProductCreated и ProductDiscussionRequested.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class ProductDiscussionRequestedListener
       extends ExchangeListener {
    ...
    @Override
    protected String exchangeName() {
       return Exchanges.AGILEPM_EXCHANGE_NAME;
    }
    ...

    @Override
    protected String[] listensToEvents() {
       return new String[] {
               "com.saasovation.agilepm.domain.model.
                                   .product.ProductCreated",
               "com.saasovation.agilepm.domain.model.
                                    .product.ProductDiscussionRequested"
       };
    }
   ...
}
```

Второй подписчик интересуется уведомлениями COLLABORATION_EXCHANGE_NAME и DiscussionStarted.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class DiscussionStartedListener extends ExchangeListener {
    ...

    @Override
    protected String exchangeName() {
```

```
        return Exchanges.COLLABORATION_EXCHANGE_NAME;
    }
    ...
    @Override
    protected String[] listensToEvents() {
        return new String[] {
            "com.saasovation.collaboration.domain.model.↵
                                forum.DiscussionStarted"
        };
    }
    ...
}
```

Легко видеть, где именно это происходит. Если первый подписчик получает экземпляры классов `ProductCreated` и `ProductDiscussionRequested`, он направит в *Контекст сотрудничества* команду, чтобы получить новые экземпляры класса `Forum` и `Discussion`, созданные для экземпляра класса `Product`. После того как запрос будет выполнен компонентами *Контекста сотрудничества*, публикуется уведомление `DiscussionStarted`. После того как оно будет получено, для экземпляра класса `Product` будет назначен соответствующий идентификатор обсуждения. Вот как вкратце выглядит ДЛИТЕЛЬНЫЙ ПРОЦЕСС. Рассмотрим работу метода `filteredDispatch()` для первого подписчика.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class ProductDiscussionRequestedListener
       extends ExchangeListener {
    private static final String COMMAND =
          "com.saasovation.collaboration.discussion.↵
                          CreateExclusiveDiscussion";
    ...
    @Override
    protected void filteredDispatch(
              String aType,
              String aTextMessage) {
        NotificationReader reader =
              new NotificationReader(aTextMessage);

        if (!reader.eventBooleanValue("requestingDiscussion")) {
            return;
        }

        Properties parameters = this.parametersFrom(reader);
        PropertiesSerializer serializer =
              PropertiesSerializer.instance();
        String serialization = serializer.serialize(parameters);
        String commandId = this.commandIdFrom(parameters);
        this.messageProducer()
            .send(
                serialization,
                MessageParameters
```

```
                    .durableTextParameters(
                            COMMAND,
                            commandId,
                            new Date()))
            .close();
    }
    ...
}
```

Если возникли СОБЫТИЯ типов `ProductCreated` и `ProductDiscussionRequested`, а атрибут `requestingDiscussion` равен `false`, то событие ИГНОРИРУЕТСЯ. В противном случае на основе состояния СОБЫТИЯ создается команда `CreateExclusiveDiscussion`, которая отправляется в механизм обмена сообщениями *Контекста сотрудничества*.

Настал удобный момент сделать паузу и поразмышлять о том, как организован этот процесс. Должен ли *Контекст управления гибким проектированием* действительно настраивать подписчика на СОБЫТИЕ, опубликованное локальным АГРЕГАТОМ? Может быть, было бы лучше вместо этого создать подписчика для СОБЫТИЯ `ProductCreated` в *Контексте сотрудничества*? Если бы мы сделали так, то были бы вынуждены просто заставить подписчика в *Контексте сотрудничества* управлять созданием эксклюзивных объектов классов `Forum` и `Discussion`, а также выбросить часть кода из *Контекста управления гибким проектированием*. Для того чтобы принять правильное решение, необходимо учесть несколько факторов.

Целесообразно ли, чтобы вышестоящий ОГРАНИЧЕННЫЙ КОНТЕКСТ прослушивал СОБЫТИЯ, поступающие из нижележащего КОНТЕКСТА? Действительно ли выше- и нижележащие системы в **СОБЫТИЙНО-ОРИЕНТИРОВАННОЙ АРХИТЕКТУРЕ (4)** обмениваются сообщениями? Надо ли их организовывать именно так? Может быть, важнее понять, следует ли интерпретировать СОБЫТИЕ `ProductCreated` в *Контексте сотрудничества* как признак того, что должны быть созданы эксклюзивные объекты класса `Forum` и `Discussion`. Имеет ли какой-либо смысл СОБЫТИЕ `ProductCreated` в *Контексте сотрудничества*? В скольких других КОНТЕКСТАХ может потребоваться аналогичная автоматическая поддержка этого конкретного свойства настолько специальных типов СОБЫТИЙ? Не лучше ли поместить поддержку любого количества внешних СОБЫТИЙ в виде команд создания в *Контексте сотрудничества*? Впрочем, существует еще один фактор, который необходимо учесть, чтобы более тщательно управлять ДЛИТЕЛЬНЫМИ ПРОЦЕССАМИ. Эта тема, которую мы обсудим немного позднее, поможет понять, почему мы выбрали этот конкретный путь.

Вернемся к примеру... Команда, поступившая в *Контекст сотрудничества*, направляется экземпляру класса `ForumService`, представляющему собой ПРИКЛАДНУЮ СЛУЖБУ. Отметим, что этот интерфейс API пока настроен на получение не командных параметров, а отдельных атрибутов.

```
package com.saasovation.collaboration.infrastructure.messaging;
...
public class ExclusiveDiscussionCreationListener
       extends ExchangeListener {

   @Autowired
   private ForumService forumService;
   ...
   @Override
   protected void filteredDispatch(
             String aType,
             String aTextMessage) {
        NotificationReader reader =
             new NotificationReader(aTextMessage);

        String tenantId = reader.eventStringValue("tenantId");
        String exclusiveOwnerId =
             reader.eventStringValue("exclusiveOwnerId");
        String forumSubject = reader.eventStringValue("forumTitle");
        String forumDescription =
             reader.eventStringValue("forumDescription");
        String discussionSubject =
             reader.eventStringValue("discussionSubject");
        String creatorId = reader.eventStringValue("creatorId");
        String moderatorId = reader.eventStringValue("moderatorId");

        forumService.startExclusiveForumWithDiscussion(
             tenantId,
             creatorId,
             moderatorId,
             forumSubject,
             forumDescription,
             discussionSubject,
             exclusiveOwnerId);
    }
    ...
}
```

Все логично, но не должен ли экземпляр класса `ExclusiveDiscussionCreationListener` послать ответное сообщение в *Контекст управления гибким проектированием*? Не совсем. АГРЕГАТЫ `Forum` и `Discussion` публикуют в ответ на свое создание СОБЫТИЯ `ForumStarted` и `DiscussionStarted`. Данный ОГРАНИЧЕННЫЙ КОНТЕКСТ публикует эти СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ с помощью механизма обмена сообщениями, определенного параметром `COLLABORATION_EXCHANGE_NAME`. Вот почему подписчик класса `DiscussionStartedListener` в *Контексте управления гибким проектированием* получает СОБЫТИЕ класса `DiscussionStarted`. Покажем, что делает подписчик в ответ на получение СОБЫТИЯ.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class DiscussionStartedListener extends ExchangeListener {

    @Autowired
    private ProductService productService;
    ...
    @Override
    protected void filteredDispatch(
                String aType,
                String aTextMessage) {
        NotificationReader reader =
                new NotificationReader(aTextMessage);

        String tenantId = reader.eventStringValue("tenant.id");
        String productId = reader.eventStringValue("exclusiveOwner");
        String discussionId =
               reader.eventStringValue("discussionId.id");

        productService.initiateDiscussion(
               new InitiateDiscussionCommand(
                   tenantId,
                   productId,
                   discussionId));
         }
         ...
}
```

Этот подписчик адаптирует свойства полученного уведомления о СОБЫТИИ и передает их в виде команды ПРИКЛАДНОЙ СЛУЖБЕ `ProductService`. Метод `initiateDiscussion()` работает следующим образом.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {

    @Autowired
    private ProductRepository productRepository;
    ...
    @Transactional
    public void initiateDiscussion(
                InitiateDiscussionCommand aCommand) {
        Product product =
                productRepository
                       .productOfId(
                            new TenantId(aCommand.getTenantId()),
                            new ProductId(aCommand.getProductId()));
                if (product == null) {
                    throw new IllegalStateException(
                          "Unknown product of tenant id: "
                          + aCommand.getTenantId()
                          + " and product id: "
                          + aCommand.getProductId());
```

```
            }

            product.initiateDiscussion(
                 new DiscussionDescriptor(
                         aCommand.getDiscussionId()));
    }
    ...
}
```

В конце выполняется поведенческий метод `initiateDiscussion()` АГРЕГАТА `Product`.

```
package com.saasovation.agilepm.domain.model.product;
...
public class Product extends ConcurrencySafeEntity {
    ...
    public void initiateDiscussion(DiscussionDescriptor aDescriptor) {
        if (aDescriptor == null) {
            throw new IllegalArgumentException(
                "The descriptor must not be null.");
        }

        if (this.discussion().availability().isRequested()) {
            this.setDiscussion(this.discussion()
                .nowReady(aDescriptor));
            DomainEventPublisher
                .instance()
                .publish(new ProductDiscussionInitiated(
                        this.tenantId(),
                        this.productId(),
                        this.discussion()));
        }
    }
    ...
}
```

Если свойство `discussion` класса `Product` все еще находится в состоянии `REQUESTED`, то оно переводится в состояние `READY` с параметром `DiscussionDescriptor`, содержащим идентификационную ссылку на эксклюзивный объект класса `Discussion` в *Контексте сотрудничества*. Запросы экземпляров класса `Forum` и `Discussion`, которые должны быть созданы и связаны с объектом класса `Product`, становятся непротиворечивыми.

Однако, если в момент вызова этой команды свойство `discussion` находится в состоянии `READ`, дальнейший переход не состоится. Является ли это ошибкой? Нет. Для того чтобы обеспечить идемпотентность операции `initiateDiscussion()`, следует сделать предположение, что свойство в данный момент находится в состоянии `READY`, а ДЛИТЕЛЬНЫЙ ПРОЦЕСС уже завершен. Возможные последующие вызовы команды будут происходить в результате повторной доставки

уведомления, а команда решила использовать механизм обмена сообщениями, доставляющий сообщения только один раз. В любом случае нам не стоит беспокоиться, потому что идемпотентная операция позволяет игнорировать влияние инфраструктуры и архитектуры системы. Кроме того, в данном конкретном случае нам не обязательно проектировать ОБЪЕКТ-ЗНАЧЕНИЕ `ProductChangeTracker`, как мы поступали при разработке подкласса `Member` и класса `MemberChangeTracker`. Тот простой факт, что свойство `discussion` находится в состоянии `READY`, содержит всю необходимую информацию.

Впрочем, этот подход все же не лишен недостатков. Что произойдет, если ДЛИТЕЛЬНЫЙ ПРОЦЕСС столкнется с проблемой, связанной с механизмом обмена сообщениями? Как сделать так, чтобы этот процесс всегда завершался полностью? Возможно, настало время немного повзрослеть.

Конечные автоматы и механизмы для отслеживания простоев

Этот процесс можно сделать более зрелым, добавив концепцию, аналогичную описанной в главе о **ДЛИТЕЛЬНЫХ ПРОЦЕССАХ (4)**. Разработчики компании SaaSOvation создали концепцию механизма слежения `TimeConstrainedProcessTracker`, допускающую повторное использование. Этот механизм наблюдает за процессами, у которых истекло отведенное для них время, а также за теми, которые могут повторно стартовать сколько угодно раз до истечения отведенного для них времени. Схема этого механизма позволяет при желании повторно запускать процессы через фиксированные интервалы времени или делать перерыв, либо вообще не предпринимая попыток возобновить процесс, либо после определенного количества таких попыток.

Подчеркнем для ясности, что этот механизм слежения не является частью СМЫСЛОВОГО ЯДРА. Он является частью технической ПОДОБЛАСТИ, которую могут повторно использовать все проекты компании SaaSOvation. Это значит, что в некоторых случаях мы можем пренебречь правилами работы с АГРЕГАТАМИ при их сохранении и последующих изменениях. Механизмы слежения являются относительно изолированными и, как правило, не создают конфликтов, связанных с параллельным выполнением, поскольку между ними и соответствующим процессом установлено отношение "один-к-одному". Если же конфликты все же возникают, можно подсчитать количество сообщений после возобновления процесса. Любое исключение в контексте доставки уведомлений заставит подписчика отказаться подтвердить его получение (NAK — negative acknowledge), что, в свою очередь, заставит систему RabbitMQ повторить доставку. Впрочем, нет никакой необходимости повторять доставку много раз.

Именно класс `Product` содержит информацию о текущем состоянии процесса и именно в этом контексте механизм слежения опубликует следующее

СООБЩЕНИЕ, когда начнется интервал простоя до повторной попытки или наблюдаемый процесс будет полностью прекращен.

```
package com.saasovation.agilepm.domain.model.product;

import com.saasovation.common.domain.model.process.ProcessId;
import com.saasovation.common.domain.model.process.ProcessTimedOut;

public class ProductDiscussionRequestTimedOut extends ProcessTimedOut {

    public ProductDiscussionRequestTimedOut(
            String aTenantId,
            ProcessId aProcessId,
            int aTotalRetriesPermitted,
            int aRetryCount) {

        super(aTenantId, aProcessId,
                aTotalRetriesPermitted, aRetryCount);
    }
}
```

Когда начинается интервал простоя до повторной попытки или наблюдаемый процесс полностью прекращен, механизм слежения использует СОБЫТИЯ `ProcessTimedOut`. Подписчики СОБЫТИЙ могут использовать метод СОБЫТИЯ `hasFullyTimedOut()`, чтобы определить, что именно означает СОБЫТИЕ — полную остановку или простое повторение. Если повторные попытки разрешены и подписчики используют класс `ProcessTimedOut`, они запрашивают у СОБЫТИЯ соответствующие индикаторы и значения с помощью методов `allowsRetries()`, `retryCount()`, `totalRetriesPermitted()` и `totalRetriesReached()`.

Имея возможность получения уведомлений о повторных попытках и полной остановке, мы можем лучше организовать процесс, используя класс `Product`. Во-первых, мы должны начать процесс с помощью существующего класса `ProductDiscussionRequestedListener`.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class ProductDiscussionRequestedListener
        extends ExchangeListener {
    @Override
    protected void filteredDispatch(
            String aType,
            String aTextMessage) {
        NotificationReader reader =
            new NotificationReader(aTextMessage);

        if (!reader.eventBooleanValue("requestingDiscussion")) {
            return;
        }
```

```
        String tenantId = reader.eventStringValue("tenantId.id");
        String productId = reader.eventStringValue("product.id");

        productService.startDiscussionInitiation(
                new StartDiscussionInitiationCommand(
                    tenantId,
                    productId));

        // посылаем команду в КОНТЕКСТ СОТРУДНИЧЕСТВА
        ...
    }
    ...
}
```

Экземпляр класса `ProductService` создает механизм слежения и сохраняет его, а также устанавливает связь между процессом и указанным объектом класса `Product`.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {
    ...
    @Transactional
    public void startDiscussionInitiation(
            StartDiscussionInitiationCommand aCommand) {

        Product product =
                productRepository
                        .productOfId(
                                new TenantId(aCommand.getTenantId()),
                                new ProductId(aCommand.getProductId()));

        if (product == null) {
            throw new IllegalStateException(
                    "Unknown product of tenant id: "
                    + aCommand.getTenantId()
                    + " and product id: "
                    + aCommand.getProductId());
        }

        String timedOutEventName =
                ProductDiscussionRequestTimedOut.class.getName();

        TimeConstrainedProcessTracker tracker =
                new TimeConstrainedProcessTracker(
                        product.tenantId().id(),
                        ProcessId.newProcessId(),
                        "Create discussion for product: "
                            + product.name(), new Date(),
                        5L * 60L * 1000L, // повтор каждые 5 минут
                        3,                // не больше 3 повторов
                        timedOutEventName);

        processTrackerRepository.add(tracker);
```

```
        product.setDiscussionInitiationId(
                tracker.processId().id());
        }
        ...
}
```

При необходимости объект класса `TimeConstrainedProcessTracker` может повторять попытки запуска три раза каждые пять минут. Правда, обычно эти значения не зашиваются в коде, и мы сделали это лишь для иллюстрации.

Нет ли здесь проблем?

Описанная выше спецификация повторного запуска процесса может вызывать проблемы, но пока мы оставим проект в нынешнем виде и будем делать вид, что никаких проблем нет.

Такой способ создания механизма слежения, связанного с объектом класса `Product`, лучше всего объясняет, почему СОБЫТИЕ `ProductCreated` обрабатывается локально, а не в *Контексте сотрудничества*. Это дает нашей системе возможность настраивать управление процессом и изолировать СОБЫТИЕ `ProductCreated` от команды `CreateExclusiveDiscussion` в *Контексте сотрудничества*.

Таймер, работающий в фоновом режиме, регулярно проверяет, не истекло ли время выполнения процесса. Таймер выполняет делегирование управления методу `checkForTimedOutProcesses()` в классе `ProcessService`.

```
package com.saasovation.agilepm.application;
...
public class ProcessService ... {
    ...
    @Transactional
    public void checkForTimedOutProcesses() {
        Collection<TimeConstrainedProcessTracker> trackers =
             processTrackerRepository.allTimedOut();

        for (TimeConstrainedProcessTracker tracker : trackers) {
            tracker.informProcessTimedOut();
        }
    }
    ...
}
```

Метод механизма слежения `informProcessTimedOut()`, подтверждающий необходимость повтора или прекращение процесса, публикует событие подкласса `ProcessTimedOut`.

Далее нам необходимо добавить нового подписчика для обработки повторных попыток и завершения процессов. При необходимости можно выполнять три попытки возобновления процесса каждые пять минут. Рассмотрим класс `ProductDiscussionRetryListener`.

```
package com.saasovation.agilepm.infrastructure.messaging;
...
public class ProductDiscussionRetryListener extends ExchangeListener {

    @Autowired
    private ProcessService processService;
    ...
    @Override
    protected String exchangeName() {
        return Exchanges.AGILEPM_EXCHANGE_NAME;
    }

    @Override
    protected void filteredDispatch(
                String aType,
                String aTextMessage) {
        Notification notification =
                NotificationSerializer
                    .instance()
                    .deserialize(aTextMessage, Notification.class);

        ProductDiscussionRequestTimedOut event =
                notification.event();

        if (event.hasFullyTimedOut()) {
                productService.timeOutProductDiscussionRequest(
                        new TimeOutProductDiscussionRequestCommand(
                                event.tenantId(),
                                event.processId().id(),
                                event.occurredOn()));
        } else {
                productService.retryProductDiscussionRequest(
                        new RetryProductDiscussionRequestCommand(
                                event.tenantId(),
                                event.processId().id()));
        }
    }

    @Override
    protected String[] listensToEvents() {
        return new String[] {
                "com.saasovation.agilepm.process.
                                ProductDiscussionRequestTimedOut"
                };
     }
}
```

Этот подписчик интересуется только событиями ProductDiscussionRequestTimedOut и может выполнять произвольное количество повторных попыток запуска и обработки завершения процесса. Именно процесс и механизм слежения определяют количество возможных уведомлений. СОБЫТИЯ будут отсылаться при выполнении одного из двух условий: процесс либо полностью прекращен, либо может быть возобновлен. В обоих случаях подписчик передает управление новому экземпляру класса ProductService. Если обнаруживается полное прекращение процесса, ситуацию обрабатывает ПРИКЛАДНАЯ СЛУЖБА.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {
    ...
    @Transactional
    public void timeOutProductDiscussionRequest(
           TimeOutProductDiscussionRequestCommand aCommand) {

       ProcessId processId =
              ProcessId.existingProcessId(
                     aCommand.getProcessId());

       TenantId tenantId = new TenantId(aCommand.getTenantId());

       Product product =
              productRepository
                    .productOfDiscussionInitiationId(
                           tenantId,
                           processId.id());

       this.sendEmailForTimedOutProcess(product);

       product.failDiscussionInitiation();
    }
    ...
}
```

Первое сообщение электронной почты отправляется владельцу продукта и содержит информацию о том, что настройка обсуждения завершилась неудачей, а затем экземпляр класса Product помечается как неудачное инициирование обсуждения. Анализируя новый метод failDiscussionInitiation() класса Product, легко увидеть, что мы должны объявить дополнительное состояние FAILED как свойство DiscussionAvailability. Метод failDiscussionInitiation() выполняет простую компенсацию, необходимую для сохранения экземпляра класса Product в правильном состоянии.

```
package com.saasovation.agilepm.domain.model.product;
...
public class Product extends ConcurrencySafeEntity {
```

```
    ...
    public void failDiscussionInitiation() {
        if (!this.discussion().availability().isReady()) {
            this.setDiscussionInitiationId(null);
            this.setDiscussion(
                 ProductDiscussion
                        .fromAvailability(
                             DiscussionAvailability.FAILED));
        }
    }
    ...
}
```

Здесь пропущена возможность того, что новое СОБЫТИЕ `DiscussionRequestFailed` может публиковаться методом `failDiscussionInitiation()`. Команда должна рассмотреть возможные преимущества такого решения. Возможно, сообщения электронной почты, которые могут отправляться владельцам продукта и другим администраторам, лучше всего было бы обрабатывать как результат именно этого СОБЫТИЯ. Помимо всего прочего, что может произойти, если метод `timeOutProductDiscussionRequest()` класса `ProductService` столкнется с проблемами при отправке электронного сообщения? Ситуация становится запутанной. (Ну да!) Команда это заметила и решила вернуться к этому вопросу позже.

С другой стороны, если СОБЫТИЕ означает, что следует повторить попытку, то подписчик делегирует управление следующей операции в классе `ProductService`.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {
    ...
    @Transactional
    public void retryProductDiscussionRequest(
           RetryProductDiscussionRequestCommand aCommand) {

        ProcessId processId =
            ProcessId.existingProcessId(
                      aCommand.getProcessId());

        TenantId tenantId = new TenantId(aCommand.getTenantId());

        Product product =
            productRepository
                   .productOfDiscussionInitiationId(
                           tenantId,
                           processId.id());

        if (product == null) {
            throw new IllegalStateException(
```

```
                    "Unknown product of tenant id: "
                     + aCommand.getTenantId()
                     + " and discussion initiation id: "
                     + processId.id());
        }

        this.requestProductDiscussion(
             new RequestProductDiscussionCommand(
                 aCommand.getTenantId(),
                    product.productId().id()));
    }
    ...
}
```

Экземпляр `Product` извлекается из ХРАНИЛИЩА по ассоциированному с ним идентификатору `ProcessId`, который содержится в атрибуте `discussionInitiationId` класса `Product`. Полученный экземпляр класса `Product` используется экземпляром класса `ProductService` (самоделегирование) для повторного запроса обсуждения.

Наконец-то мы получили желаемый результат. Если обсуждение началось успешно, то *Контекст сотрудничества* публикует СОБЫТИЕ `DiscussionStarted`. После этого экземпляр класса `DiscussionStartedListener` в *Контексте управления гибким проектированием* получает уведомление и передает управление объекту класса `ProductService`, как раньше. Однако на этот раз выполняется новая поведенческая функция.

```
package com.saasovation.agilepm.application;
...
public class ProductService ... {
    ...
    @Transactional
    public void initiateDiscussion(
             InitiateDiscussionCommand aCommand) {
       Product product =
             productRepository
                    .productOfId(
                           new TenantId(aCommand.getTenantId()),
                           new ProductId(aCommand.getProductId()));

      if (product == null) {
          throw new IllegalStateException(
                "Unknown product of tenant id: "
                + aCommand.getTenantId()
                + " and product id: "
                + aCommand.getProductId());
      }

      product.initiateDiscussion(
          new DiscussionDescriptor(
                aCommand.getDiscussionId()));
```

```
        TimeConstrainedProcessTracker tracker =
            this.processTrackerRepository.trackerOfProcessId(
                  ProcessId.existingProcessId(
                        product.discussionInitiationId()));
            tracker.completed();
    }
    ...
}
```

Экземпляр класса `ProductService` теперь обеспечивает завершение процесса, информируя об этом механизм слежения с помощью метода `completed()`. Начиная с этого момента механизм слежения больше не будет выбираться в качестве уведомителя о повторном запуске или простое. Процесс завершен.

Несмотря на внешнее благополучие, этот проект не лишен недостатков. Если оставить проект *Контекста сотрудничества* в его нынешнем виде, то при повторной попытке запроса создать экземпляр класса `Product` возникнут осложнения. Основная проблема заключается в том, что операции в *Контексте сотрудничества* пока не являются идемпотентными. Перечислим мелкие недостатки и способы их устранения.

- Поскольку проектом гарантируется хотя бы одна доставка сообщений, то после отправления уведомления механизму обмена сообщениями оно рано или поздно будет получено подписчиком или подписчиками. Если при создании новых сотрудничающих объектов возникнет какая-то задержка, которая вызовет хотя бы одну повторную попытку, то эта попытка, в свою очередь, станет причиной многократной рассылки одной и той же команды `CreateExclusiveDiscussion`. Все эти команды рано или поздно будут доставлены адресатам. Таким образом, любые попытки повторного запуска процесса заставят *Контекст сотрудничества* попытаться несколько раз создать одни и те же объекты классов `Forum` и `Discussion`. Это нежелательно, поскольку свойства классов `Forum` и `Discussion` требуют, чтобы объекты этих классов были уникальными. Таким образом, возможность многократных попыток создать объект становится совсем не безобидной ошибкой. Впрочем, с точки зрения регистрации ошибок возможно, что повторные попытки создать объекты являются следствием ошибок программирования. Возникает вопрос: "Если мы хотим полностью прервать процесс, то не следует ли отключить периодические попытки его возобновления?"
- На первый взгляд, следует отключить эти попытки в *Контексте управления гибким проектированием*, но следует помнить, что операции в *Контексте сотрудничества* должны быть идемпотентными. Напомним, что система RabbitMQ гарантирует по крайней мере *однократную доставку*, а значит,

может доставлять одно и то же командное сообщение несколько раз, даже если оно было послано только однажды. Сделав операции сотрудничества идемпотентными, мы предотвратим неоднократное создание одних и тех же экземпляров классов Forum и Discussion и подавим регистрацию неопасных отказов.

- *Контекст управления гибким проектированием* может отказать при попытке отправить команду CreateExclusiveDiscussion. Если при отправлении сообщения возникает проблема, следует позаботиться о повторном отправлении, пока не будет достигнут успех. В противном случае запрос на создание экземпляров классов Forum и Discussion никогда не будет выполнен. Есть несколько способов обеспечить повторную отправку сообщения. Например, при сбое во время отправки сообщения можно сгенерировать исключение в методе filteredDispatch(), которое приведет к появлению сообщения о недоставке (NAK). В результате система RabbitMQ поймет, что необходимо выполнить повторную отправку уведомлений о СОБЫТИЯХ ProductCreated или ProductDiscussionRequested, и объект класса ProductDiscussionRequestedListener снова их получит. Кроме того, можно просто повторять отправку сообщения, пока оно не будет получено, например с помощью алгоритма ОГРАНИЧЕННОГО СВЕРХУ ЭКСПОНЕНЦИАЛЬНОГО ОТКАТА (CAPPED EXPONENTIAL BACK-OFF). Если система RabbitMQ работает в автономном режиме, повторные отправки долгое время могут завершаться неудачей. Таким образом, лучше всего использовать комбинацию сообщений NAK и повторных попыток. Впрочем, вполне достаточно, если наш процесс будет повторять попытки три раза в течение пяти минут. В конце концов, возникновение таймаута в работе системы электронной почты требует вмешательства человека.

Итак, многие проблемы будут решены, если объект класса ExclusiveDiscussionCreationListener в *Контексте сотрудничества* будет делегировать управление идемпотентной операции ПРИКЛАДНОЙ СЛУЖБЫ.

```
package com.saasovation.collaboration.application;
...
public class ForumService ... {
    ...
    @Transactional
    public Discussion startExclusiveForumWithDiscussion(
            String aTenantId,
            String aCreatorId,
            String aModeratorId,
            String aForumSubject,
            String aForumDescription,
```

```
            String aDiscussionSubject,
            String anExclusiveOwner) {

        Tenant tenant = new Tenant(aTenantId);

        Forum forum =
                forumRepository
                     .exclusiveForumOfOwner(
                              tenant,
                              anExclusiveOwner);

        if (forum == null) {
            forum = this.startForum(
                     tenant,
                     aCreatorId,
                     aModeratorId,
                     aForumSubject,
                     aForumDescription,
                     anExclusiveOwner);
        }

        Discussion discussion =
            discussionRepository
                     .exclusiveDiscussionOfOwner(
                              tenant,
                              anExclusiveOwner);

        if (discussion == null) {
            Author author =
                    collaboratorService
                         .authorFrom(
                              tenant,
                              aModeratorId);

            discussion =
                    forum.startDiscussion(
                              forumNavigationService,
                              author,
                              aDiscussionSubject);

             discussionRepository.add(discussion);
        }

        return discussion;
    }
    ...
}
```

Пытаясь найти экземпляры классов `Forum` и `Discussion` по их эксклюзивным атрибутам владельца, мы предотвращаем попытку создать два экземпляра АГРЕГАТА, который, возможно, уже существовал. Отлично, всего несколько строчек кода намного улучшили наш событийно-ориентированный проект!

Проектирование более сложных процессов

Рассмотрим проектирование более сложных процессов. Если требуется выполнить несколько этапов завершения, то необходимо разработать более сложный конечный автомат. Для этого рассмотрим определение интерфейса класса `Process`.

```
package com.saasovation.common.domain.model.process;

import java.util.Date;

public interface Process {

   public enum ProcessCompletionType {
       NotCompleted,
       CompletedNormally,
       TimedOut
   }

    public long allowableDuration();
    public boolean canTimeout();
    public long currentDuration();
    public String description();
    public boolean didProcessingComplete();
    public void informTimeout(Date aTimedOutDate);
    public boolean isCompleted();
    public boolean isTimedOut();
    public boolean notCompleted();
    public ProcessCompletionType processCompletionType();
    public ProcessId processId();
    public Date startTime();
    public TimeConstrainedProcessTracker
               timeConstrainedProcessTracker();
    public Date timedOutDate();
    public long totalAllowableDuration();
    public int totalRetriesPermitted();
}
```

В классе `Process` доступны следующие важные операции.

- `allowableDuration()`: если процесс может быть прерван, этот метод возвращает общее время простоя или длительность интервалов простоя между повторными попытками возобновления процесса.
- `canTimeout()`: если процесс может быть прерван, этот метод возвращает значение `true`.
- `timeConstrainedProcessTracker()`: если процесс может быть прерван, этот метод возвращает новый уникальный экземпляр класса `TimeConstrainedProcessTracker`.

- `totalAllowableDuration()`: возвращает общее доступное время выполнения процесса. Если возобновление процесса невозможно, то ответом является значение, возвращаемое методом `allowableDuration()`. Если возобновление разрешено, то ответом является значение, возвращаемое методом `allowableDuration()`, умноженное на значение, возвращаемое методом `totalRetriesPermitted()`.
- `totalRetriesPermitted()`: если процесс допускает как прерывания, так и возобновления, этот метод возвращает общее допустимое количество попыток возобновления процесса.

Разработчики класса `Process` могут отслеживать прерывания и попытки возобновления с помощью класса `TimeConstrainedProcessTracker`. Создав объект класса `Process`, мы можем запросить для него уникальный механизм слежения. Следующий тест демонстрирует, как взаимодействуют экземпляр класса `Product` и механизм слежения.

```
Process process =
    new TestableTimeConstrainedProcess(
          TENANT_ID,
          ProcessId.newProcessId(),
          "Testable Time Constrained Process",
          5000L);

TimeConstrainedProcessTracker tracker =
    process.timeConstrainedProcessTracker();

process.confirm1();

assertFalse(process.isCompleted());
assertFalse(process.didProcessingComplete());
assertEquals(process.processCompletionType(),
      ProcessCompletionType.NotCompleted);

process.confirm2();

assertTrue(process.isCompleted());
assertTrue(process.didProcessingComplete());
assertEquals(process.processCompletionType(),
      ProcessCompletionType.CompletedNormally);
assertNull(process.timedOutDate());

tracker.informProcessTimedOut();

assertFalse(process.isTimedOut());
```

Экземпляр класса `Process`, созданный в этом тесте, должен быть выполнен (без попыток возобновления) в течение пяти секунд (`5000L` мс). Экземпляр

класса Process будет помечен как полностью выполненный только после вызова методов confirm1() и confirm2(). Экземпляр класса Process ожидает подтверждения обоих состояний.

```
public class TestableTimeConstrainedProcess extends AbstractProcess {
    ...
    public void confirm1() {
        this.confirm1 = true;

        this.completeProcess(ProcessCompletionType.CompletedNormally);
    }

    public void confirm2() {
        this.confirm2 = true;
        this.completeProcess(ProcessCompletionType.CompletedNormally);
    }
    ...
    protected boolean completenessVerified() {
        return this.confirm1 && this.confirm2;
    }

    protected void completeProcess(
        ProcessCompletionType aProcessCompletionType) {

        if (!this.isCompleted() && this.completenessVerified()) {
            this.setProcessCompletionType(aProcessCompletionType);
        }
    }
    ...
}
```

Даже когда экземпляр класса Process выполняет вызов метода completeProcess(), он не может быть помечен как выполненный, пока метод completenessVerified() возвращает значение true. Это будет происходить, только когда оба свойства, confirm1 и confirm2, будут равны true. Иначе говоря, обе операции, confirm1() и confirm2(), должны быть выполнены. Следовательно, метод completenessVerified() допускает подтверждение нескольких шагов процесса как выполненных еще до прекращения всего процесса, причем каждый специализированный класс Process может иметь свое определение метода completenessVerified().

А что происходит при выполнении последнего этапа нашего теста?

```
...
tracker.informProcessTimedOut();
assertFalse(process.isTimedOut());
```

По своему внутреннему состоянию механизм слежения знает, что работа экземпляра класса `Process` на самом деле не прервана. Таким образом, предположение в следующей строке кода всегда будет ложным. (Конечно, предполагается, что тест всегда должен выполняться менее чем за пять секунд и в обычных условиях.)

Базовый класс `AbstractProcess` реализует класс `Process`, играя роль АДАПТЕРА. Это позволяет действительно просто разрабатывать сложные ДЛИТЕЛЬНЫЕ ПРОЦЕССЫ. Поскольку класс `AbstractProcess` расширяет базовый класс `Entity`, можно создавать АГРЕГАТЫ как объекты класса `Process`. Например, можно было бы создать подкласс `AbstractProcess` класса `Product`, хотя на данном уровне сложности этого не требуется. Впрочем, можно представить себе более сложный процесс, для которого метод `completenessVerified()` должен проверять, выполнены ли все требуемые этапы.

Если система не допускает обмен сообщениями

Ни один подход к разработке сложных систем программного обеспечения не является панацеей. У любого подхода всегда есть проблемы и недостатки, часть из которых мы уже обсудили. Одна из проблем связанных с системой обмена сообщениями, состоит в том, что она может стать временно недоступной. Эта ситуация возникает не часто, но когда это действительно происходит, следует иметь в виду несколько аспектов.

Если механизм обмена сообщениями на какое-то время отключится, издатели уведомления не смогут отправлять через него сообщения. Так как эта ситуация может быть обнаружена клиентом публикации, вероятно, лучше всего было бы задержать попытку отправить уведомление до момента, когда система обмена сообщениями не станет доступной снова. Когда произойдет успешное отправление какого-то сообщения, станет ясно, что система вновь работает. Но сначала необходимо удостовериться, что попытки отправления происходят реже, чем обычно, когда все работает хорошо. Целесообразно установить длительность интервала между повторениями равным 30 секунд или минуте. Помните, если у вашей системы есть ХРАНИЛИЩЕ СОБЫТИЙ, то ваши СОБЫТИЯ будут продолжать ставиться в очередь и могут быть отправлены, как только обмен сообщениями станет доступен снова.

Конечно, слушатели не получат новые уведомления о СОБЫТИЯХ, если инфраструктура выйдет из строя на время. Будут ли ваши клиентские слушатели повторно активированы автоматически или им придется снова подписаться на ваш клиентский механизм на стороне потребителя, когда механизм обмена сообщениями станет доступным снова? Если автоматическое восстановление потребителей не поддерживается, то вы должны будете удостовериться, что ваши потребители повторно зарегистрированы. Иначе вы в конечном счете обнаружите, что

ваш ОГРАНИЧЕННЫЙ КОНТЕКСТ не получает уведомлений, которые необходимы, чтобы сохранить его взаимодействие с ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ, от которых он зависит. Это разновидность конечной непротиворечивости, которой следует избегать.

Источником проблем не всегда бывает механизм обмена сообщениями. Рассмотрим следующую ситуацию. Допустим, что ваш ОГРАНИЧЕННЫЙ КОНТЕКСТ становится недоступным в течение долгого промежутка времени. Когда он станет доступным снова, в очереди сообщений соберется много неотправленных сообщений. Когда ваш ОГРАНИЧЕННЫЙ КОНТЕКСТ запустится снова и зарегистрирует своих потребителей, получение и обработка всех доступных уведомлений может занять много времени. Возможно, этот интервал окажется не настолько долгим, чтобы упорно стремиться к ограничению времени простоя, разрабатывать "живую" схему развертывания и проект с избыточными узлами (кластерами) так, чтобы потеря одного узла не делала вашу систему недоступной. Однако в некоторых ситуациях невозможно избежать простоев. Например, если изменения в коде вашего приложения потребуют модификации в базе данных и вы не сможете внести изменения, не вызывая проблем, то вам придется смириться с определенным временем простоя системы. В таких случаях вашей системе обработки сообщений, вероятно, просто придется играть в догонялки. Очевидно, что этой ситуации необходимо избегать.

Резюме

В этой главе мы рассмотрели несколько способов успешной интеграции многочисленных ОГРАНИЧЕННЫХ КОНТЕКСТОВ.

- Мы рассмотрели основы интеграции и выработали соответствующие принципы интеграции систем в среде распределенных вычислений.
- Мы описали подход к интеграции многочисленных КОНТЕКСТОВ на основе ресурсов RESTful.
- Мы научились интегрировать системы с помощью сообщений.

- Мы изучили несколько примеров интеграции с помощью сообщений, включая разработку и управление ДЛИТЕЛЬНЫХ ПРОЦЕССОВ, от простых до сложных.
- Мы изучили проблемы, с которыми связано дублирование информации в разных ОГРАНИЧЕННЫХ КОНТЕКСТАХ, научились решать или избегать их.
- Мы рассмотрели простые примеры, а затем перешли к более сложным примерам, обогащающим наш опыт проектирования.

Теперь, научившись интегрировать несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, мы вернемся к работе с одним изолированным ОГРАНИЧЕННЫМ КОНТЕКСТОМ и посмотрим, как проектировать части приложения, окружающие модель предметной области.

Глава 14

Приложение

Любая программа настолько хороша, насколько она полезна.
Линус Торвальдс

Модель предметной области часто лежит в основе *приложения*. У приложения может быть пользовательский интерфейс, представляющий концепции модели предметной области и позволяющий пользователю выполнять различные действия с моделью. Пользовательский интерфейс использует службы прикладного уровня, которые координируют задачи сценария использования, управляют транзакциями и устанавливают необходимые полномочия безопасности. Кроме того, пользовательский интерфейс, **ПРИКЛАДНЫЕ СЛУЖБЫ** и модель предметной области полагаются на промышленную инфраструктурную поддержку, зависящую от платформы. Детали реализации инфраструктуры обычно включают в себя средства контейнера компонентов, управления приложением, обмена сообщениями и базы данных.

Назначение главы

- Изучить несколько способов передачи данных о модели предметной области в пользовательский интерфейс.
- Описать реализацию ПРИКЛАДНЫХ СЛУЖБ и их предназначение.
- Изучить способы отделения выходной информации от ПРИКЛАДНОЙ СЛУЖБЫ и разных клиентских типов.
- Выяснить, почему иногда возникает необходимость объединить в пользовательском интерфейсе несколько моделей и как это сделать.
- Научиться использовать инфраструктуру для технической реализации приложения.

Иногда мы работаем над моделями, которые существуют, чтобы поддерживать приложения. В частности, это относится к *Контексту Идентификации и Доступа*. Компания SaaSOvation обнаружила необходимость выделить концепции, связанные с идентификацией и управлением доступом, и сформировать вспомогательную модель, которая станет отдельным продуктом, предоставляемым по подписке. Даже для системы IdOvation нужен собственный административный

и пользовательский интерфейс самообслуживания. **НЕСПЕЦИАЛИЗИРОВАННЫЕ** и **СЛУЖЕБНЫЕ ПОДОБЛАСТИ (2)** также иногда испытывают недостаток в дополнениях, связанных с приложением, и это прекрасно. Если модель существует, чтобы поддерживать другую модель, то вспомогательная модель может быть простой совокупностью классов в отдельном **МОДУЛЕ (9)**, реализующем специальную концепцию и обеспечивающем определенные алгоритмы.[1] Другие модели требуют по крайней мере наличия некоторого человеческого пользовательского опыта и компонентов приложения. Эта глава фокусируется на последнем, более сложном разнообразии.

Мы здесь используем термин *приложение* в некотором смысле как синоним *системы* и *бизнес-службы*. Я не буду формально анализировать, в какой точке приложение становится системой, но сказал бы, что, когда приложение зависит от других приложений или служб посредством интеграции, такое решение в целом можно назвать системой. Иногда термины *приложение* и *система* используются взаимозаменяемо. Отдельную бизнес-службу, обеспечивающую одну или несколько конечных точек технической службы, тоже можно было бы назвать системой в общем смысле. Я не хочу вдаваться в подробности и разъяснять, чем различаются эти понятия в действительности, поэтому буду использовать единственный термин, позволяющий обсуждать проблемы и обязанности, характерные для приложения, системы и бизнес-службы.

Что такое приложение

Короче говоря, термином *приложение* называется тщательно подобранный набор компонентов, предназначенный для обеспечения взаимодействия и поддержки модели **СМЫСЛОВОГО ЯДРА (2)**. Это понятие обычно включает в себя саму модель предметной области, пользовательский интерфейс, внутренне используемые ПРИКЛАДНЫЕ СЛУЖБЫ и инфраструктурные компоненты. Точный состав каждого из этих компонентов изменяется от приложения к приложению и зависит от конкретной используемой **АРХИТЕКТУРЫ (4)**.

В то время как приложение открывает доступ к своим службам программно, пользовательский интерфейс интерпретируется в более широком смысле и включает в себя прикладной программный интерфейс (API). Существуют разные способы открыть доступ к службам, но этот интерфейс не предназначен для использования человеком. Этот вид пользовательского интерфейса обсуждается в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**. В этой главе я описываю аспекты графических пользовательских интерфейсов, предназначенных для человека.

---

[1] Пример неспециализированной предметной подобласти, представляющей собой отдельную модель, описан в статье Эрика Эванса (Eric Evans) *Time and Money Code Library* (`http://timeandmoney.sourceforge.net`).

Излагая эту тему, я попытаюсь избежать определенной АРХИТЕКТУРЫ. В качестве образца я использую рис. 14.1, на котором намеренно нет следов никакой типичной архитектуры. Пунктирные линии с треугольными наконечниками изображают реализацию **ПРИНЦИПА ИНВЕРСИИ ЗАВИСИМОСТИ (4)**, или **DIP**, с помощью языка UML. Сплошные линии со стреловидными наконечниками означают диспетчеризацию работы. Например, инфраструктура реализует интерфейсные абстракции из пользовательского интерфейса, ПРИКЛАДНЫХ СЛУЖБ и модели предметной области. Она также передает управление операциям ПРИКЛАДНЫХ СЛУЖБ, модели предметной области и хранилищу данных.

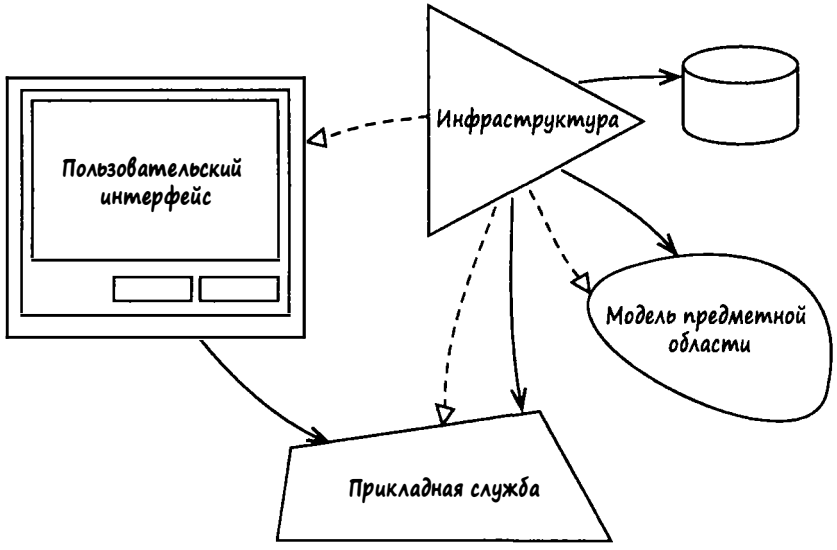


Рис. 14.1. Основные области, относящиеся к приложениям, без привязки к конкретной архитектуре. Эти области подчеркивают важность принципа DIP и зависимость абстракций от инфраструктуры

Несмотря на неизбежное перекрытие с некоторыми архитектурными стилями, главное предназначение главы состоит в описании того, как обеспечить соответствие архитектуры целям приложения. Рисунки, на которых используется конкретная архитектура, я комментирую отдельно.

Трудно не использовать термин *уровень*, как в **МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ (4)**. Это полезный термин независимо от того, какой архитектурный стиль мы обсуждаем. Например, посмотрите на место, в котором расположены ПРИКЛАДНЫЕ СЛУЖБЫ. Независимо от того, думаете ли вы о ПРИКЛАДНЫХ СЛУЖБАХ как о расположенных в кольце вокруг модели предметной области, в шестиугольнике, охватывающем модель, в капсуле, подвешенной к шине передачи сообщений, или на уровне, расположенном ниже пользовательского интерфейса и выше

модели, в каждом из этих случаев для описания этого концептуального места можно использовать термин *ПРИКЛАДНОЙ УРОВЕНЬ*. Хотя я попытаюсь воздержаться от злоупотребления этим термином, *уровень* полезен для маркировки места, в котором находятся компоненты. Это, конечно, не означает, что подход DDD ограничен существующей МНОГОУРОВНЕВОЙ АРХИТЕКТУРОЙ.[2]

Я начинаю изложение пользовательского интерфейса, затем перехожу к ПРИКЛАДНЫМ СЛУЖБАМ, а потом к инфраструктуре. Описывая каждый из этих предметов, я расскажу, где расположена модель, но не буду копаться в самой модели, потому что считаю это излишним.

Пользовательский интерфейс

На платформах Java, .NET и других существует столько каркасов пользовательского интерфейса, ориентированного на человека, что описывать их преимущества здесь мне кажется неинтересным и нецелесообразным.

Лучше разобраться в категориях более общего характера, перечисленных в следующем списке. Они перечислены в порядке важности, а не популярности. Во время работы над этой книгой, наиболее распространенной была вторая категория пользовательского веб-ориентированного интерфейса, испытывающего влияние языка HTML5. Приложения первой категории, основанные на чистых пользовательских веб-интерфейсах типа "запрос–ответ", чаще всего можно встретить в унаследованных приложениях, чем в Web 2.0.

- Чистые пользовательские веб-интерфейсы типа "запрос–ответ", широко известны как Web 1.0. Эту категорию интерфейсов поддерживают такие каркасы, как Struts, Spring MVC, Web Flow и ASP.NET.
- Пользовательские интерфейсы насыщенных веб-приложений, доступных через Интернет (Rich Internet Application — RIA), включая приложения, использующие технологии DHTML и Ajax. Эта категория известна под именем Web 2.0. К ней относятся интерфейсы GWT компании Google , YUI компании Yahoo!, Ext JS, Flex компании Adobe и Silverlight компании Microsoft.
- Естественные клиентские графические пользовательские интерфейсы (например, настольные пользовательские интерфейсы Windows, Mac и Linux), которые могут подразумевать использование абстрактных библиотек (таких, как Eclipse SWT, Java Swing, WinForms и WPF на платформе Windows). Эта категория не обязательно подразумевает насыщенное настольное приложение, хотя это возможно. Естественный клиентский графический пользовательский интерфейс может получать доступ к службе через протокол

[2] См. подробности в главе 4.

HTTP, например, делая пользовательский интерфейс единственным инсталлированным компонентом клиента.

Каждая из этих категорий порождает несколько фундаментальных вопросов: "Как перенести объекты предметной области на экран?" и "Как описать жесты пользователя в модели?"

Визуализация объектов предметной области

Существует изрядное количество противоречий и разногласий по поводу того, как лучше всего представить объекты модели предметной области в пользовательском интерфейсе, извлекая выгоду из представлений данных, более богатых, чем требуется, чтобы выполнить прямую задачу. Для того чтобы предоставить информацию о поддержке, в которой нуждаются пользователи, принимающие интеллектуальные решения при решении очередной задачи, необходимо отображение дополнительных данных. Эти данные могут также включать опции выбора. Таким образом, пользовательский интерфейс часто должен будет отображать свойства многочисленных экземпляров **АГРЕГАТОВ (10)**, несмотря на то что в большинстве случаев пользователь должен выполнять изменяющую состояние задачу, которая применяется только к одному экземпляру единственного типа АГРЕГАТА. Эта ситуация проиллюстрирована на рис. 14.2.

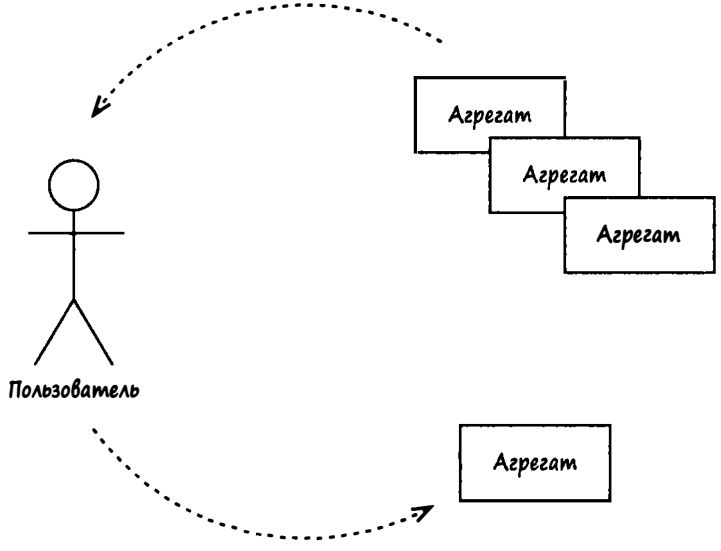


Рис. 14.2. Пользовательский интерфейс может отображать свойства многочисленных экземпляров АГРЕГАТА, но подавать запрос на изменение только одного экземпляра в каждый момент времени

Визуализация объектов передачи данных с помощью экземпляров АГРЕГАТА

Популярный способ решения проблемы визуализации многочисленных экземпляров АГРЕГАТОВ в единственном представлении состоит в использовании **ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ** [Фаулер, P of EAA], или **DTO** (Data Transfer Object). Объект DTO содержит все атрибуты, которые должны быть выведены на экран в представлении. ПРИКЛАДНАЯ СЛУЖБА (см. раздел "Прикладные службы") использует **ХРАНИЛИЩА (12)**, чтобы считывать необходимые экземпляры АГРЕГАТОВ и затем делегировать их **СБОРЩИКУ ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ (DTO ASSEMBLER)** [Фаулер, P of EAA] для отображения атрибутов DTO. Таким образом, DTO переносит полное дополнение информации, которая должна быть представлена на экране. Компонент пользовательского интерфейса имеет доступ к каждому атрибуту DTO и визуализирует его на представлении.

В этом подходе операции чтения и записи выполняются через ХРАНИЛИЩА. Он позволяет выполнять поиск в коллекциях с "ленивой" загрузкой, потому что СБОРЩИК ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ получает непосредственный доступ к каждой части АГРЕГАТА, которая необходима для создания объекта DTO. Он также решает специфичную проблему, которая заключается в том, что уровень представления физически отделен от бизнес-уровня, и вы должны сериализовать контейнеры данных и передавать их по сети на другой уровень.

Интересно, что шаблон DTO был первоначально разработан для работы с удаленным уровнем представлений, который использует экземпляры DTO. Объект DTO создается на бизнес-уровне, сериализируется, отправляется по сети и десериализируется на уровне представлений. Если уровень представлений не является удаленным, то этот шаблон часто создает ненужную сложность в проекте приложения подобно принципу YAGNI ("Вам это никогда не понадобится — You Ain't Gonna Need It"). В этом случае иногда приходится создавать классы, которые напоминают объекты предметной области, но не точно соответствуют им. Кроме того, в рамках этого подхода необходимо дополнительно создавать потенциально большие объекты, которыми должна управлять виртуальная машина (например, JVM), в то время как фактически они не соответствуют архитектуре приложения с единственной виртуальной машиной.

АГРЕГАТЫ необходимо разрабатывать так, чтобы СБОРЩИКИ ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ могли запрашивать необходимые данные. Серьезно подумайте о том, как показать состояние, не раскрывая слишком много информации о внутренней форме или структуре АГРЕГАТОВ. Попытайтесь устранить связь клиента со всеми внутренними деталями АГРЕГАТА. Следует ли позволить клиентам — в этом случае СБОРЩИКАМ — перемещаться глубоко внутри АГРЕГАТОВ? Это может быть плохой идеей, так как она сильно связывает каждый клиент с определенной реализацией АГРЕГАТА.

Использование посредника для публикации внутреннего состояния агрегата

Для устранения тесной связи между моделью и ее клиентами можно выбрать шаблон интерфейса **ПОСРЕДНИК (MEDIATOR)** [Гамма и др.] (он же — **ДВОЙНАЯ ДИСПЕТЧЕРИЗАЦИЯ (DOUBLE-DISPATCH)** и **ОБРАТНЫЙ ВЫЗОВ (CALLBACK)**), в котором АГРЕГАТ публикует свое внутреннее состояние. Клиенты могут реализовать интерфейс ПОСРЕДНИКА, передавая АГРЕГАТУ ссылку на объект реализации в качестве аргумента метода. Затем АГРЕГАТ может выполнить двойную диспетчеризацию управления ПОСРЕДНИКУ и опубликовать запрошенное состояние, не раскрывая свою форму и структуру. Трюк заключается в том, что интерфейс ПОСРЕДНИКА не связан со спецификацией представления, а настроен на визуализацию требуемого состояния АГРЕГАТА.

```
public class BacklogItem ... {
    ...
    public void provideBacklogItemInterest(
             BacklogItemInterest anInterest) {
       anInterest.informTenantId(this.tenantId().id());
       anInterest.informProductId(this.productId().id());
       anInterest.informBacklogItemId(this.backlogItemId().id());
       anInterest.informStory(this.story());
       anInterest.informSummary(this.summary());
       anInterest.informType(this.type().toString());
       ...
}

    public void provideTasksInterest(TasksInterest anInterest) {
       Set<Task> tasks = this.allTasks();
       anInterest.informTaskCount(tasks.size());
       for (Task task : tasks) {
            ...
       }
    }
    ...
}
```

Разнообразные провайдеры можно реализовать с помощью других классов почти так же, как **СУЩНОСТИ (5)** описывают способ делегирования проверки разным тестовым классам.

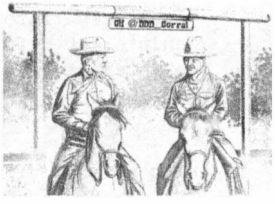
Следует помнить, что одни специалисты считают этот подход никак не связанным с АГРЕГАТАМИ. Другие считают его абсолютно естественным расширением хорошо разработанной модели предметной области. Как всегда, такие компромиссы следует обсудить со всеми техническими членами команды.

Визуализация экземпляров агрегата с помощью объекта полезных данных предметной области

Существует подход, позволяющий улучшить проект без использования объектов DTO. Он подразумевает собирание нескольких экземпляров АГРЕГАТА, предназначенных для визуализации, в одном **ОБЪЕКТЕ ПОЛЕЗНЫХ ДАННЫХ ПРЕДМЕТНОЙ ОБЛАСТИ (DPO — DOMAIN PAYLOAD OBJECT)** [Vernon, DPO]. Мотивы разработки объекта DPO похожи на мотивы создания объекта DTO, но его преимущество заключается в использовании архитектуры приложения с единственной виртуальной машиной. Такой объект содержит ссылки на экземпляры АГРЕГАТА в целом, а не на отдельные атрибуты. Кластеры экземпляров АГРЕГАТА можно передавать между логическими ярусами или уровнями как простой контейнерный объект ПОЛЕЗНЫХ ДАННЫХ (PAYLOAD). ПРИКЛАДНАЯ СЛУЖБА (см. раздел "Прикладные службы") использует ХРАНИЛИЩА для извлечения требуемых экземпляров АГРЕГАТА, а затем создает объект DPO, в котором хранятся ссылки на каждый из них. Компонент презентации запрашивает у объекта DPO ссылки на экземпляр АГРЕГАТА, а затем запрашивает у АГРЕГАТОВ атрибуты представления.

Ковбойская логика

LB: Если вы ни разу не падали с лошади, значит, еще мало занимались верховой ездой.



Преимущество этого подхода заключается в том, что он облегчает проектирование объектов для перемещения кластеров данных между логическими ярусами. Объекты DPO намного проще проектировать, к тому же они занимают меньше памяти. Поскольку экземпляры АГРЕГАТОВ в любом случае считываются в память, если они уже существуют, ситуация существенно упрощается.

Есть несколько потенциальных негативных последствий, заслуживающих внимания. Из-за схожести объектов DPO с объектами DTO этот подход требует наличия у АГРЕГАТОВ механизма для считывания их состояний. Для того чтобы избежать слишком сильной связи между пользовательским интерфейсом и моделью, в данном случае можно использовать ПОСРЕДНИКА, ДВОЙНУЮ ДИСПЕТЧЕРИЗАЦИЮ или интерфейс запросов КОРНЯ АГРЕГАТА, предложенные ранее для использования в СБОРЩИКЕ ОБЪЕКТОВ ПЕРЕДАЧИ ДАННЫХ.

Есть еще одна ситуация, которую необходимо рассмотреть. Так как объект DPO содержит ссылки на целые экземпляры АГГРЕГАТОВ, доступ к любым лениво загружаемым объектам и коллекциям пока невозможен. Нет никакой возможности

получить доступ ко всем свойствам АГРЕГАТА, необходимым для создания ОБЪЕКТА ПОЛЕЗНЫХ ДАННЫХ ПРЕДМЕТНОЙ ОБЛАСТИ. Так как после завершения метода ПРИКЛАДНОЙ СЛУЖБЫ обычно фиксируются даже транзакции, предназначенные только для чтения, любой компонент представления, содержащий ссылки на неопределенные лениво загружаемые объекты, вызовет исключение.[3]

Для того чтобы решить проблему с ленивыми загрузками, мы могли бы выбрать стратегию ранней загрузки или использовать **ПРЕОБРАЗОВАТЕЛЬ ЗАВИСИМОСТЕЙ ПРЕДМЕТНОЙ ОБЛАСТИ (DOMAIN DEPENDENCY RESOLVER)** [Vernon, DDR]. Это разновидность шаблона СТРАТЕГИЯ [Гамма и др.], в котором обычно используется по одной СТРАТЕГИИ на каждый поток сценария использования. Каждая СТРАТЕГИЯ обеспечивает доступ ко всем лениво загружаемым свойствам АГРЕГАТА, используемым определенным потоком сценария использования. Принудительный доступ происходит до того, как ПРИКЛАДНАЯ СЛУЖБА зафиксирует транзакцию и вернет объект DPO своему клиенту. Шаблон СТРАТЕГИЯ иногда трудно запрограммировать, чтобы вручную получить доступ к лениво загружаемым свойствам. Иногда он использует простой язык выражения, описывающий интроспективную и рефлексивную навигацию по экземплярам АГРЕГАТА. Робот, использующий рефлексивную навигацию, имеет одно преимущество: он может работать со скрытыми атрибутами. Однако вы можете удовлетвориться настройкой своих запросов на раннюю загрузку объектов, которые в обычных условиях загружаются лениво, если такая опция доступна.

Представления состояний экземпляров агрегата

Если ваше приложение предоставляет ресурсы, основанные на технологии **REST (4)**, то возникнет необходимость создать представления состояний объектов предметной области для клиентов. Очень важно создавать представления, основываясь на сценариях использования, а не на экземплярах АГРЕГАТОВ. Это очень похоже на мотивацию объектов DTO, которые также настроены на сценарии использования. Однако намного точнее интерпретировать ресурсы RESTful в виде отдельной самостоятельной модели — **МОДЕЛИ ПРЕДСТАВЛЕНИЯ (VIEW MODEL)**, или **МОДЕЛИ ПРЕЗЕНТАЦИИ (PRESENTATION MODEL)** [Fowler, PM]. Не поддавайтесь искушению создавать представления, которые являются однозначным отражением состояний АГРЕГАТА в модели предметной области, возможно, со ссылками на более глубокое состояние. Иначе вашим клиентам придется вникать в вашу модель предметной области и сами АГРЕГАТЫ. Клиентам придется

[3] Это напоминает использование шаблона ОТКРЫТЫЙ СЕАНС В ПРЕДСТАВЛЕНИИ (OPEN SESSION IN VIEW — OSIV) для управления транзакциями на уровне запросов и ответов в пользовательском интерфейсе. По разным причинам я считаю шаблон OSIV опасным, но вы можете со мной не соглашаться.

стать полностью осведомленными обо всех тонкостях поведения и изменениях состояния, и вы потеряете все преимущества абстракции.

Оптимальные запросы сценария использования

Вместо считывания нескольких целых экземпляров АГРЕГАТОВ разных типов и последующего программного создания отдельного контейнера (DTO или DPO) на их основе, можно использовать так называемый *оптимальный запрос сценария использования* (use case optimal query). В таком случае проектируется ХРАНИЛИЩЕ, имеющее методы поиска по запросу, которые создают требуемый объект как надмножество, состоящее из одного или нескольких экземпляров АГРЕГАТОВ. Этот запрос динамически помещает результаты в **ОБЪЕКТ-ЗНАЧЕНИЕ (6)**, специально разработанный для удовлетворения потребностей сценария использования. В этом случае проектируется ОБЪЕКТ-ЗНАЧЕНИЕ, а не DTO, потому что запрос зависит от предметной области, а не от приложения (как DTO). Затем оптимальный ОБЪЕКТ-ЗНАЧЕНИЕ сценария использования передается непосредственно механизму визуализации представления.

Мотивация подхода, основанного на оптимальном запросе сценария использования, похожа на мотивацию принципа **CQRS (4)**. Однако в отличие от принципа CQRS оптимальный запрос сценария использования — это запрос к ХРАНИЛИЩУ как унифицированному механизму постоянного хранения модели предметной области, а не простой запрос к базе данных (например, SQL) как отдельному хранилищу, предназначенному только для чтения. Для того чтобы лучше понять разницу между этими подходами, вернитесь к главе, посвященной **ХРАНИЛИЩАМ (12)**. Впрочем, встав на путь использования оптимальных запросов сценария использования, вы в той или иной степени будете применять принцип CQRS, так что имеет смысл продолжить чтение.

Работа с многочисленными и разнородными клиентами

Что делать, если приложение должно поддерживать работу с многочисленными и разнородными клиентами? К ним могут относиться насыщенные интернет-приложения RIA, "толстые" графические клиенты, службы REST и механизмы обмена сообщениями. Кроме того, такими клиентами могут быть тест-драйверы. Немного позже мы подробнее обсудим эту тему, а пока будем проектировать ПРИКЛАДНУЮ СЛУЖБУ, в которой каждый клиент задает свой тип **ПРЕОБРАЗОВАТЕЛЯ ДАННЫХ (DATA TRANSFORMER)**. Эта ПРИКЛАДНАЯ СЛУЖБА должна выполнять ДВОЙНУЮ ДИСПЕТЧЕРИЗАЦИЮ в зависимости от параметра ПРЕОБРАЗОВАТЕЛЯ ДАННЫХ, который определяет требуемый формат данных. Вот как может выглядеть пользовательский интерфейс клиента, основанного на архитектуре REST.

```
...
CalendarWeekData calendarWeekData =
    calendarAppService
       .calendarWeek(date, new CalendarWeekXMLDataTransformer());

Response response =
    Response.ok(calendarWeekData.value())
       .cacheControl(this.cacheControlFor(30)).build();

return response;
```

Метод `calendarWeek()` класса `CalendarApplicationService` получает экземпляр класса `Date` с указанным номером недели и реализацией интерфейса `CalendarWeekDataTransformer`. Выбранный механизм реализации представляет собой класс `CalendarWeekXMLDataTransformer`, создающий XML-документ в качестве представления состояния экземпляра класса `CalendarWeekData`. Метод `value()` в классе `CalendarWeekData` возвращает предпочтительный тип заданного формата данных, который в нашем случае является XML-документом типа `String`.

Этот пример демонстрирует преимущество *внедрения зависимости* в экземпляр ПРЕОБРАЗОВАТЕЛЯ ДАННЫХ. Эта зависимость жестко зашита в код, чтобы его было легче понять.

Ниже перечислены возможные механизмы реализации класса `CalendarWeekDataTransformer`.

- `CalendarWeekCSVDataTransformer`
- `CalendarWeekDPODataTransformer`
- `CalendarWeekDTODataTransformer`
- `CalendarWeekJSONDataTransformer`
- `CalendarWeekTextDataTransformer`
- `CalendarWeekXMLDataTransformer`

Альтернативный подход заключается в абстрагировании типов выходной информации приложения для разнородных клиентов. Этот вопрос будет обсуждаться позже, в разделе "Прикладные службы".

Адаптеры визуализации и реакция на пользовательские операции редактирования

Если данные предметной области просматриваются и редактируются пользователями, то можно использовать шаблоны, которые могут помочь разделить обязанности. Снова приходится констатировать, что существует слишком много

каркасов и способов работы с данными, чтобы рекомендовать какой-то один безошибочный вариант. Некоторые каркасы пользовательского интерфейса вынуждают придерживаться определенных шаблонов, которые они поддерживают. Иногда шаблоны оказываются удачными, а иногда не очень. Одни шаблоны обеспечивают повышенную гибкость, а другие нет.

В каком бы виде вы ни получали данные предметной области от ПРИКЛАДНЫХ СЛУЖБ — в виде объектов DTO, DPO или представлений состояния — и какой бы каркас вы ни использовали, всегда можно с выгодой применить МОДЕЛЬ ПРЕЗЕНТАЦИИ (PRESENTATION MODEL).[4] Его цель — разделить ответственность между презентацией и представлением. Сейчас это возможно при работе с приложениями Web 1.0, но я думаю, что его мощь только увеличится при работе с приложениями Web 2.0 RIA или настольными клиентами (вторая и третья категории в списке, приведенном ранее).

Используя этот шаблон, мы хотим сделать представления пассивными, т.е. чтобы они только управляли отображением данных и элементами пользовательского интерфейса и что-нибудь еще. Есть два возможных способа визуализации представлений.

1. Представления сами визуализируют себя на основе МОДЕЛИ ПРЕЗЕНТАЦИИ. Я думаю, что это наиболее естественный способ, который исключает связь между МОДЕЛЬЮ ПРЕЗЕНТАЦИИ и ПРЕДСТАВЛЕНИЕМ.
2. Представления визуализируются МОДЕЛЬЮ ПРЕЗЕНТАЦИИ. Этот способ легко тестировать, но он требует тесной связи между МОДЕЛЬЮ ПРЕЗЕНТАЦИИ и представлением.

МОДЕЛЬ ПРЕЗЕНТАЦИИ действует как **АДАПТЕР** [Гамма и др.]. Она маскирует детали модели предметной области, предоставляя свойства и поведенческие функции, которые ориентированы на представление. Это значит, что она представляет собой нечто большее, чем тонкую оболочку вокруг атрибутов объектов предметной области или DTO. Это значит также, что решения, принимаемые в АДАПТЕРЕ, основаны на состоянии модели, которое относится к представлению. Например, добавление конкретного управляющего элемента в представлении может не иметь прямого отношения к модели предметной области, но может быть выведено из нее. Вместо требования, чтобы модель предметной области специально поддерживала необходимые свойства представления, обязанность вывести из состояния модели предметной области индикаторы и свойства, специфичные для представления, возлагается на МОДЕЛЬ ПРЕЗЕНТАЦИИ.

---

[4] См. также шаблон МОДЕЛЬ-ПРЕДСТАВЛЕНИЕ-ПРЕЗЕНТАЦИЯ (MODEL-VIEW-PRESENTER) [Dolphin], которую Фаулер [Fowler, PM] называет КООРДИНИРУЮЩИМ КОНТРОЛЛЕРОМ (SUPERVISING CONTROLLER) и ПАССИВНЫМ ПРЕДСТАВЛЕНИЕМ (PASSIVE VIEW).

Еще одно, возможно, менее заметное преимущество МОДЕЛИ ПРЕЗЕНТАЦИИ, состоит в том, что она может адаптировать АГРЕГАТЫ, не поддерживающие интерфейс методов получателей (get) JavaBean, для каркасов интерфейсов, в которых необходимо использовать методы получатели. Многие, если не все, веб-каркасы, основанные на языке Java, требуют, чтобы объекты имели открытые методы получатели, такие как `getSummary()` и `getStory()`, в то время как проект модели предметной области поощряет использование предметно-ориентированных выражений, тесно связанных с **ЕДИНЫМ ЯЗЫКОМ (1)**. Разница может быть простой и выражаться в виде методов `summary()` и `story()`, но при этом возникает рассогласование каркаса пользовательского интерфейса. С помощью МОДЕЛИ ПРЕЗЕНТАЦИИ можно просто превратить метод `summary()` в метод `getSummary()`, а метод `story()` — в метод `getStory()`, исключив разногласия между моделью и представлением.

```
public class BacklogItemPresentationModel
       extends AbstractPresentationModel {

    private BacklogItem backlogItem;

    public BacklogItemPresentationModel(BacklogItem aBacklogItem) {
        super();
        this.backlogItem = backlogItem;
    }

    public String getSummary() {
        return this.backlogItem.summary();
    }

    public String getStory() {
        return this.backlogItem.story();
    }
    ...
}
```

Конечно, МОДЕЛЬ ПРЕЗЕНТАЦИИ может адаптировать любое количество ранее рассмотренных подходов, включая использование объектов DTO или DPO, или шаблона ПОСРЕДНИК, публикующего внутреннее состояние агрегата.

Кроме того, операции редактирования, выполняемые пользователем, отслеживаются МОДЕЛЬЮ ПРЕЗЕНТАЦИИ. Это не означает перегрузку МОДЕЛИ ПРЕЗЕНТАЦИИ обязанностями, поскольку адаптация осуществляется в обоих направлениях — от модели к представлению и от представления к модели.

Следует помнить, что МОДЕЛЬ ПРЕЗЕНТАЦИИ не совпадает с тяжеловесным шаблоном **ФАСАД** [Гамма и др.], развернутым вокруг ПРИКЛАДНЫХ СЛУЖБ или модели предметной области. К счастью, после того как пользователи завершают задачу с помощью пользовательского интерфейса, они обычно выполняют

действия типа "применить" или "отмена" или какую-то явную команду. Для этого МОДЕЛЬ ПРЕЗЕНТАЦИИ должна отражать действие пользователя в приложении, что по существу совпадает с минимальным ФАСАДОМ, развернутым вокруг ПРИКЛАДНОЙ СЛУЖБЫ.

```
public class BacklogItemPresentationModel
    extends AbstractPresentationModel {

    private BacklogItem backlogItem;
    private BacklogItemEditTracker editTracker;
    // инъекция
    private BacklogItemApplicationService backlogItemAppService;

    public BacklogItemPresentationModel(BacklogItem aBacklogItem) {
        super();
        this.backlogItem = backlogItem;
        this.editTracker = new BacklogItemEditTracker(aBacklogItem);
    }
    ...
    public void changeSummaryWithType() {
        this.backlogItemAppService
            .changeSummaryWithType(
                  this.editTracker.summary(),
                  this.editTracker.type());
    }
    ...
}
```

Пользователь щелкает на командной кнопке, расположенной на представлении, инициируя вызов метода `changeSummaryWithType()`. Ответственность за взаимодействие с ПРИКЛАДНОЙ СЛУЖБОЙ и реакцию на операции редактирования, возникшие в объекте `editTracker`, возложена на класс `BacklogItemPresentationModel`. Других обработчиков операции редактирования нет. Таким образом, можно сказать, что МОДЕЛЬ ПРЕЗЕНТАЦИИ — это минимальный ФАСАД, возведенный перед ПРИКЛАДНЫМИ СЛУЖБАМИ со стороны представления, но только потому, что метод `changeSummaryWithType()` является интерфейсом более высокого уровня, который делает класс `BacklogItemApplicationService` более легким в использовании. Однако не хотелось бы, чтобы несколько строк кода в классе МОДЕЛИ ПРЕЗЕНТАЦИИ управляли детализированным использованием ПРИКЛАДНОЙ СЛУЖБЫ, или, что еще хуже, сами действовали, как ПРИКЛАДНАЯ СЛУЖБА по отношению к модели предметной области. Это выходило бы за пределы ответственности МОДЕЛИ ПРЕЗЕНТАЦИИ. Вместо этого мы хотим видеть простое делегирование операции более сложному и тяжеловесному ФАСАДУ `BacklogItemApplicationService`.

Мы описали мощный подход к координированию модели предметной области и пользовательского интерфейса. Возможно, его даже можно назвать универсальным шаблоном управления пользовательским интерфейсом. Однако, используя любой из методов управления представлением, мы все еще часто взаимодействуем с интерфейсом программирования ПРИКЛАДНЫХ СЛУЖБ.

Прикладные службы

В некоторых случаях пользовательский интерфейс агрегирует многочисленные **ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (2)**, использующие независимые компоненты МОДЕЛИ ПРЕЗЕНТАЦИИ, собранные в единственном представлении. Независимо от того, является ли ваш пользовательский интерфейс единственной моделью или состоит из многоуровневых моделей, он, вероятно, будет взаимодействовать с ПРИКЛАДНЫМИ СЛУЖБАМИ, поэтому целесообразно их рассмотреть.

ПРИКЛАДНЫЕ СЛУЖБЫ — непосредственные клиенты модели предметной области. Возможное расположение ПРИКЛАДНЫХ СЛУЖБ с логической точки зрения зависит от **АРХИТЕКТУРЫ (4)**. Они ответственны за координацию потоков сценария использования, при этом один метод службы отвечает за один поток. При использовании базы данных ACID ПРИКЛАДНЫЕ СЛУЖБЫ также контролируют транзакции, гарантируя атомарное сохранение переходов между состояниями модели. Управление транзакциями в этом разделе обсуждается мимоходом, а основное внимание уделяется **ХРАНИЛИЩАМ (12)**. Кроме того, обычно ПРИКЛАДНЫЕ СЛУЖБЫ решают задачи безопасности.

Не следует путать ПРИКЛАДНЫЕ СЛУЖБЫ со **СЛУЖБАМИ ПРЕДМЕТНОЙ ОБЛАСТИ (7)**. Это разные службы. Явный контраст между ними четко демонстрируется в следующем разделе. Мы должны стремиться перенести всю бизнес-логику предметной области в модель предметной области в виде АГРЕГАТОВ, ОБЪЕКТОВ-ЗНАЧЕНИЙ или СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ. *Сохраняйте ПРИКЛАДНЫЕ СЛУЖБЫ тонкими, используя их только для координации задач в модели.*

Пример прикладной службы

Рассмотрим пример интерфейса и реализации ПРИКЛАДНОЙ СЛУЖБЫ, управляющей задачами сценария использования для арендаторов в *Контексте идентификации и доступа*. Это простой пример и его не следует считать идеалом. Компромиссы, принятые в этом примере, станут очевидны после его анализа.

Сначала рассмотрим основной интерфейс.

```
package com.saasovation.identityaccess.application;

public interface TenantIdentityService {

    public void activateTenant(TenantId aTenantId);

    public void deactivateTenant(TenantId aTenantId);

    public String offerLimitedRegistrationInvitation(
            TenantId aTenantId,
            Date aStartsOnDate,
            Date anUntilDate);

    public String offerOpenEndedRegistrationInvitation(
            TenantId aTenantId);

    public Tenant provisionTenant(
            String aTenantName,
            String aTenantDescription,
            boolean isActive,
            FullName anAdministratorName,
            EmailAddress anEmailAddress,
            PostalAddress aPostalAddress,
            Telephone aPrimaryTelephone,
            Telephone aSecondaryTelephone,
            String aTimeZone);

    public Tenant tenant(TenantId aTenantId);
    ...
}
```

Эти шесть методов ПРИКЛАДНОЙ СЛУЖБЫ используются для создания или настройки нового арендатора, активизации и деактивизации существующего арендатора, предложения ограниченных и открытых регистрационных приглашений будущим пользователям и выполнения запроса для определенного арендатора.

В сигнатурах этих методов используются некоторые типы из модели предметной области. Для этого необходимо, чтобы пользовательский интерфейс знал об этих типах и зависел от них. Иногда ПРИКЛАДНЫЕ СЛУЖБЫ разрабатываются для того, чтобы полностью экранировать пользовательский интерфейс от всех таких знаний о проблемной области. В таком случае в сигнатурах методов ПРИКЛАДНОЙ СЛУЖБЫ используются только элементарные типы примитивов (`int`, `long`, `double`), `Strings` и, возможно, объекты DTO. В качестве альтернативы этим подходам иногда лучше разрабатывать объекты по шаблону **КОМАНДА (COMMAND)** [Гамма и др.]. Трудно сказать, какой из этих вариантов правильный, а какой неправильный. Все зависит от ваших предпочтений и целей. В этой книге в различных примерах представлен каждый из этих стилей.

Рассмотрим упомянутые компромиссы. Если вы устраняете типы из модели, то избегаете зависимостей и связей, но не можете обеспечить строгую проверку типов и их допустимости, т.е. защиту, которую автоматически предоставляют типы ОБЪЕКТОВ-ЗНАЧЕНИЙ. Если вы не раскрываете объекты предметной области в виде возвращаемых типов, то должны будете создать объекты DTO. Если вы создаете объекты DTO, то может возникнуть излишняя сложность решения из-за издержек, связанных с дополнительными типами. В таком случае в приложениях, связанных с интенсивным трафиком, вызванным ненужными объектами DTO, которые постоянно создаются и удаляются, наблюдается чрезмерный расход памяти.

Конечно, если вы предоставите объекты предметной области разнородным клиентам, то каждый клиентский тип должен будет иметь дело с ними по отдельности. Связь усиливается и при большом количестве клиентов становится проблемой. Учитывая это, по крайней мере несколько из указанных методов можно было бы улучшить, чтобы ввести возвращаемые типы. Как указывалось ранее, мы могли бы вместо этого использовать ПРЕОБРАЗОВАТЕЛИ ДАННЫХ.

```
package com.saasovation.identityaccess.application;

public interface TenantIdentityService {
    ...
    public TenantData provisionTenant(
            String aTenantName,
            String aTenantDescription,
            boolean isActive,
            FullName anAdministratorName,
            EmailAddress anEmailAddress,
            PostalAddress aPostalAddress,
            Telephone aPrimaryTelephone,
            Telephone aSecondaryTelephone,
            String aTimeZone,
            TenantDataTransformer aDataTransformer);

    public TenantData tenant(
            TenantId aTenantId,
            TenantDataTransformer aDataTransformer);

}
```

Пока я предпочитаю раскрыть объекты предметной области клиенту и предполагаю, что у нас есть только один пользовательский веб-интерфейс. Это позволит упростить пример. Позже мы вернемся к подходу, основанному на ПРЕОБРАЗОВАТЕЛЕ ДАННЫХ.

Рассмотрим реализацию ПРИКЛАДНОЙ СЛУЖБЫ. Анализ нескольких более простых методов позволяет осветить несколько важных моментов. Отметим,

что в данном случае применение **ВЫДЕЛЕННОГО ИНТЕРФЕЙСА (SEPARATED INTERFACE)** [Fowler, P of EAA] ничего не даст. Итак, перейдем к примеру, в котором приводится определение интерфейса с классом реализации.

```
package com.saasovation.identityaccess.application;

public class TenantIdentityService {

    @Transactional
    public void activateTenant(TenantId aTenantId) {
        this.nonNullTenant(aTenantId).activate();
    }

    @Transactional
    public void deactivateTenant(TenantId aTenantId) {
        this.nonNullTenant(aTenantId).deactivate();
    }



    @Transactional(readOnly=true)
    public Tenant tenant(TenantId aTenantId) {
        Tenant tenant =
            this
                .tenantRepository()
                .tenantOfId(aTenantId);

        return tenant;
    }

    private Tenant nonNullTenant(TenantId aTenantId) {
        Tenant tenant = this.tenant(aTenantId);

        if (tenant == null) {
            throw new IllegalArgumentException(
                    "Tenant does not exist.");
        }

        return tenant;
    }
}
```

Клиент отправляет запрос на деактивацию существующего экземпляра класса `Tenant` с помощью метода `deactivateTenant()`. Для взаимодействия с реальным объектом класса `Tenant` необходимо найти его в ХРАНИЛИЩЕ по соответствующему идентификатору `TenantId`. Мы создали вспомогательный метод `nonNullTenant()`, который выполняет самоделегирование методу `tenant()`. Этот метод предназначен для защиты от несуществующих экземпляров класса `Tenant` и используется всеми служебными методами, которые должны получать существующий объект класса `Tenant`.

Методы `activateTenant()` и `deactivateTenant()` помечены как транзакционные методы записи с помощью аннотации `Transactional` каркаса Spring. Метод `tenant()` помечен как транзакционный метод, предназначенный только для чтения. Во всех трех случаях, когда клиент получает этот пакет в контексте Spring и вызывает служебный метод, начинается транзакция. Когда метод нормально завершает работу и возвращает результат, транзакция фиксируется. В зависимости от конфигурации в области видимости метода может генерироваться исключение, вынуждающее откат транзакции.

Но как же предотвратить неправильное использование этих методов, например злонамеренное? Когда мы говорим о деактивизации и реактивизации арендатора, то имеем в виду операцию, которую может выполнять только авторизованный пользователь системы SaaSOvation или новый подписчик арендатора.

Что если использовать функции системы Spring Security? Можно использовать другую аннотацию, `PreAuthorize`.

```
public class TenantIdentityService {

   @Transactional
   @PreAuthorize("hasRole('SubscriberRepresentative')")
   public void activateTenant(TenantId aTenantId) {
       this.nonNullTenant(aTenantId).activate();
   }

   @Transactional
   @PreAuthorize("hasRole('SubscriberRepresentative')")
   public void deactivateTenant(TenantId aTenantId) {
       this.nonNullTenant(aTenantId).deactivate();
   }



   @Transactional
   @PreAuthorize("hasRole('SubscriberRepresentative')")
   public Tenant provisionTenant(
           String aTenantName,
           String aTenantDescription,
           boolean isActive,
           FullName anAdministratorName,
           EmailAddress anEmailAddress,
           PostalAddress aPostalAddress,
           Telephone aPrimaryTelephone,
           Telephone aSecondaryTelephone,
           String aTimeZone) {

   return
           this
                 .tenantProvisioningService
                 .provisionTenant(
                      aTenantName,
```

```
                    aTenantDescription,
                    isActive,
                    anAdministratorName,
                    anEmailAddress,
                    aPostalAddress,
                    aPrimaryTelephone,
                    aSecondaryTelephone,
                    aTimeZone);
    }
    ...
}
```

Это объявление прав доступа на уровне метода не позволяет неавторизованным пользователям обращаться к ПРИКЛАДНЫМ СЛУЖБАМ. Конечно, пользовательский интерфейс в принципе должен разрабатываться так, чтобы скрыть любой навигационный доступ к таким средствам, если пользователь не был авторизован, но остановить злонамеренную атаку можно только с помощью обеспечения декларативной безопасности метода.

Декларативная безопасность на уровне метода отличается от того, что обеспечивает система IdOvation. Сотрудники компании SaaSOvation должны входить в систему IdOvation иначе, чем пользователи арендатора. Пользователи, играющие специальную роль `SubscriberRepresentative`, могут получить разрешение на выполнение этих особенных методов, а подписчики ни при каких обстоятельствах не должны получать такое разрешение. Это, конечно, потребовало бы интеграции между системами Spring Security и IdOvation.

Теперь, рассмотрев реализацию метода `provisionTenant()`, мы видим, что он выполняет делегирование задачи СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ. Это обстоятельство подчеркивает различие между этими двумя видами служб, особенно в предметной области `TenantProvisioningService`. Значительная часть логики предметной области размещена в этой СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ и лишь незначительная часть — в ПРИКЛАДНОЙ СЛУЖБЕ. Рассмотрим, что делает эта СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ (несмотря на то, что я не представляю здесь ее код).

1. Создает новый агрегат типа `Tenant` и добавляет его в ХРАНИЛИЩЕ.
2. Назначает нового администратора для нового экземпляра класса `Tenant`. Это подразумевает предоставление роли АДМИНИСТРАТОР для нового объекта класса `Tenant` и публикацию СОБЫТИЯ `TenantAdministratorRegistered`.
3. Публикует СОБЫТИЕ `TenantProvisioned`.

Если бы ПРИКЛАДНАЯ СЛУЖБА должна была сделать больше, чем указано в п. 1, то мы допустили бы серьезную утечку информации о логике предметной области из модели. Поскольку есть два дополнительных пункта, которые не

относятся к ответственности ПРИКЛАДНОЙ СЛУЖБЫ, мы включаем все три пункта во внутреннюю часть СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Используя СЛУЖБУ ПРЕДМЕТНОЙ ОБЛАСТИ, мы добавляем этот "существенно важный процесс... в предметную область" [Эванс].[5] Мы также должным образом следуем определению ПРИКЛАДНОЙ СЛУЖБЫ, управляя транзакцией, безопасностью и задачей делегирования этого важного процесса настройки арендатора по модели.

Однако представьте себе на мгновение шум, вызванный списком параметров метода `provisionTenant()`. Список содержит в общей сложности девять параметров, что по меньшей мере многовато. Эту ситуацию можно предотвратить с помощью объектов простой **КОМАНДЫ** [Гамма и др.]: "Инкапсулируйте запрос в виде объекта, тем самым позволяя пользователям параметризовать клиентов с различными запросами, поставьте в очередь или зарегистрируйте в журнале регистрации запросы, а также организуйте поддержку операций, которые можно отменить". Другими словами, мы могли бы думать об объекте КОМАНДЫ как о сериализованном вызове метода, и в нашем случае мы заинтересованы во всех функциях КОМАНДЫ, за исключением операции отмены. Рассмотрим простой класс КОМАНДЫ.

```
public class ProvisionTenantCommand {
    private String tenantName;
    private String tenantDescription;
    private boolean isActive;
    private String administratorFirstName;
    private String administratorLastName;
    private String emailAddress;
    private String primaryTelephone;
    private String secondaryTelephone;
    private String addressStreetAddress;
    private String addressCity;
    private String addressStateProvince;
    private String addressPostalCode;
    private String addressCountryCode;
    private String timeZone;

    public ProvisionTenantCommand(...) {
        ...
    }

    public ProvisionTenantCommand() {
        super();
    }

    public String getTenantName() {
        return tenantName;
    }
```

[5] См. главу 7.

```
    public void setTenantName(String tenantName) {
        this.tenantName = tenantName;
    }
    ...
}
```

Класс ProvisionTenantCommand использует только элементарные типы, а не объекты модели. В нем есть конструктор с несколькими аргументами, а также конструктор без аргументов. Наличие конструктора без аргументов и открытых методов установщиков позволяет заполнить КОМАНДУ отображениями "поле формы – объект" (например, свойствами JavaBean или .NET CLR). Может показаться, что КОМАНДА — это объект DTO, но на самом деле она представляет собой нечто намного большее. Поскольку объект КОМАНДЫ называется именем операции, для которой он предназначен, он носит более явный характер. Экземпляр КОМАНДЫ можно передать методу ПРИКЛАДНОЙ СЛУЖБЫ.

```
public class TenantIdentityService {
    ...

    @Transactional
    public String provisionTenant(ProvisionTenantCommand aCommand) {
       ...
       return tenant.tenantId().id();
    }
    ...
}
```

Помимо этого подхода к пересылке КОМАНДЫ методу API ПРИКЛАДНОЙ СЛУЖБЫ в соответствии с шаблоном, мы могли бы вместо этого или в дополнение к этому отправлять КОМАНДЫ в очередь, чтобы выполнить их впоследствии с помощью ОБРАБОТЧИКА КОМАНД. Можно считать, что ОБРАБОТЧИК КОМАНД семантически эквивалентен методу ПРИКЛАДНОЙ СЛУЖБЫ, но временно отделенному от нее. Как будет показано в приложении, это обеспечивает более высокую пропускную способность и масштабируемость обработки КОМАНД.

Не связанный вывод службы

Ранее я уже несколько раз обсуждал использование ПРЕОБРАЗОВАТЕЛЕЙ ДАННЫХ как способ снабдить разнородные клиентские типы определенным типом данных, в котором они нуждаются. Этот подход использует ПРЕОБРАЗОВАТЕЛИ, которые приводят данные к определенному типу, реализующему абстрактный интерфейс, совместно использующему все связанные типы. С точки зрения клиента это может выглядеть следующим образом.

```
TenantData tenantData =
    tenantIdentityService.provisionTenant(
        ..., myTenantDataTransformer);

TenantPresentationModel tenantPresentationModel =
    new TenantPresentationModel(tenantData.value());
```

ПРИКЛАДНЫЕ СЛУЖБЫ разрабатываются как API с вводом и выводом. Цель передачи управления в ПРЕОБРАЗОВАТЕЛЬ ДАННЫХ состоит в том, чтобы создать определенный выходной тип, необходимый клиенту.

А что если взять совершенно другой курс и принять правило, что ПРИКЛАДНЫЕ СЛУЖБЫ всегда объявляются пустыми и, таким образом, никогда не возвращают данные клиентам? Как это работало бы? Ответ заключается в идее, лежащей в основе **ГЕКСАГОНАЛЬНОЙ АРХИТЕКТУРЫ (4)** — в использовании стиля ПОРТЫ И АДАПТЕРЫ. В этом экземпляре мы использовали бы единственный ПОРТ стандартного вывода с любым количеством адаптеров, по одному для каждого клиентского типа. В таком случае мы могли бы создать метод `provisionTenant()` ПРИКЛАДНОЙ СЛУЖБЫ, показанный ниже.

```
public class TenantIdentityService {
    ...

    @Transactional
    @PreAuthorize("hasRole('SubscriberRepresentative')")
    public void provisionTenant(
            String aTenantName,
            String aTenantDescription,
            boolean isActive,
            FullName anAdministratorName,
            EmailAddress anEmailAddress,
            PostalAddress aPostalAddress,
            Telephone aPrimaryTelephone,
            Telephone aSecondaryTelephone,
            String aTimeZone) {

       Tenant tenant =
            this
                .tenantProvisioningService
                .provisionTenant(

                      aTenantName,
                      aTenantDescription,
                      isActive,
                      anAdministratorName,
                      anEmailAddress,
                      aPostalAddress,
                      aPrimaryTelephone,
                      aSecondaryTelephone,
                      aTimeZone);
```

```
        this.tenantIdentityOutputPort().write(tenant);
    }
    ...
}
```

ПОРТ вывода в этой архитектуре — это определенный именованный ПОРТ на краю схемы приложения. Если используется система Spring, то он представляет собой пакет, внедренный в службу. Единственная вещь, которую должен знать метод `provisionTenant()`, состоит в том, что он должен с помощью метода `write()` записать в ПОРТ экземпляр класса `Tenant`, который он получает от СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ. Этот ПОРТ может иметь любое количество читателей, которые регистрируются перед использованием ПРИКЛАДНОЙ СЛУЖБЫ. После выполнения метода `write()` каждый из зарегистрированных читателей получает сигнал о том, что он должен считать выходную информацию, как свою входную информацию. В этой точке читатели могут преобразовать вывод, используя установленный механизм, такой как ПРЕОБРАЗОВАТЕЛЬ ДАННЫХ.

Эта схема — не просто необычное изобретение, усложняющее архитектуру. По силе она не отличается от архитектуры ПОРТЫ И АДАПТЕРЫ как для систем программного обеспечения, так и для аппаратных устройств. Каждый компонент должен лишь понимать считанную информацию, реализовать собственное поведение и знать ПОРТ, в который он пишет выходную информацию.

Запись в ПОРТ — это примерно то, что делает чисто командный метод АГРЕГАТА, не возвращающий никаких значений, но публикующий **СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ (8)**. В случае АГРЕГАТА **ИЗДАТЕЛЕМ СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ (8)** является ПОРТ вывода. Далее, если мы решим запросить состояние АГРЕГАТА с помощью ДВОЙНОЙ ДИСПЕТЧЕРИЗАЦИИ, выполняемой ПОСРЕДНИКОМ, возникнет схема, похожая на ПОРТЫ И АДАПТЕРЫ.

Недостатком подхода ПОРТЫ И АДАПТЕРЫ является то, что он может затруднить выбор имен методов ПРИКЛАДНОЙ СЛУЖБЫ, выполняющих запросы. Рассмотрите метод `tenant()` из примера службы. Теперь это имя кажется неподходящим, потому что метод больше не возвращает значения АРЕНДАТОРУ, который его запрашивает. Имя `provisionTenant()` все еще означает настройку API, потому что этот метод фактически становится чистым командным методом, больше не возвращая значение. Но мы могли бы выбрать более точное имя для метода `tenant()`. Посмотрим, как это можно сделать.

```
...
    @Override
    @Transactional(readOnly=true)
    public void findTenant(TenantId aTenantId) {
        Tenant tenant =
            this
                .tenantRepository
                .tenantOfId(aTenantId);
```

```
        this.tenantIdentityOutputPort().write(tenant);
    }
    ...
}
```

Имя `findTenant()` может оказаться подходящим, потому что поиск не обязательно означает возвращение его результата. Какое бы имя ни было выбрано, эта ситуация подтверждает, что каждое архитектурное решение может иметь как положительные, так и отрицательные последствия.

Компоновка многочисленных ОГРАНИЧЕННЫХ КОНТЕКСТОВ

Примеры, приведенные выше, не учитывают возможность, что единственный пользовательский интерфейс может объединять две или более моделей предметной области. В моих примерах понятия из вышележащих моделей интегрированы в нижележащие модели с помощью их трансляции в термины нижележащей модели.

Эта ситуация отличается от необходимости соединять многоуровневые модели в одно объединенное представление, как показано на рис. 14.3. Внешними моделями в этом примере являются *Контекст продуктов*, *Контекст Обсуждений* и *Контекст Обзоров*. Пользовательский интерфейс не должен знать, из чего состоят многоуровневые модели. Когда аналогичная ситуация возникает в вашем приложении, вы должны задуматься о структуре **МОДУЛЯ (9)**, назвать требуемые функции поддержки и указать, как ПРИКЛАДНЫЕ СЛУЖБЫ могут сгладить вероятный разрыв между различными моделями.

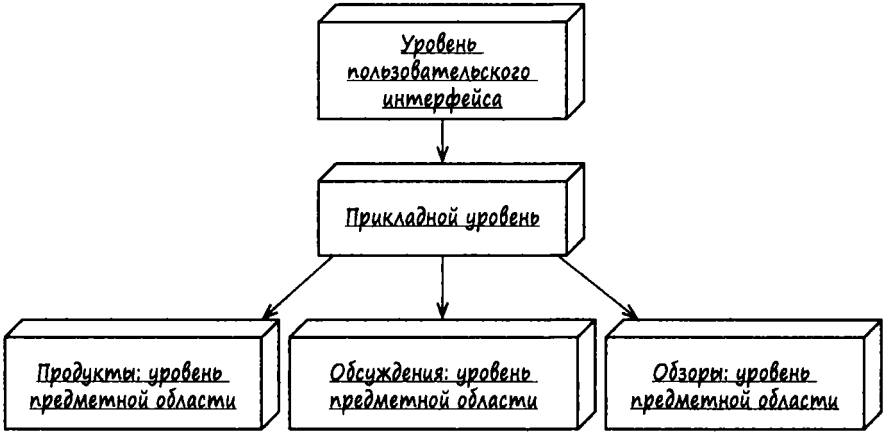


Рис. 14.3. Иногда пользовательский интерфейс должен объединять несколько моделей. Здесь показаны три модели, объединенные одним ПРИКЛАДНЫМ УРОВНЕМ

В качестве одного из решений можно использовать многочисленные ПРИКЛАДНЫЕ УРОВНИ, но это не похоже на ситуацию, продемонстрированную на рис. 14.3. Работая с многочисленными ПРИКЛАДНЫМИ УРОВНЯМИ, вы должны были бы предоставить каждому из них независимые компоненты пользовательского интерфейса, причем эти компоненты должны иметь определенную связь с базовой моделью предметной области. По существу, это стиль "портал–портлет". Однако еще более трудно согласовать разнородные ПРИКЛАДНЫЕ УРОВНИ и независимые компоненты пользовательского интерфейса с потоками сценария использования, для чего, собственно, и предназначен пользовательский интерфейс.

Поскольку сценариями использования управляет ПРИКЛАДНОЙ УРОВЕНЬ, возможно, легче всего было бы осуществить композицию моделей с помощью единственного ПРИКЛАДНОГО УРОВНЯ, как показано на рис. 14.3. Службы в этом единственном слое не содержат бизнес-логику, а лишь предоставляют услуги объектам АГРЕГАТА в каждой из связанных моделей в соответствии с требованиями пользовательского интерфейса. Вероятно, в этом случае вы назвали бы МОДУЛИ в ПОЛЬЗОВАТЕЛЬСКОМ ИНТЕРФЕЙСЕ и на ПРИКЛАДНЫХ УРОВНЯХ в соответствии с их предназначением и контекстом.

```
com.consumerhive.productreviews.presentation
com.consumerhive.productreviews.application
```

ГРУППА ПОТРЕБИТЕЛЕЙ (CONSUMER HIVE) предоставляет обзоры потребительского товара и их обсуждения. *Контекст продуктов* отделен от *Контекста Обсуждений* и *Контекста Обзоров*. МОДУЛИ представления и применения объединены под одним пользовательским интерфейсом. Вероятно, потребитель получает свой каталог продукции от одного или более внешних источников, тогда как обсуждения и обзоры — его СМЫСЛОВОЕ ЯДРО.

Кстати, о СМЫСЛОВОМ ЯДРЕ... Не странно ли обнаружить его здесь? Не служит ли этот ПРИКЛАДНОЙ УРОВЕНЬ новой моделью области со встроенным **ПРЕДОХРАНИТЕЛЬНЫМ УРОВНЕМ (3)**? Да, это новый упрощенный ОГРАНИЧЕННЫЙ КОНТЕКСТ. Здесь ПРИКЛАДНЫЕ СЛУЖБЫ управляют слиянием различных объектов DTO, которые имитируют своего рода **АНЕМИЧНУЮ МОДЕЛЬ ПРЕДМЕТНОЙ ОБЛАСТИ (1)**. Это своего рода **СЦЕНАРИЙ ТРАНЗАКЦИЙ (1)**, моделирующий СМЫСЛОВОЕ ЯДРО.

Если бы вы пришли к выводу, что ГРУППА ПОТРЕБИТЕЛЕЙ, состоящая из трех моделей, требует новой **МОДЕЛИ ПРЕДМЕТНОЙ ОБЛАСТИ (1)**, которая является объединенной объектной моделью в единственном ОГРАНИЧЕННОМ КОНТЕКСТЕ, то вы могли бы назвать МОДУЛИ новой модели следующим образом.

```
com.consumerhive.productreviews.domain.model.product

com.consumerhive.productreviews.domain.model.discussion

com.consumerhive.productreviews.domain.model.review
```

В конце вы должны будете решить, как моделировать эту ситуацию. Использовать ли стратегический или даже тактический шаблон, чтобы создать новую модель? Как минимум эта ситуация порождает вопрос "Где мы проводим черту между объединением многочисленных ОГРАНИЧЕННЫХ КОНТЕКСТОВ в единственный пользовательский интерфейс и созданием нового, ясного ОГРАНИЧЕННОГО КОНТЕКСТА с объединенной моделью предметной области? Каждый случай нужно рассматривать отдельно. У менее важной системы были бы другие цели и приоритеты. Однако мы не должны рассматривать такие решения произвольно. Следует учесть критерии, установленные в ОГРАНИЧЕННЫХ КОНТЕКСТАХ. В конце лучший подход — тот, который приносит наибольшую пользу бизнесу.

Инфраструктура

Предназначение инфраструктуры — предоставлять технические средства остальным частям вашего приложения. Мы не будем обсуждать **УРОВНИ (4)**, но все же напомним, что полезно придерживаться ПРИНЦИПА ИНВЕРСИИ ЗАВИСИМОСТИ. Независимо от места инфраструктуры в архитектуре, она считается очень хорошо спроектированной, если ее компоненты зависят от пользовательского интерфейса, ПРИКЛАДНЫХ СЛУЖБ и модели предметной области, требующих специальных технических возможностей. Таким образом, когда ПРИКЛАДНАЯ СЛУЖБА ищет ХРАНИЛИЩЕ, она зависит только от его интерфейса из предметной области, используя его реализацию из инфраструктуры. Соответствующая статическая структурная диаграмма на языке UML представлена на рис. 14.4.

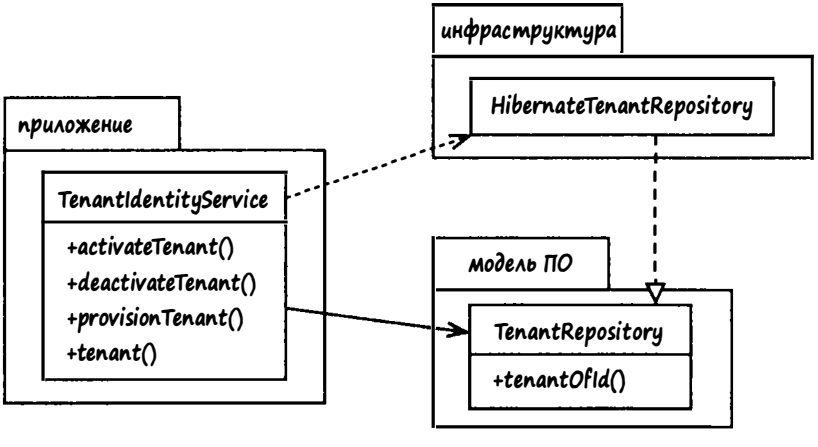


Рис. 14.4. ПРИКЛАДНАЯ СЛУЖБА зависит от интерфейса ХРАНИЛИЩА из предметной области, но использует класс реализации из инфраструктуры. Обязанности инкапсулированы в пакетах

Поиск в ХРАНИЛИЩЕ может осуществляться неявно посредством **ВНЕДРЕНИЯ ЗАВИСИМОСТИ** [Fowler, DI] или **ФАБРИКИ СЛУЖБ**. Эта возможность обсуждается в последнем разделе этой главы. Возвращаясь к нашему примеру ПРИКЛАДНОЙ СЛУЖБЫ, мы снова увидим, как ФАБРИКА СЛУЖБ используется для просмотра ХРАНИЛИЩА.

```
package com.saasovation.identityaccess.application;

public class TenantIdentityService {
    ...
    @Override
    @Transactional(readOnly=true)
    public Tenant tenant(TenantId aTenantId) {
        Tenant tenant =
            DomainRegistry
                  .tenantRepository()
                  .tenantOfId(aTenantId);
            return tenant;
    }
    ...
}
```

В качестве альтернативы можно было бы внедрить эту ПРИКЛАДНУЮ СЛУЖБУ в ХРАНИЛИЩЕ или установить входящие зависимости с помощью параметров конструктора.

Реализации ХРАНИЛИЩ хранятся в инфраструктуре, потому что они работают с запоминающими устройствами, которые вообще не должны входить в сферу компетенции модели. Можно было бы также применить инфраструктуру, использующую механизм обмена сообщениями, такую как очередь сообщений или электронная почта. Специальные компоненты пользовательского интерфейса, генерирующие диаграммы, карты и тому подобное, тоже можно реализовать в инфраструктуре.

Контейнеры стандартных компонентов

В настоящее время серверы промышленных приложений стали товаром. На первый взгляд, сами серверы и контейнеры компонентов внутри них не представляют собой практически ничего нового. Для поддержки работы ПРИКЛАДНЫХ СЛУЖБ можно использовать спецификации Enterprise JavaBeans (EJB) как **ФАСАДЫ СЕССИЙ (SESSION FACADE)** [Crupi et al.] или простые спецификации JavaBeans, хранящиеся в контейнерах с инверсией управления. Существуют и более эффективные алгоритмы, но каркасы во многом похожи друг на друга. Например, функции считывания в некоторых серверах JEE напоминают функции, реализованные в системе Spring.

Что это — WebLogic или Spring?

Если бы вам пришлось просматривать трассировку стека на сервере Oracle WebLogic Server, то вы, вероятно, увидели бы ссылки на классы из каркаса Spring Framework. Они не являются частью развертывания вашего приложения. В этом случае вы просто используете стандартную технологию JEE с компонентами EJB Session Beans. Классы Spring, которые вы видите, являются частью реализации контейнера EJB сервера WebLogic. Не является ли это реализацией лозунга "если не можете победить противника, станьте его союзником"?

Я решил реализовать три примера ОГРАНИЧЕННЫХ КОНТЕКСТОВ с помощью каркаса Spring Framework. Эти примеры легко перенести на другие платформы стандартных контейнеров. Таким образом, использование каркаса Spring в наших проектах не ограничивает общность примеров. Разные контейнеры очень мало отличаются друг от друга с точки зрения их логической организации.

Конфигурация системы Spring, обеспечивающей поддержку транзакций для ПРИКЛАДНЫХ СЛУЖБ, использующихся для постоянного хранения объектов предметной области, продемонстрирована в главе, посвященной **ХРАНИЛИЩАМ (12)**. Здесь мы рассмотрим другие части этой конфигурации. Нас интересуют два файла.

```
config/spring/applicationContext-application.xml
config/spring/applicationContext-domain.xml
```

Как следует из их имен, в них осуществляется связывание ПРИКЛАДНЫХ СЛУЖБ и компонентов модели предметной области. Рассмотрим некоторые аспекты этого вопроса.

```
<beans ...>
    <aop:aspectj-autoproxy/>

    <tx:annotation-driven transaction-manager="transactionManager"/>
    ...
    <bean
        id="applicationServiceRegistry"
        class="com.saasovation.identityaccess.application.
                .ApplicationServiceRegistry"
        autowire="byName">
    </bean>
    ...
    <bean
        id="tenantIdentityService"
        class="com.saasovation.identityaccess.application.
                .TenantIdentityService"
        autowire="byName">
    </bean>
    ...
</beans>
```

Компонент tenantIdentityService был рассмотрен ранее. Этот компонент можно связать с другими компонентами каркаса Spring, например с пользовательским интерфейсом. Если вы предпочитаете использовать ФАБРИКУ СЛУЖБ, а не внедрение экземпляров компонентов в другие компоненты, то можете использовать в этой конфигурации другой компонент, applicationServiceRegistry. Этот компонент обеспечивает доступ для просмотра всех ПРИКЛАДНЫХ СЛУЖБ:

```
...
ApplicationServiceRegistry
    .tenantIdentityService()
    .deactivateTenant(tenantId);
```

Это возможно благодаря тому, что в момент создания данный компонент внедряется в компонент ApplicationContext системы Spring.

Для доступа к элементам предметной модели, например к ХРАНИЛИЩАМ или СЛУЖБАМ ПРЕДМЕТНОЙ ОБЛАСТИ, можно использовать аналогичные регистрационные компоненты. Ниже приведена конфигурация компонентов ЖУРНАЛ, ХРАНИЛИЩЕ и СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ для нашей предметной области.

```
<beans ...>
    ...
    <bean
        id="authenticationService"
        class="com.saasovation.identityaccess.infrastructure.⏎
                    .services.DefaultEncryptionAuthenticationService"⏎
        autowire="byName">
    </bean>
    <bean
        id="domainRegistry"
        class="com.saasovation.identityaccess.domain.model.⏎
                    .DomainRegistry"
        autowire="byName">
    </bean>

    <bean
        id="encryptionService"
        class="com.saasovation.identityaccess.infrastructure.⏎
                    .services.MessageDigestEncryptionService"
        autowire="byName">
    </bean>

    <bean
        id="groupRepository"
        class="com.saasovation.identityaccess.infrastructure.⏎
                    .persistence.HibernateGroupRepository"
        autowire="byName">
    </bean>
```

```
    <bean
        id="roleRepository"
        class="com.saasovation.identityaccess.infrastructure.
                .persistence.HibernateRoleRepository"
        autowire="byName">
    </bean>

    <bean
        id="tenantProvisioningService"
        class="com.saasovation.identityaccess.domain.model.
                .identity.TenantProvisioningService"
        autowire="byName">
    </bean>

    <bean
        id="tenantRepository"
        class="com.saasovation.identityaccess.infrastructure.
                .persistence.HibernateTenantRepository"
        autowire="byName">
    </bean>

    <bean
        id="userRepository"
        class="com.saasovation.identityaccess.infrastructure.
                .persistence.HibernateUserRepository"
        autowire="byName">
    </bean>
</beans>
```

Используя компонент `DomainRegistry`, мы можем получить доступ ко всем зарегистрированным компонентам системы Spring. Все эти компоненты могут внедрять зависимость во все другие компоненты системы Spring. Таким образом, ПРИКЛАДНЫЕ СЛУЖБЫ могут использовать либо ФАБРИКУ СЛУЖБ, либо ВНЕДРЕНИЕ ЗАВИСИМОСТИ. Более подробное обсуждение этих двух подходов, а также подхода, основанного на использовании конструкторов, изложено в главе, посвященной **СЛУЖБАМ (7)**.

Резюме

В этой главе мы рассмотрели работу приложения за пределами модели предметной области.

- Мы рассмотрели несколько способов передачи данных о модели предметной области в пользовательский интерфейс.
- Мы рассмотрели способы обработки информации, которая вводится пользователем в модель предметной области.
- Мы рассмотрели способы передачи выходной информации в условиях, когда используются разные виды пользовательского интерфейса.
- Мы изучили реализацию ПРИКЛАДНЫХ СЛУЖБ и их предназначение.
- Мы рассмотрели возможность объединения в пользовательском интерфейсе нескольких моделей.
- Мы научились использовать инфраструктуру для технической реализации приложения.
- Мы увидели, как, используя подход DIP, заставить клиентов каждого аспекта приложения зависеть от абстракций, а не от деталей реализации, что способствует слабой связанности.
- В заключение мы увидели, как можно усилить приложения с помощью стандартных серверов приложений и контейнеров компонентов.

Теперь вы владеете солидными знаниями о предметно-ориентированном проектировании — от тщательно разработанной модели предметной области до компонентов целого приложения.

Приложение

Агрегаты и источники событий

Автор — Ринат Абдуллин

Концепция **ИСТОЧНИКОВ СОБЫТИЙ** используется уже несколько десятилетий, но лишь недавно получила популярность благодаря работам Грэга Янга (Greg Young) по предметно-ориентированному проектированию [Young, ES].

Шаблон ИСТОЧНИК СОБЫТИЙ может использоваться для представления полного состояния **АГРЕГАТА (10)** в виде последовательности **СОБЫТИЙ (8)**, которые произошли после его создания. С помощью СОБЫТИЙ можно восстановить состояние АГРЕГАТА, воспроизводя их в том же порядке, в котором они произошли. Этот подход упрощает постоянное хранение и позволяет выражать концепции со сложными поведенческими свойствами.

Набор СОБЫТИЙ, представляющих состояние каждого АГРЕГАТА, фиксируется в ПОТОКЕ СОБЫТИЙ, допускающем только добавление. Это состояние АГРЕГАТА затем видоизменяется последовательными операциями, которые добавляют новые СОБЫТИЯ в конец ПОТОКА СОБЫТИЙ, как показано на рис. 1. (В этом приложении СОБЫТИЯ представлены как светло-серые прямоугольники, чтобы выделить их среди других концепций.)

Рис. 1. ПОТОК СОБЫТИЙ, **содержащий** СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, **в порядке их появления**

ПОТОК СОБЫТИЙ, связанных с каждым АГРЕГАТОМ, обычно сохраняется в **ХРАНИЛИЩАХ СОБЫТИЙ (8)** в единственном экземпляре, имеющем идентификатор корневой **СУЩНОСТИ (5)**. Создание ХРАНИЛИЩА СОБЫТИЙ, в частности — для использования в шаблоне ИСТОЧНИК СОБЫТИЙ, рассматривается в этом приложении ниже.

Начиная с этого момента мы будем называть подход, основанный на использовании шаблона ИСТОЧНИК СОБЫТИЙ для поддержки и сохранения состояния АГРЕГАТА, как **A+ES**.

Перечислим основные преимущества подхода A+ES.

- Шаблон ИСТОЧНИК СОБЫТИЙ гарантирует, что ни одна причина всех изменений в экземпляре АГРЕГАТА не будет потеряна. При использовании традиционного подхода, основанного на сериализации текущего состояния АГРЕГАТА в базе данных, мы всегда перезаписываем предыдущее сериализованное состояние, которое уже никогда не будет восстановлено. Однако сохранение причины каждого изменения, начиная с момента создания экземпляра АГРЕГАТА на протяжении всего времени его существования, может оказаться неоценимым для бизнеса. Как указывалось в главе, посвященной **АРХИТЕКТУРЕ (4)**, эти преимущества могут быть далеко идущими: надежность, краткосрочный и долгосрочный прогнозы, аналитические открытия, полный журнал аудита, возможность оглянуться для уточнения целей.
- Благодаря тому, что СОБЫТИЯ в ПОТОК СОБЫТИЙ можно только добавлять, существует возможность поддерживать массив данных, предназначенных для репликации. Это позволяет компаниям, таким как LMAX, поддерживать работу систем электронной торговли ценными бумагами с очень небольшим временем задержки.
- Событийно-ориентированный подход к проектированию АГРЕГАТОВ позволяет разработчикам сосредоточиться на выражении поведения с помощью **ЕДИНОГО ЯЗЫКА (1)**, избегая потенциального рассогласования интерфейсов при объектно-реляционном отображении, и создавать более надежные и легко изменяемые системы.

Однако не следует считать, что подход A+ES — это панацея. Рассмотрим его реальные недостатки.

- Определение СОБЫТИЙ в подходе A+ES требует глубоких знаний в предметной области. В любом проекте DDD такой уровень усилий оправдан только для сложных моделей, благодаря которым организация может получить конкурентное преимущество.
- На момент написания этого Приложения, существовал дефицит инструментов и систематизированных знаний в данной области. Это увеличивает стоимость и риски, связанные с применением этого подхода неопытными коллективами разработчиков.
- В этой области мало опытных разработчиков.
- Реализация подхода A+ES практически всегда требует применения принципа **CQRS (4)** (Command-Query Responsibility Segregation — разделение

ответственности на команды и запросы), поскольку запросы к ПОТОКАМ СОБЫТИЙ трудно выполнять. Это повышает нагрузку на разработчика и замедляет его обучение.

Тем, кого не испугали перечисленные проблемы, реализация подхода A+ES может предоставить много преимуществ. Давайте рассмотрим некоторые способы реализации этого мощного подхода в объектно-ориентированном мире.

Внутри прикладной службы

Рассмотрим общую картину применения A+ES внутри **ПРИКЛАДНОЙ СЛУЖБЫ (4, 14)**. АГРЕГАТЫ часто лежат в основе ПРИКЛАДНЫХ СЛУЖБ, которые являются прямыми клиентами модели предметной области.

Когда ПРИКЛАДНАЯ СЛУЖБА получает управление, она загружает АГРЕГАТ и выбирает все **ПРИКЛАДНЫЕ СЛУЖБЫ (7)**, необходимые для работы АГРЕГАТА. Когда ПРИКЛАДНАЯ СЛУЖБА делегирует управление бизнес-операции АГРЕГАТА, метод АГРЕГАТА производит СОБЫТИЕ как результат. Эти СОБЫТИЯ видоизменяют состояние АГРЕГАТА и публикуют уведомления всем подписчикам. Бизнес-метод АГРЕГАТА может получать в качестве параметров несколько СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ. С их помощью можно вычислять значения, вызывающие побочные эффекты, влияющие на состояние АГРЕГАТА. К операциям СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ могут относиться вызов платежной системы, запрос уникальных идентификаторов или запросы данных от удаленной системы (рис. 2).

Следующая ПРИКЛАДНАЯ СЛУЖБА, реализованная на языке C#, демонстрирует этапы, показанные на рис. 2.

```
public class CustomerApplicationService
{
    // Хранилище событий для доступа к потокам событий
    IEventStore _eventStore;

    // Служба предметной области, необходимая для агрегата
    IPricingService _pricingService;

    // Передаем зависимости для прикладной службы с помощью конструктора
    public CustomerApplicationService(
       IEventStore eventStore,
       IPricingService pricing)
    {
       _eventStore = eventStore;
       _pricingService = pricing;
    }
```

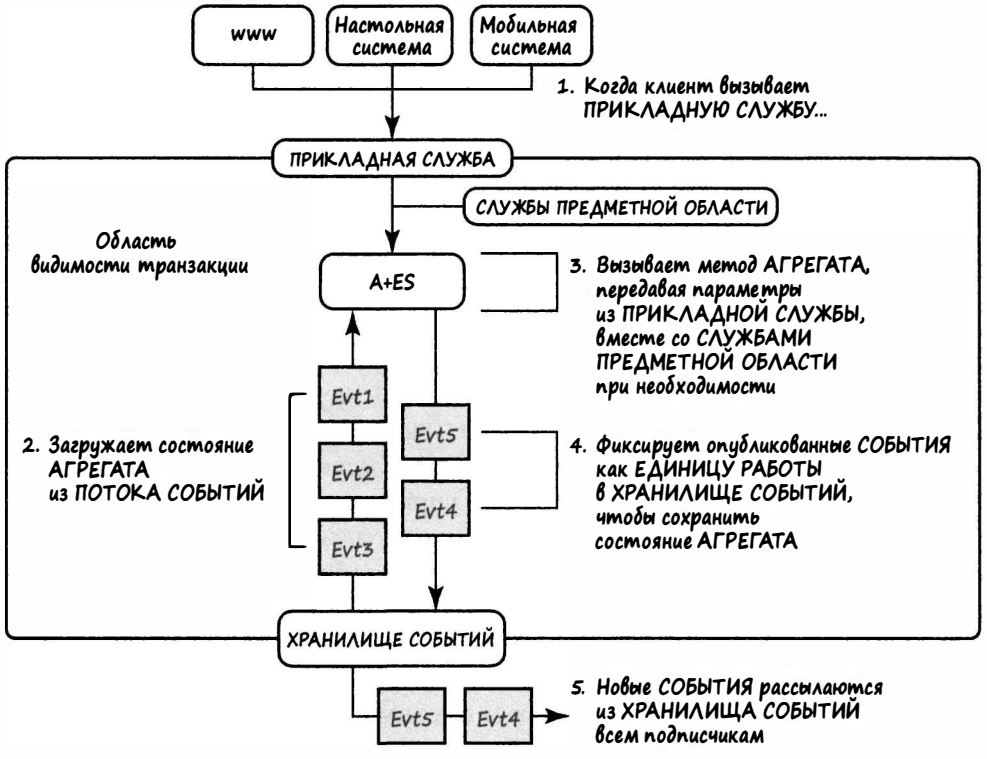


Рис. 2. ПРЕДМЕТНАЯ СЛУЖБА **управляет доступом к** АГРЕГАТУ **и его использованием**

```
// Этап 1: вызываем метод LockForAccountOverdraft
// из класса Customer Application Service
public void LockForAccountOverdraft(
    CustomerId customerId, string comment)
{
   // Этап 2.1: загружаем поток событий для объекта класса Customer,
   // заданного своим идентификатором
   var stream = _eventStore.LoadEventStream(customerId);
   // Этап 2.2: создаем агрегат из потока событий
   var customer = new Customer(stream.Events);
   // Этап 3: вызываем метод агрегата, передавая аргументы и
   // платежную службу предметной области
   customer.LockForAccountOverdraft(comment, _pricingService);
   // Этап 4: фиксируем изменения в потоке событий по идентификатору
   _eventStore.AppendToStream(
      customerId, stream.Version, customer.Changes);
}

 public void LockCustomer(CustomerId customerId, string reason)
 {
    var stream = _eventStore.LoadEventStream(customerId);
```

```
        var customer = new Customer(stream.Events);
        customer.Lock(reason);
        _eventStore.AppendToStream(
            customerId, stream.Version, customer.Changes);
    }

    // Другие методы прикладной службы
}
```

Объект класса `CustomerApplicationService` инициализируется двумя зависимостями, `IEventStore` и `IPricingService`, с помощью конструктора. Инициализация с помощью конструктора удобна для реализации зависимостей, хотя их можно также установить посредством ФАБРИКИ СЛУЖБ или внедрения зависимостей. Выбор за вами.

Где найти образцы кода

Все исходные коды примеров реализации шаблона A+ES можно загрузить с веб-сайта `http://lokad.github.com/lokad-iddd-sample/`.

Интерфейс `IEventStore` и класс `EventStream` могут иметь простые определения.

```
public interface IEventStore
{
    EventStream LoadEventStream(IIdentity id);

    EventStream LoadEventStream(
        IIdentity id, int skipEvents, int maxCount);

    void AppendToStream(
        IIdentity id, int expectedVersion, ICollection<IEvent> events);
}

public class EventStream
{
    // Версия возвращаемого потока событий
    public int Version;

    // Все события в потоке
    public List<IEvent> Events;
}
```

Это ХРАНИЛИЩЕ СОБЫТИЙ довольно просто реализовать с помощью реляционной базы данных (Microsoft SQL, Oracle или MySQL) и хранилища NoSQL, гарантирующих сильную согласованность (файловая система или хранилища MongoDB, RavenDB и Azure Blob).

Мы загружаем СОБЫТИЯ из ХРАНИЛИЩА СОБЫТИЙ, используя уникальный идентификатор экземпляра АГРЕГАТА, который необходимо восстановить. Посмотрим, как это может быть сделано для АГРЕГАТА под названием `Customer`. Несмотря на то что уникальный идентификатор может иметь любой тип, мы будем использовать интерфейс `IIdentity`, реализованный классом `CustomerId`.

Мы должны загрузить СОБЫТИЯ, принадлежащие определенному экземпляру класса `Customer`, и передать их конструктору класса `Customer`, чтобы создать экземпляр АГРЕГАТА.

```
var eventStream = _eventStore.LoadEventStream(customerId);
var customer = new Customer(eventStream.Events);
```

Как показано на рис. 3, АГРЕГАТ применяет СОБЫТИЯ с помощью метода `Mutate()`.

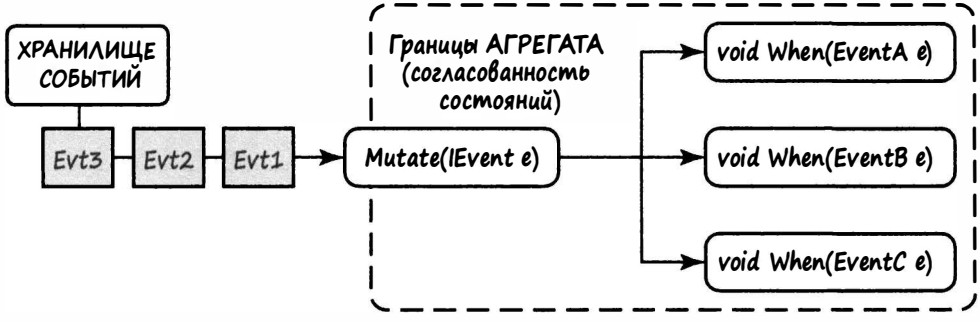


Рис. 3. Состояние АГРЕГАТА восстанавливается путем применения СОБЫТИЙ в порядке их появления

```
public partial class Customer
{
   public Customer(IEnumerable<IEvent> events)
   {
      // Восстанавливаем последнюю версию экземпляра агрегата
      foreach (var @event in events)
      {
          Mutate(@event);
      }
   }

   public bool ConsumptionLocked { get; private set; }

   public void Mutate(IEvent e)
   {
       // Вызов обработчика событий 'When'
       // с соответствующей сигнатурой на платформе .NET
```

```
        ((dynamic) this).When((dynamic)e);
    }

    public void When(CustomerLocked e)
    {
        ConsumptionLocked = true;
    }

    public void When(CustomerUnlocked e)
    {
        ConsumptionLocked = false;
    }

    // и т.д.
```

Метод `Mutate()` находит (с помощью средств платформы .NET) соответствующий перегруженный метод `When()` по конкретному параметру типа СОБЫТИЯ, а затем выполняет этот метод, передавая ему СОБЫТИЕ. После выполнения метода `Mutate()` экземпляр класса `Customer` имеет полностью восстановленное состояние.

Мы можем выполнить операцию запроса для восстановления экземпляра АГРЕГАТА из ХРАНИЛИЩА СОБЫТИЙ.

```
public Customer LoadCustomerById(CustomerId id)
{
    var eventStream = _eventStore.LoadEventStream(id);
    var customer = new Customer(eventStream.Events);
    return customer;
}
```

Рассмотрев пример восстановления экземпляра АГРЕГАТА с помощью ПОТОКА хронологических СОБЫТИЙ, легко представить себе другие его применения. Например, мы можем использовать их, чтобы выяснить, что и когда произошло в прошлом. Эти возможности становятся более действенными при отладке развертывания промышленного приложения.

Как выполняются бизнес-операции? Как только АГРЕГАТ воссоздается с помощью ХРАНИЛИЩА СОБЫТИЙ, ПРИКЛАДНАЯ СЛУЖБА делегирует управление командной операции экземпляра АГРЕГАТА. Для выполнения своей работы он использует свое текущее состояние и ПРИКЛАДНЫЕ СЛУЖБЫ, требуемые по контракту. После выполнения поведенческой функции, изменения состояния выражаются в виде новых СОБЫТИЙ. Новое СОБЫТИЕ передается АГРЕГАТУ и к нему применяется метод `Apply()`, как показано на рис. 4.

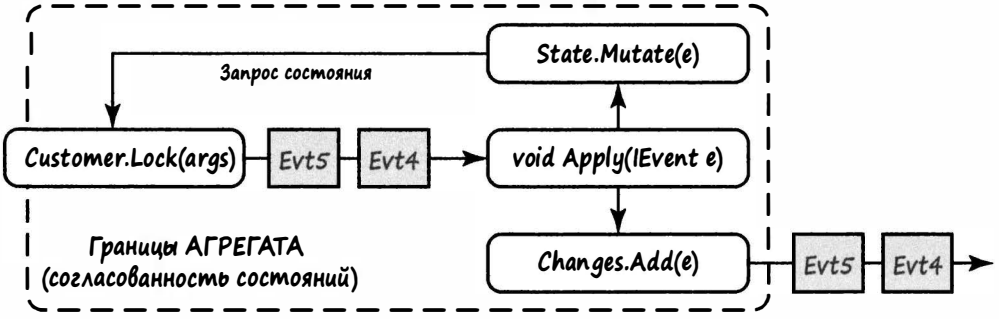


Рис. 4. Состояние АГРЕГАТА основано на прошлых СОБЫТИЯХ и результате работы поведенческой функции, возвращающей новые СОБЫТИЯ

Как показано в следующем коде, новые СОБЫТИЯ собираются в коллекции Changes, а затем используются для изменения текущего состояния АГРЕГАТА.

```
public partial class Customer
{
    ...
    void Apply(IEvent event)
    {
       // Добавляем событие для изменения списка, предназначенного для хранения
       Changes.Add(event);

       // Передаем событие для изменения текущего состояния в памяти
       Mutate(event);
    }
    ...
}
```

Все события, добавляемые в коллекцию Changes, сохраняются как вновь добавленные. Поскольку каждое СОБЫТИЕ используется для немедленного изменения состояния АГРЕГАТА, его поведенческая функция раскладывается на несколько этапов, каждый из которых обновляет состояние АГРЕГАТА.

Рассмотрим поведение агрегата класса Customer.

```
public partial class Customer
{
   // Вторая часть класса агрегата
   public List<IEvent> Changes = new List<IEvent>();

   public void LockForAccountOverdraft(
       string comment, IPricingService pricing)
```

```
    {
        if (!ManualBilling)
        {
            var balance = pricing.GetOverdraftThreshold(Currency);
            if (Balance < balance)
            {
                LockCustomer("Overdraft. " + comment);
            }
    }
}

public void LockCustomer(string reason)
{
    if (!ConsumptionLocked)
    {
        Apply(new CustomerLocked(_state.Id, reason));
    }
}

    // Другие бизнес-методы не показаны...

    void Apply(IEvent e)
    {
        Changes.Add(e);
        Mutate(e);
    }
}
```

Два класса реализации

Для того чтобы сделать код яснее, реализацию шаблона A+ES можно разделить на два разных класса (один — для состояния, а другой — для поведения) так, чтобы объект состояния содержался в поведенческом объекте. Эти два объекта могли бы взаимодействовать исключительно с помощью метода `Apply()`. Это гарантировало бы, что состояние изменяется только посредством СОБЫТИЙ.

После завершения поведенческих функций, изменяющих состояние, объект класса `Changes` необходимо зафиксировать в ХРАНИЛИЩЕ СОБЫТИЙ. Мы добавляем все новые изменения, проверяя отсутствие конфликтов между параллельными потоками записи. Эта проверка возможна, потому что мы передаем параллельную версию переменной от метода `Load()` в метод `Append()`.

В простейшей реализации используется фоновый процессор, перехватывающий все вновь добавленные СОБЫТИЯ и публикующий их в инфраструктуре обмена сообщениями (например, в системах RabbitMQ, JMS, MSMQ или системах облачных запросов), доставляя их всем заинтересованным сторонам (рис. 5).

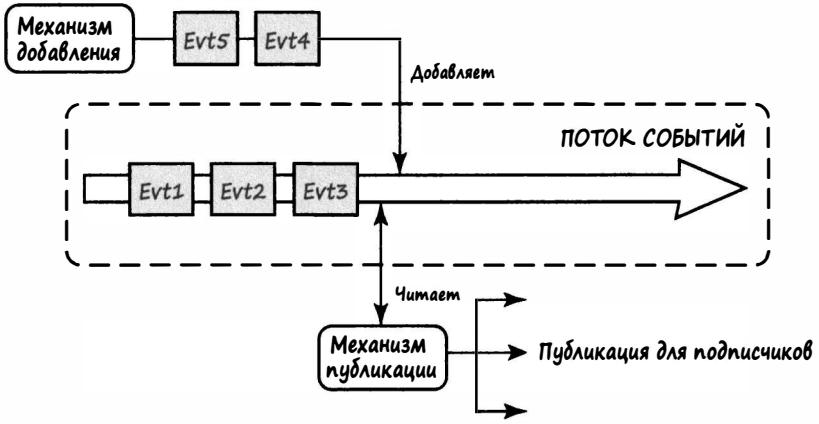


Рис. 5. Вновь добавленные поведенческие события АГРЕГАТА публикуются для подписчиков

Эту простую реализацию можно заменить более сложной, в которой она немедленно или впоследствии создает один или несколько клонов СОБЫТИЯ, повышая отказоустойчивость системы. На рис. 6 показана схема, в которой клон СОБЫТИЯ создается немедленно.

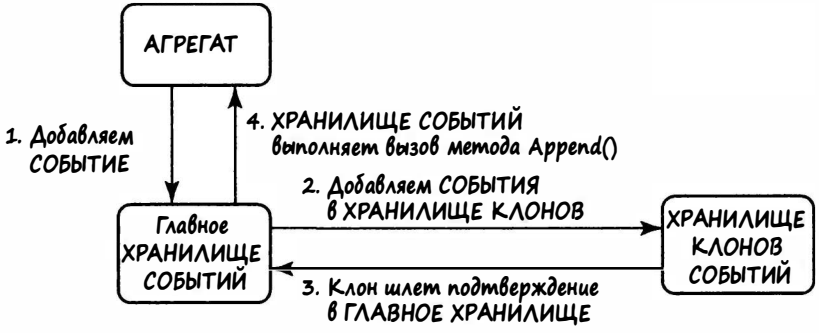


Рис. 6. Немедленная запись: ГЛАВНОЕ ХРАНИЛИЩЕ СОБЫТИЙ немедленно пересылает реплики всех вновь добавленных событий в ХРАНИЛИЩЕ КЛОНОВ СОБЫТИЙ

Стратегия немедленной записи состоит в том, что ГЛАВНОЕ ХРАНИЛИЩЕ СОБЫТИЙ сохраняет свои СОБЫТИЯ только после того, как они будут успешно реплицированы в ХРАНИЛИЩЕ КЛОНОВ СОБЫТИЙ.

Альтернативная стратегия отложенной записи подразумевает репликацию СОБЫТИЙ в виде КЛОНОВ после того, как изменения будут сохранены в главном хранилище с помощью отдельного потока (рис. 7). В этом случае между ХРАНИЛИЩЕМ КЛОНОВ и ГЛАВНЫМ ХРАНИЛИЩЕМ нет согласованности, особенно если сервер дал сбой или возникла сетевая задержка.

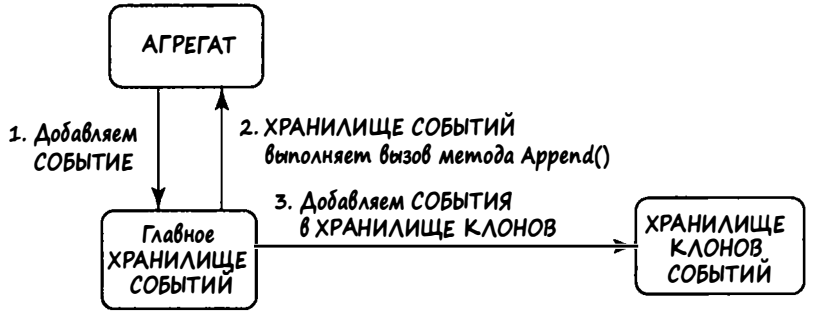


Рис. 7. Отложенная запись: ГЛАВНОЕ ХРАНИЛИЩЕ СОБЫТИЙ с задержкой пересылает реплики всех вновь добавленных событий в ХРАНИЛИЩЕ КЛОНОВ СОБЫТИЙ

Для подведения итогов рассмотрим последовательность действий, которая начинается с вызова операции ПРИКЛАДНОЙ СЛУЖБЫ.

1. Клиент вызывает метод ПРИКЛАДНОЙ СЛУЖБЫ.
2. Клиент получает услуги всех СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ, необходимые для выполнения бизнес-операции.
3. Клиент указывает идентификатор экземпляра АГРЕГАТА и находит этот АГРЕГАТ в ПОТОКЕ СОБЫТИЙ.
4. Клиент восстанавливает экземпляр АГРЕГАТА, применяя его ко всем СОБЫТИЯМ из ПОТОКА.
5. Клиент выполняет бизнес-операцию, предоставленную АГРЕГАТОМ, передавая все параметры, предусмотренные контрактом интерфейса.
6. АГРЕГАТ может выполнить ДВОЙНУЮ ДИСПЕТЧЕРИЗАЦИЮ к любой из предоставленных СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ, экземплярам других агрегатов и так далее и генерирует новые СОБЫТИЯ в качестве результата операции.
7. Если бизнес-операция выполнена успешно, то все вновь сгенерированные СОБЫТИЯ добавляются в ту версию ПОТОКА, которая не создает конфликты в параллельной работе.
8. Система публикует все вновь добавленные СОБЫТИЯ из ХРАНИЛИЩА СОБЫТИЙ для подписчиков с помощью инфраструктуры обмена сообщениями.

Реализацию шаблона A+ES можно улучшить по-разному. Например, можно использовать **ХРАНИЛИЩЕ (12)** для инкапсуляции доступа к ХРАНИЛИЩУ СОБЫТИЙ и деталям процесса восстановления экземпляров АГРЕГАТА. С помощью приведенных ранее фрагментов кода можно легко создать базовый класс

ХРАНИЛИЩА. Остановимся на двух практичных способах усовершенствования подхода A+ES: ОБРАБОТЧИКАХ КОМАНД и ЛЯМБДА-ВЫРАЖЕНИЯХ.

Обработчики команд

Рассмотрим преимущества использования шаблона **КОМАНДА (4, 14)** и ОБРАБОТЧИКОВ КОМАНД для управления задачами в рамках нашего приложения. Для начала еще раз взглянем на ПРИКЛАДНУЮ СЛУЖБУ и ее метод `LockCustomer()`.

```
public class CustomerApplicationService
{
    ...
    public void LockCustomer(CustomerId id, string reason)
    {
       var eventStream = _eventStore.LoadEventStream(id);
       var customer = new Customer(stream.Events);
       customer.LockCustomer(reason);
       _store.AppendToStream(id, eventStream.Version, customer.Changes);
    }
    ...
}
```

Теперь представьте себе процесс создания сериализованного представления имени метода и его параметров. Как он может выглядеть? Мы могли бы создать класс, имя которого совпадает с именем прикладной операции и создать свойства экземпляра, совпадающие с параметрами служебного метода. Этот класс образует КОМАНДУ.

```
public sealed class LockCustomerCommand
{
   public CustomerId { get; set; }
   public string Reason { get; set; }
}
```

Контракты команды следуют той же семантике СОБЫТИЙ и могут использоваться в системе аналогичным способом. Эту КОМАНДУ можно было бы передать методу ПРИКЛАДНОЙ СЛУЖБЫ.

```
public class CustomerApplicationService
{
   ...
   public void When(LockCustomerCommand command)
   {
      var eventStream = _eventStore.LoadEventStream(command.CustomerId);
```

```
        var customer = new Customer(stream.Events);
        customer.LockCustomer(command.Reason);
        _eventStore.AppendToStream(
             command.CustomerId, eventStream.Version, customer.Changes);
    }
    ...
}
```

Этот простой рефакторинг может принести системе долгосрочные преимущества. Посмотрим, как этого достичь.

Поскольку объекты КОМАНДЫ можно сериализовать, мы можем послать их текстовое или бинарное представление в виде сообщения, помещенного в очередь сообщений. Объект, которому передано сообщение, является ОБРАБОТЧИКОМ КОМАНДЫ. По существу, он заменяет метод ПРИЛОЖЕНИЯ, хотя и не совпадает с ним. В любом случае разрыв связи клиента со СЛУЖБОЙ *улучшает баланс нагрузки, допускает конкуренцию пользователей и поддерживает разделение системы.* Для начала рассмотрим баланс нагрузки. Мы можем распределить нагрузку, запустив тот же самый ОБРАБОТЧИК КОМАНД (который с семантической точки зрения является ПРИКЛАДНОЙ СЛУЖБОЙ) на любом количестве серверов. После того как КОМАНДЫ будут помещены в очередь сообщений, командные сообщения могут быть доставлены одному из серверных ОБРАБОТЧИКОВ КОМАНД, которые их ожидают. Эта схема изображена на рис. 8. (На рисунках в этом Приложении КОМАНДЫ изображаются кружками.) Реальное распределение можно осуществить с помощью простого алгоритма циклического обслуживания или более сложного алгоритма доставки, предоставленного инфраструктурой сообщений.

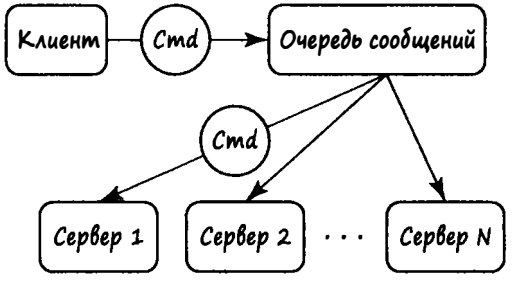


Рис. 8. Распределение ПРИКЛАДНЫХ КОМАНД среди произвольного количества ОБРАБОТЧИКОВ КОМАНД

Этот подход допускает периодический разрыв связи между клиентами и ПРИКЛАДНОЙ СЛУЖБОЙ, повышая надежность системы. Теперь клиент больше не блокируется, если ПРИКЛАДНАЯ СЛУЖБА недоступна в течение короткого периода времени (например, если она остановлена на профилактику или обновляется). Вместо этого КОМАНДА помещается в постоянную очередь, которая будет

обработана ОБРАБОТЧИКАМИ КОМАНД (прикладной службой), когда сервер вернется в строй (рис. 9).

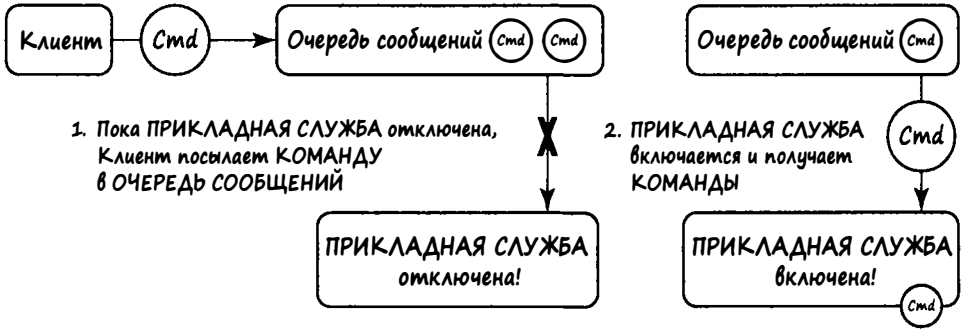


Рис. 9. Использование командных сообщений и их обработчиков, допускающее временное разъединение клиентов и ПРИКЛАДНОЙ СЛУЖБЫ, повышает гибкость систем

Другим преимуществом является возможность создавать цепочки дополнительных действий перед диспетчеризацией КОМАНДЫ. Например, можно легко связать в цепочку аудит, регистрацию, авторизацию и проверку корректности.

Посмотрим, как можно включить в цепочку регистрацию. Сначала определим стандартный интерфейс и реализуем его в классе ПРИКЛАДНОЙ СЛУЖБЫ.

```
public interface IApplicationService
{
    void Execute(ICommand cmd);
}

public partial class CustomerApplicationService : IApplicationService
{
    public void Execute(ICommand command)
    {
       // Передаем команду конкретному методу When(),
       // а затем обрабатываем ее
       ((dynamic)this).When((dynamic)command);
    }
}
```

Методы `Execute()` и `Mutate()` имеют похожие реализации

Обратите внимание на то, что реализация метода `Execute()` похожа на реализацию метода `Mutate()`, описанного ранее как часть шаблона A+ES.

Определив стандартный интерфейс для всех ОБРАБОТЧИКОВ КОМАНД (ПРИКЛАДНЫХ СЛУЖБ), мы можем связывать любые свойства, например универсальную регистрацию.

```
public class LoggingWrapper : IApplicationService
{
    readonly IApplicationService _service;

    public LoggingWrapper(IApplicationService service)
    {
       _service = service;
    }

    public void Execute(ICommand cmd)
    {
       Console.WriteLine("Command: " + cmd);
       try
       {
           var watch = Stopwatch.StartNew();
           _service.Execute(cmd);
           var ms = watch.ElapsedMilliseconds;
           Console.WriteLine(" Completed in {0} ms", ms);
       }
       catch( Exception ex)
       {
           Console.WriteLine("Error: {0}", ex);
       }
    }
}
```

Поскольку все ПРИКЛАДНЫЕ СЛУЖБЫ имеют стандартный интерфейс, мы можем связать в цепочку любое количество универсальных утилит, которые выполняются до и/или после функций ОБРАБОТЧИКА КОМАНДЫ. Посмотрим, как инициализируется объект класса `CustomerApplicationService` при выполнении регистрации до и после выполнения команды.

```
var customerService =
    new CustomerApplicationService(eventStore, pricingService);
var customerServiceWithLogging = new LoggingWrapper(customerService);
```

Конечно, тот факт, что КОМАНДЫ представляют собой сериализованные объекты, позволяет обрабатывать разнообразные ошибки и проверять условия в одном и том же месте. При возникновении ошибки определенного класса, например при выявлении конкуренции за ресурсы при параллельной работе, можно выбрать стандартные средства ее исправления, например выполнить операцию заданное количество раз. При этом можно использовать стратегию ОГРАНИЧЕННОГО СВЕРХУ ЭКСПОНЕНЦИАЛЬНОГО ОТКАТА (CAPPED EXPONENTIAL BACK-OFF). Благодаря этой стратегии все попытки выполняются единообразно, надежно и поддерживаются одним классом.

Синтаксис лямбда-выражений

Если используемый вами язык программирования поддерживает лямбда-выражения, можно сделать часто повторяющийся код более компактным, избежав повторяющихся операций по управлению ПОТОКОМ СООБЩЕНИЙ. Для демонстрации этой возможности включим в нашу ПРИКЛАДНУЮ СЛУЖБУ вспомогательный метод.

```
public class CustomerApplicationService
{
    ...
   public void Update(CustomerId id, Action<Customer> execute)
    {
      EventStream eventStream = _eventStore.LoadEventStream(id);
      Customer customer = new Customer(eventStream.Events);
      execute(customer);
      _eventStore.AppendToStream(
         id, eventStream.Version, customer.Changes);
    }
    ...
}
```

В этом методе параметр `Action<Customer> execute` ссылается на анонимную функцию (делегат C#), которая может оперировать любым экземпляром класса `Customer`. Лаконичность лямбда-выражений можно продемонстрировать с помощью параметра, который передается методу `Update()`.

```
public class CustomerApplicationService
{
    ...
    public void When(LockCustomer c)
    {
       Update(c.Id, customer => customer.LockCustomer(c.Reason));
    }
    ...
}
```

В действительности компилятор языка C# генерирует нечто похожее на следующий код, выполняющий предназначение лямбда-функции.

```
public class AnonymousClass_X
{
    public string Reason;
    public void Execute(Customer customer);
    {
       Customer.LockCustomer(Reason);
    }
}
```

```
public delegate void Action<T>(T argument);

public void When(LockCustomer c)
{
    var x = new AnonymousClass_X();
    x.Reason = c.Reason
    Update(c.Id, new Action<Customer>(customer => x.Execute(customer));
}
```

Поскольку эта сгенерированная функция получает аргумент класса `Customer`, она может использоваться для перехвата поведения в коде и многократного выполнения на разных экземплярах класса `Customer`. Мощь лямбда-выражений демонстрируется в следующем разделе.

Управление параллельной работой

АГРЕГАТНЫЕ ПОТОКИ СОБЫТИЙ разрешают одновременный доступ и чтение нескольким потокам одновременно. Если не предусмотреть проверку, то возникает возможность потенциальных конфликтов, которые могут привести к непредсказуемому количеству некорректных состояний АГРЕГАТА. Рассмотрим сценарий, в котором два потока одновременно пытаются модифицировать ПОТОК СОБЫТИЙ (рис. 10).

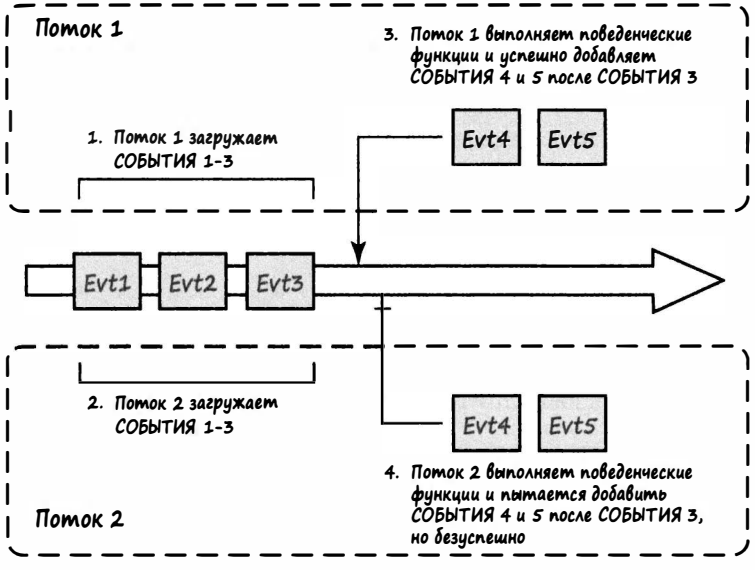


Рис. 10. Два потока конкурируют за один и тот же экземпляр АГРЕГАТА, разработанного по шаблону A+ES

Простейшее решение этой проблемы — использовать на этапе 4 класс исключения EventStoreConcurrencyException, разрешая ему достигать конечного клиента.

```
public class EventStoreConcurrencyException : Exception
{
    public List<IEvent> StoreEvents { get; set; }
    public long StoreVersion { get; set; }
}
```

Перехватывая это исключение в конечном клиенте, проинструктированный пользователь может выполнить данную бизнес-операцию вручную.

Вероятно, вы согласитесь, что стандартный подход, основанный на повторных попытках, лучше, чем описанный выше способ. Следовательно, когда наше ХРАНИЛИЩЕ СОБЫТИЙ генерирует исключение EventStoreConcurrencyException, можно попытаться немедленно выполнить попытку восстановления.

```
void Update(CustomerId id, Action<Customer> execute)
{
   while(true)
   {
       EventStream eventStream = _eventStore.LoadEventStream(c.Id);
       var customer = new Customer(eventStream.Events);
       try
       {
           execute(customer);
           _eventStore.AppendToStream(
              c.Id, eventStream.Version, customer.Changes);
           return;
       }
       catch (EventStoreConcurrencyException)
       {
          // Переходим в начало и выполняем повторную попытку,
          // возможно, с короткой задержкой
       }
   }
}
```

Если возникает конфликт параллельной обработки, для решения проблемы, возможно, потребуется выполнить еще несколько шагов.

1. Поток 2 перехватывает исключение и передает управление в начало цикла while. Теперь СОБЫТИЯ 1–5 загружаются в новый экземпляр класса Customer.
2. Поток 2 повторно выполняет делегирование управления вновь загруженному экземпляру класса Customer, создающему СОБЫТИЯ 6-7, которые будут успешно добавлены после СОБЫТИЯ 5.

Если повторное выполнение операций в АГРЕГАТЕ связано со слишком большими затратами или трудностями (например, требует дорогостоящей интеграции с посторонней системой или дополнительной платы), может понадобиться другая стратегия.

Как показано на рис. 11, одна из таких стратегий — разрешение конфликтов, связанных с СОБЫТИЯМИ. Это позволяет уменьшить количество исключений, порожденных конфликтами параллельной обработки. Рассмотрим простой пример, демонстрирующий разрешение конфликтов.

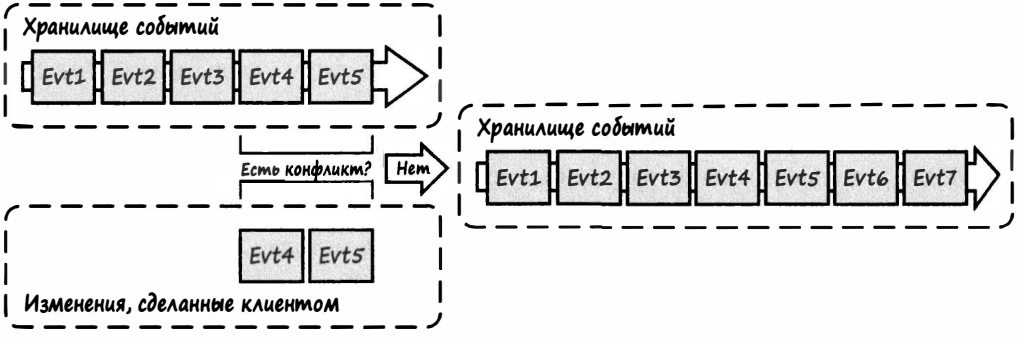


Рис. 11. Разрешение конфликтов между СОБЫТИЯМИ в ПОТОКЕ СОБЫТИЙ, связанных с АГРЕГАТОМ

```
void UpdateWithSimpleConflictResolution(
    CustomerId id, Action<Customer> execute)
    {
       while (true)
       {
          EventStream eventStream = _eventStore.LoadEventStream(id);
          Customer customer = new Customer(eventStream.Events);
          execute(customer);

          try
          {
             _eventStore.AppendToStream(
                id, eventStream.Version, customer.Changes);
             return;
          }
          catch (EventStoreConcurrencyException ex)
          {
             foreach (var failedEvent in customer.Changes)
             {
                foreach (var succeededEvent in ex.ActualEvents)
                {
                   if (ConflictsWith(failedEvent, succeededEvent))
                   {
                       var msg = string.Format("Conflict between {0} and {1}",
```

```
                        failedEvent, succeededEvent);
                    throw new RealConcurrencyException(msg, ex);
                }
            }
            // Конфликтов нет, и можно добавлять событие
            _eventStore.AppendToStream(
                id, ex.ActualVersion, customer.Changes);
        }
    }
}
```

В данном случае используется метод распознавания конфликтов `ConflictsWith()`, проверяющий, нет ли конфликта между каждым из СОБЫТИЙ, связанных с АГРЕГАТОМ, и событиями, которые параллельно добавлялись в ХРАНИЛИЩЕ СОБЫТИЙ (как указано в исключении).

Метод разрешения конфликтов обычно определяется в КОРНЕ АГРЕГАТА в зависимости от конкретного вида поведенческих функций, которые он поддерживает. Впрочем, существует универсальная реализация метода `ConflictsWith()`, которая может работать с большинством АГРЕГАТОВ.

```
bool ConflictsWith(IEvent event1, IEvent event2)
{
    return event1.GetType() == event2.GetType();
}
```

Это разрешение конфликтов основано на простом правиле: СОБЫТИЯ одного типа всегда конфликтуют друг с другом, а СОБЫТИЯ разных типов — нет.

Структурная свобода шаблона A+ES

Одно из самых больших практических преимуществ шаблона A+ES — простота постоянного хранения и универсальность. Независимо от того, насколько сложной является структура АГРЕГАТА, она может всегда представляться в виде последовательности сериализованных СОБЫТИЙ, которые могут использоваться для ее воссоздания. Под влиянием предметной области и изменяющихся требований развивающейся системы в модель со временем включаются новые функции и структурные элементы. Даже если необходимо изменить структуру внутренней реализации данного АГРЕГАТА, чтобы учесть существенные изменения, шаблон A+ES в большинстве случаев упрощает такие изменения, уменьшая риск ошибок и мучения разработчиков.

Последовательность СОБЫТИЙ, связанных с определенными идентификаторами, обычно называется ПОТОКОМ СОБЫТИЙ. В сущности, это просто список,

допускающий только добавление сообщений, сериализованных в блоки байтов с помощью выбранного механизма сериализации. ПОТОК СОБЫТИЙ может храниться как в реляционных базах данных, так и в механизмах хранения NoSQL, простых файловых системах или "облачном" хранилище, поскольку у любого выбранного хранилища есть гарантии строгой согласованности.

Механизм хранения на основе шаблона A+ES имеет три основных преимущества, особенно важных в долговременных **ОГРАНИЧЕННЫХ КОНТЕКСТАХ (2)**.

- Возможность адаптировать внутреннюю реализацию АГРЕГАТА к любому практическому структурному представлению, необходимому для выражения новых поведенческих функций, с которыми встречаются специалисты по проблемной области.
- Возможность перемещения всей инфраструктуры между различными решениями для хостинга, позволяющая реагировать на отключение "облака" и повышающая отказоустойчивость.
- Возможность загружать ПОТОК СОБЫТИЙ для любого экземпляра АГРЕГАТА, предназначенного для разработки и отладки ошибочных состояний СОБЫТИЯ.

Производительность

Иногда загрузка АГРЕГАТОВ из больших ПОТОКОВ может снижать производительность, особенно если отдельные ПОТОКИ содержат больше сотен тысяч СОБЫТИЙ. Есть несколько простых шаблонов, которые в отдельных случаях помогают решить эту проблему.

- Кеш ПОТОКА СОБЫТИЙ в памяти сервера, использующий тот факт, что СОБЫТИЯ после записи в ХРАНИЛИЩЕ СОБЫТИЙ остаются неизменными. При обращении к ХРАНИЛИЩУ СОБЫТИЯ для внесения любых изменений мы могли бы предоставить версию последнего известного СОБЫТИЯ и запросить только те из них, которые произошли с момента наступления этого СОБЫТИЯ, если такие существуют. Это может улучшить производительность за счет затрат памяти.
- *Снимки* экземпляров АГРЕГАТА позволяют избегать загрузки и воспроизведения значительной части ПОТОКА СОБЫТИЙ. При загрузке любого экземпляра АГРЕГАТА достаточно просто найти его последний снимок, а затем воспроизвести любые СОБЫТИЯ, которые были добавлены в ПОТОК СОБЫТИЙ после того, как был сделан это снимок.

Как показано на рис. 12, снимки представляют собой сериализованные копии полного состояния АГРЕГАТА, сделанные в определенные моменты времени и хранящиеся в ПОТОКЕ СОБЫТИЙ в виде отдельных версий. Они могут храниться в ХРАНИЛИЩЕ, имеющем простой интерфейс.

```
public interface ISnapshotRepository
{
  bool TryGetSnapshotById<TAggregate>(
     IIdentity id, out TAggregate snapshot, out int version);
 void SaveSnapshot(IIdentity id, TAggregate snapshot, int version);
}
```

Рис. 12. ПОТОК СОБЫТИЙ, связанных с АГРЕГАТОМ, со снимком его состояния, за которыми следуют два СОСТОЯНИЯ, возникших после снимка

Мы должны записать версию ПОТОКА вместе со всеми снимками. В этой версии мы можем загрузить снимок только вместе с СОБЫТИЯМИ, которые произошли с момента регистрации снимка. Сначала мы извлекаем снимок как основное состояние экземпляра АГРЕГАТА, а затем загружаем и воспроизводим все СОБЫТИЯ, которые произошли после того, как был сделан снимок.

```
// Простой интерфейс хранилища документов
ISnapshotRepository _snapshots;

// Хранилище событий
IEventStore _store;

public Customer LoadCustomerAggregateById(CustomerId id)
{
    Customer customer;
    long snapshotVersion = 0;
    if (_snapshots.TryGetSnapshotById(
        id, out customer, out snapshotVersion))
    {
       // Загружаем все события, возникшие после того, как был сделан снимок
       EventStream stream = _store.LoadEventStreamAfterVersion(
          id, snapshotVersion);
       // Воспроизводим эти события для обновления снимка
       customer.ReplayEvents(stream.Events);
       return customer;
    }
    else // В хранилище нет снимков
```

```
    {
        EventStream stream = _store.LoadEventStream(id);
        return new Customer(stream.Events);
    }
}
```

Метод `ReplayEvents()` обновляет состояние экземпляра АГРЕГАТА с помощью СОСТОЯНИЙ, которые произошли после того, как был сделан последний снимок. Напоминаем, что состояние экземпляра АГРЕГАТА изменяется начиная с момента последнего снимка. Следовательно, невозможно создать экземпляр класса `Customer` (в данном примере), используя только ПОТОК СОСТОЯНИЙ. Просто применить метод `Apply()` тоже нельзя, потому что он не только изменяет текущее состояние конкретного СОБЫТИЯ, но и сохраняет каждое полученное СОБЫТИЕ в коллекции `Changes`. Запись в коллекцию `Changes` СОБЫТИЯ, которое уже находится в хранилище событий, может иметь серьезные последствия. Таким образом, нам просто нужно реализовать новый метод `ReplayEvents()`.

```
public partial class Customer
{
    ...
    public void ReplayEvents(IEnumerable<IEvent> events)
    {
        foreach (var event in events)
        {
            Mutate(event);
        }
    }
    ...
}
```

Приведем простой код, генерирующий снимки объекта класса `Customer`.

```
public void GenerateSnapshotForCustomer(IIdentity id)
{
    // Сначала загружаем все события
    EventStream stream = _store.LoadEventStream(id);
    Customer customer = new Customer(stream.Events);
    _snapshots.SaveSnapshot(id, customer, stream.Version);
}
```

Генерацию снимков и их постоянное хранение можно делегировать фоновому потоку. Можно сделать так, чтобы новые снимки создавались только после того, как с момента создания последнего снимка произойдет определенное количество СОБЫТИЙ. Эти этапы представлены на рис. 13. Поскольку характеристики типов АГРЕГАТОВ могут быть разными, этот параметр, регулирующий генерацию снимков, можно настроить с учетом конкретных требований к производительности системы.

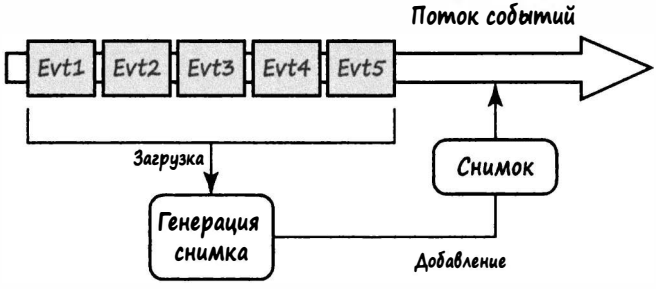


Рис. 13. Снимок АГРЕГАТА, сгенерированный после того, как произошло определенное количество новых СОБЫТИЙ

Существует еще один способ повысить производительность систем с помощью АГРЕГАТОВ, спроектированных по шаблону A+ES, — разделить АГРЕГАТЫ между несколькими процессами или машинами по их идентификаторам. Такое разделение можно осуществить с помощью хеширования идентификаторов или других алгоритмов, сочетая кеширование экземпляров АГРЕГАТОВ в памяти и генерацию их снимков.

Реализация хранилища событий

Перейдем к реализации других ХРАНИЛИЩ СОБЫТИЙ, которые удобно использовать в шаблоне A+ES. Это простые и удобные ХРАНИЛИЩА, подходящие для большинства предметных областей, хотя они и не обеспечивают чрезвычайно высокую производительность.

Несмотря на то что разные ХРАНИЛИЩА СОБЫТИЙ имеют разную реализацию, их контракты совпадают.

```
public interface IEventStore
{
    // Загружаем все события в поток
    EventStream LoadEventStream(IIdentity id);
    // Загружаем в поток подмножество событий
    EventStream LoadEventStream(
       IIdentity id, int skipEvents, int maxCount);
    // Добавляем события в поток, генерируя
    // исключение OptimisticConcurrencyException,
    // если версии событий не совпадают с ожидаемой
    void AppendToStream(
       IIdentity id, int expectedVersion, ICollection<IEvent> events);
}

public class EventStream
{
```

```
    // Версия возвращаемого потока событий
    public int Version;
    // Все события в потоке
    public IList<IEvent> Events = new List<IEvent>();
}
```

Как показано на рис. 14, класс, реализующий интерфейс `IEventStore`, представляет собой специализированную оболочку, содержащую более общий повторно используемый интерфейс `IAppendOnlyStore`. В то время как интерфейс `IEventStore` реализует сериализацию и строгую типизацию, реализации интерфейса `IAppendOnlyStore` обеспечивают низкоуровневый доступ к разным механизмам хранения.

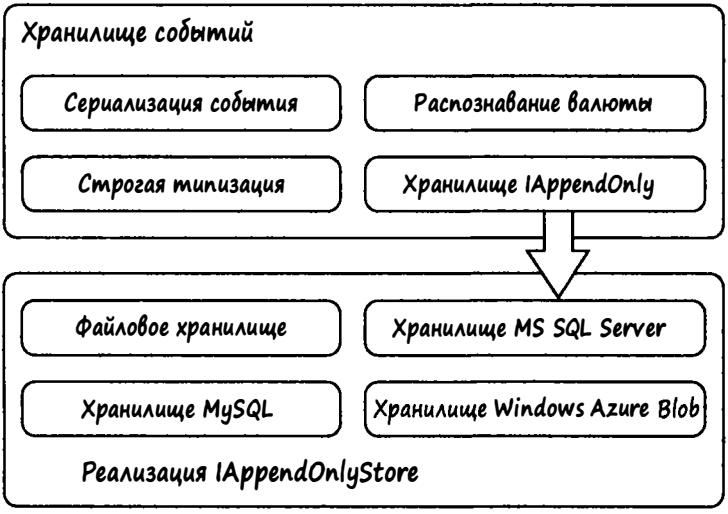


Рис. 14. Характеристики высокоуровневого интерфейса `IEventStore` и низкоуровневого интерфейса `IAppendOnlyStore`

Исходный код ХРАНИЛИЩА СОБЫТИЙ

Полный исходный код разнообразных ХРАНИЛИЩ СОБЫТИЙ с разными реализациями механизма хранения доступен как часть проекта A+ES по адресу `http://lokad.github.com/lokad-iddd-sample/`.

Приведем низкоуровневый интерфейс `IAppendOnlyStore`.

```
public interface IAppendOnlyStore : IDisposable
{
    void Append(string name, byte[] data, int expectedVersion = -1);
    IEnumerable<DataWithVersion> ReadRecords(
       string name, int afterVersion, int maxCount);
```

```
    IEnumerable<DataWithName> ReadRecords(
       int afterVersion, int maxCount);

    void Close();
}

public class DataWithVersion
{
    public int Version;
    public byte[] Data;

}

public sealed class DataWithName
{
    public string Name;
    public byte[] Data;
}
```

Легко видеть, что интерфейс `IAppendOnlyStore` работает с массивами байтов, а не с коллекциями СОБЫТИЙ, и с именами, заданными в виде строк, а не со строго типизированными идентификаторами. Преобразования между этими типами данных осуществляет класс `EventStore`.

В интерфейсе `IAppendOnlyStore` объявлены два разных метода `ReadRecords()`. Первый из них используется для считывания СОБЫТИЙ в отдельные ПОТОКИ по их именам, а второй — для считывания всех СОБЫТИЙ из ХРАНИЛИЩА. Реализации обоих методов всегда считывают СОБЫТИЯ в том порядке, в котором они были сохранены. Как вы, возможно, уже догадались, первый из перегруженных методов необходим для восстановления состояния отдельного АГРЕГАТА. Второй метод `ReadRecords()` используется инфраструктурой для репликации СОБЫТИЙ, чтобы публиковать их, не прибегая к двухфазной фиксации, а также для восстановления сохраненных моделей, например моделей, которые нужны для интерфейсов, основанных на принципе CQRS.

Простой подход к сериализации и десериализации — преобразование между байтами и строго типизированными объектами СОБЫТИЙ — может использовать класс `BinaryFormatter` платформы .NET.

```
public class EventStore : IEventStore
{
    readonly BinaryFormatter _formatter = new BinaryFormatter();
    byte[] SerializeEvent(IEvent[] e)
    {
       using (var mem = new MemoryStream())
       {
          _formatter.Serialize(mem, e);
          return mem.ToArray();
       }
    }
```

```
    IEvent[] DeserializeEvent(byte[] data)
    {
       using (var mem = new MemoryStream(data))
       {
          return (IEvent[])_formatter.Deserialize(mem);
       }
    }
}
```

Вот как можно выполнить сериализацию и десериализацию для загрузки ПОТОКА СООБЩЕНИЙ.

```
readonly IAppendOnlyStore _appendOnlyStore;
...
public EventStream LoadEventStream(IIdentity id, int skip, int take)
{
    var name = IdentityToString(id);
    var records = _appendOnlyStore.ReadRecords(name, skip, take).ToList();
    var stream = new EventStream();

    foreach (var tapeRecord in records)
    {
       stream.Events.AddRange(DeserializeEvent(tapeRecord.Data));
       stream.Version = tapeRecord.Version;
    }
    return stream;
}

string IdentityToString(IIdentity id)
{
    // В этом проекте все идентификаторы образуют соответствующее имя
    return id.ToString();
}
```

Здесь продемонстрировано добавление новых СОБЫТИЙ в ХРАНИЛИЩЕ СОБЫТИЙ с помощью интерфейса `IAppendOnlyStore`.

```
public void AppendToStream(
   IIdentity id, int originalVersion, ICollection<IEvent> events)
{
   if (events.Count == 0)
       return;
   var name = IdentityToString(id);
   var data = SerializeEvent(events.ToArray());
   try
   {
      _appendOnlyStore.Append(name, data, originalVersion);
   }
   catch(AppendOnlyStoreConcurrencyException e)
   {
       // Загружаем события сервера
```

```
        var server = LoadEventStream(id, 0, int.MaxValue);
        // Генерируем реальную проблему
        throw OptimisticConcurrencyException.Create(
           server.Version, e.ExpectedVersion, id, server.Events);
    }
}
```

Реляционный механизм постоянного хранения

Благодаря возможностям и гарантиям строгой согласованности, обеспечиваемым реляционными базами данных, они являются самыми простыми механизмами реализации постоянного хранения, допускающими только добавление. Тот факт, что много предприятий уже стандартизировали свои продукты на основе реляционных баз данных, означает, что никаких затрат и дополнительного обучения ХРАНИЛИЩА СОБЫТИЙ не требуют.

Поскольку СУБД MySQL является популярным сервером реляционных баз данных с открытым исходным кодом, который доступен на нескольких платформах, мы будем использовать ее для реализации ХРАНИЛИЩА СОБЫТИЙ. Класс `MySQLAppendOnlyStore` реализует интерфейс `IAppendOnlyStore`, действуя как уровень доступа. Он будет использоваться для сохранения СОБЫТИЙ в двоичном виде в таблице `ES_EVENTS` и последующей загрузки этих СОБЫТИЙ.

Приведем определение таблицы, управляющей ПОТОКОМ СОБЫТИЙ для каждого типа АГРЕГАТА в ОГРАНИЧЕННОМ КОНТЕКСТЕ.

```
CREATE TABLE IF NOT EXISTS 'ES_Events' (
   'Id' int NOT NULL AUTO_INCREMENT, -- уникальный идентификатор
   'Name' nvarchar(50) NOT NULL,     -- имя потока
   'Version' int NOT NULL,           -- увеличение номера версии потока
   'Data' LONGBLOB NOT NULL          -- полезные данные
)
```

Для того чтобы добавить СОБЫТИЕ в конкретный ПОТОК с помощью транзакции, выполняются следующие шаги.

1. Начинается транзакция.
2. Проверяем, отличается ли номер версии хранилища событий от ожидаемого; если нет, генерируем исключение.
3. Если не возникло никаких конфликтов параллельной обработки, то добавляем СОБЫТИЯ.
4. Фиксируем транзакцию.

Приведем исходный код метода `Append()`.

```
public void Append(string name, byte[] data, int expectedVersion)
{
    using (var conn = new MySqlConnection(_connectionString))
    {
       conn.Open();
       using (var tx = conn.BeginTransaction())
       {
           const string sql =
              @"SELECT COALESCE(MAX(Version),0)
                FROM 'ES_Events'
                WHERE Name=?name";
           int version;
           using (var cmd = new MySqlCommand(sql, conn, tx))
           {
                cmd.Parameters.AddWithValue("?name", name);
                version = (int)cmd.ExecuteScalar();
                if (expectedVersion != -1)
                {
                    if (version != expectedVersion)
                    {
                        throw new AppendOnlyStoreConcurrencyException(
                           version, expectedVersion, name);
                    }
                }
           }

           const string txt =
               @"INSERT INTO 'ES_Events' ('Name', 'Version', 'Data')
                 VALUES(?name, ?version, ?data)";

           using (var cmd = new MySqlCommand(txt, conn, tx))
           {
               cmd.Parameters.AddWithValue("?name", name);
               cmd.Parameters.AddWithValue("?version", version+1);
               cmd.Parameters.AddWithValue("?data", data);
               cmd.ExecuteNonQuery();
           }
           tx.Commit();
       }
    }
}
```

Считывание из объекта класса `IAppendOnlyStore` выполняется довольно просто, поскольку для этого требуется только один запрос. Например, приведенный ниже код демонстрирует, как получить список записей из ПОТОКА СОБЫТИЙ АГРЕГАТА.

```
public IEnumerable<DataWithVersion> ReadRecords(
   string name, int afterVersion, int maxCount)
   {
```

```
        using (var conn = new MySqlConnection(_connectionString))
        {
            conn.Open();
            const string sql =
                @"SELECT 'Data', 'Version' FROM 'ES_Events'
                  WHERE 'Name' = ?name AND 'Version'>?version
                  ORDER BY 'Version'
                  LIMIT 0,?take";
            using (var cmd = new MySqlCommand(sql, conn))
            {
               cmd.Parameters.AddWithValue("?name", name);
               cmd.Parameters.AddWithValue("?version", afterVersion);
               cmd.Parameters.AddWithValue("?take", maxCount);
               using (var reader = cmd.ExecuteReader())
               {
                   while (reader.Read())
                   {
                      var data = (byte[])reader["Data"];
                      var version = (int)reader["Version"];
                      yield return new DataWithVersion(version, data);
                   }
               }
            }
    }
}
```

Полный исходный код этого ХРАНИЛИЩА СОБЫТИЙ, основанного на технологии MySQL, хранится на веб-сайте вместе с остальными примерами. Там же приводится аналогичная реализация для Microsoft SQL Server.

Хранение объектов BLOB

Применение сервера базы данных (такого, как MySQL или SQL Server MS) существенно облегчает работу. Сервер позволяет значительно упростить параллельную работу, фрагментацию файла, кеширование и обеспечение согласованности данных. Очевидно, что отказ от использования готовой базы данных потребовал бы, чтобы мы самостоятельно решали многие из перечисленных проблем.

Однако, если мы примем смелое решение встать на тернистый путь использования ХРАНИЛИЩ СОБЫТИЙ, нам потребуется определенная помощь. Допустим, что в нашем распоряжении есть хранилище Windows Axure Blob и простое файловое хранилище, а также демонстрационный проект, включающий реализации обоих хранилищ.

Рассмотрим несколько принципов проектирования ХРАНИЛИЩА СОБЫТИЙ без базы данных. Некоторые из них проиллюстрированы на рис. 15.

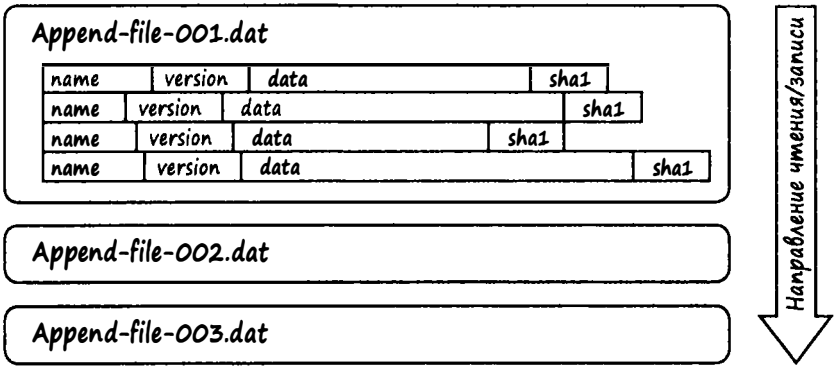


Рис. 15. Файловое BLOB-хранилище, в котором каждому экземпляру АГРЕГАТА соответствует отдельный файл, содержащий одну запись для каждого СОБЫТИЯ

1. Пользовательское хранилище должно состоять из одного или нескольких больших двоичных файловых объектов (binary large object — BLOB) или их эквивалентов. Компонент, выполняющий запись в хранилище, осуществляет его эксклюзивную блокировку во время записи, но допускает параллельное чтение.
2. В зависимости от выбранной стратегии можно было бы использовать всего одно хранилище BLOB для всех типов и экземпляров АГРЕГАТОВ в ОГРАНИЧЕННОМ КОНТЕКСТЕ. В качестве альтернативы можно было бы создать одно хранилище BLOB для каждого типа АГРЕГАТА, где сохранялись бы все экземпляры данного типа. Кроме того, можно было бы разделить хранилища BLOB для каждого типа АГРЕГАТА по экземплярам, где сохранялся бы ПОТОК СОБЫТИЙ для единственного экземпляра.
3. Когда компонент записи добавляет объект в хранилище, он открывает соответствующее хранилище BLOB, записывает в него объект и ведет индекс хранилища.
4. Независимо от выбранной стратегии хранения BLOB, все новые СОБЫТИЯ добавляются в конец. Каждая запись состоит из имени, версии и полей двоичных данных. Это напоминает способ, с помощью которого мы записываем СОБЫТИЯ в реляционную базу данных. Однако в хранилище BLOB должно существовать префиксное поле для указания переменной длины, которая равна подсчитанному количеству байтов, и хеш-код или контрольный циклический избыточный код (cyclic redundancy check — CRC) для проверки целостности данных при считывании записей.
5. Хранилище BLOB, допускающее только добавление, позволяет выполнить перебор всех СОБЫТИЙ во всех ПОТОКАХ СОБЫТИЙ с помощью простого

чтения всех файлов и их содержания. Для того чтобы ускорить поиск на диске и считывание СОБЫТИЙ из определенного ПОТОКА, мы должны были бы поддерживать специальный индекс в памяти и/или кешировать ПОТОКИ СОБЫТИЙ в памяти. Если используется кеширование памяти, то каждое добавление потребовало бы обновления кеша. Кроме того, повысить производительность позволят снимки состояния АГРЕГАТА и дефрагментация файла.

6. Конечно, можно избежать многих дисковых проблем, связанных с фрагментацией файловой системы, предварительно выделяя большие области файлового пространства для BLOB при создании каждого файлового ПОТОКА СОБЫТИЙ.

Образцом для этого проекта была модель Riak Bitcask. Более подробную информацию об архитектуре Riak Bitcask можно найти по адресу `http://docs.basho.com/riak/latest/dev/using/application-guide/`.

Специализированные агрегаты

При разработке АГРЕГАТОВ с традиционными механизмами постоянного хранения (например, реляционной базы данных без использования ИСТОЧНИКА СОБЫТИЙ) сложность разработки вследствие введения в систему нового ОБЪЕКТА или усовершенствования существующего может оказаться значительной. Мы должны создать новые таблицы и определить новые схемы отображения и методы ХРАНИЛИЩА. Если мы стремимся не допустить возрастания сложности проектирования, то должны сосредоточиться на совершенствовании АГРЕГАТОВ, концентрируя в них более сложные структуры и поведенческие функции. Иногда намного проще улучшить существующий АГРЕГАТ, чем создать новый.

Однако, если АГРЕГАТЫ проще разработать заново, наше решение может измениться. При использовании ИСТОЧНИКА СОБЫТИЙ именно так и происходит. Судя по моему опыту, АГРЕГАТЫ, разработанные по шаблону A+ES, часто оказываются меньше, что является одним из основных эмпирических правил работы с агрегатами.

Например, если услугой компании является предоставление программного обеспечения, реальный потребитель может быть представлен с помощью разных АГРЕГАТОВ, сосредоточенных на разных поведенческих аспектах.

- Класс `Customer:505` содержит поведенческие функции для выставления счетов, выписывания счетов-фактур и ведения бухгалтерии.

- Класс `Security-Account:505` поддерживает работу нескольких пользователей, предоставляя доступ каждому из них.
- Класс `Consumer:505` отслеживает потребность в реальных услугах.

Каждый из этих типов АГРЕГАТОВ может быть реализован в отдельном ОГРАНИЧЕННОМ КОНТЕКСТЕ, причем в каждом ОГРАНИЧЕННОМ КОНТЕКСТЕ могут использоваться разные технологии и архитектурные подходы. Например, с помощью аспекта `Consumer` можно обеспечить высокую масштабируемость и каждую секунду обрабатывать тысячи сообщений для клиентов. В этом случае следует разместить такой ПОТОК СОБЫТИЙ в облачной фабрике с автоматическим масштабированием. Другие аспекты могут быть менее требовательными и размещаться в менее требовательной среде.

Конечно, АГРЕГАТЫ никогда не должны разрабатываться произвольно малыми. Мы всегда стремимся разработать АГРЕГАТЫ, защищающие истинные инварианты бизнеса, а в этом случае АГРЕГАТЫ состоят из многочисленных ОБЪЕКТОВ и ОБЪЕКТОВ-ЗНАЧЕНИЙ. И все же простота использования шаблона A+ES дает возможность успешно бороться за простые и эффективные проекты. Это преимущество, которым необходимо пользоваться, когда это возможно.

На практике иногда полезно начать моделирование предметной области с определения ядра ЕДИНОГО ЯЗЫКА, основных входящих КОМАНД и исходящих СОБЫТИЙ, а также с выполняемых поведенческих функций. Только на более позднем этапе мы могли бы сгруппировать некоторые концепции, такие как АГРЕГАТЫ, на основе сходства, релевантности и бизнес-правил. Такой подход, даже если он лишь временно используется в рамках моделирования предметной области, может привести к более глубокому пониманию основных концепций бизнеса.

Проекции модели чтения

Одна из общих проблем подхода к проектированию A+ES состоит в том, как запросить АГРЕГАТЫ по их свойствам. Шаблон ИСТОЧНИК СОБЫТИЙ не обеспечивает простого способа для ответа, например, на вопрос "Каков общий объем всех потребительских заказов в течение прошлого месяца?" Мы должны были бы фактически загрузить каждый экземпляр класса `Customer`, перебрать все экземпляры класса `Order` за прошлый месяц и вычислить их общее количество, что чрезвычайно неэффективно.

Решить эту проблему поможет шаблон **ПРОЕКЦИИ МОДЕЛИ ЧТЕНИЯ (READ MODEL PROJECTIONS)**. Его можно реализовать с помощью подписчиков на СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, которые используются для генерации и обновления постоянной МОДЕЛИ ЧТЕНИЯ. Другими словами, *они проектируют СОБЫТИЯ*

на постоянную МОДЕЛЬ ЧТЕНИЯ. Когда подписчики получают новые СОБЫТИЯ, они вычисляют некоторые результаты запроса и сохраняют их в МОДЕЛИ ЧТЕНИЯ для последующего потребления.

Короче говоря, ПРОЕКЦИЯ очень похожа на экземпляр АГРЕГАТА. После того как СОБЫТИЯ получены и обработаны, мы извлекаем из них данные, чтобы создать состояние ПРОЕКЦИИ. ПРОЕКЦИИ МОДЕЛИ ЧТЕНИЯ после каждого обновления сохраняются и могут быть предоставлены многим читателям как внутри, так и снаружи ОГРАНИЧЕННОГО КОНТЕКСТА.

Примеры проекций

Более подробная информация о ПРОЕКЦИЯХ, включая исходный код для разных сценариев постоянного хранения и автоматической перестройки МОДЕЛЕЙ ЧТЕНИЯ, доступна в учебном проекте, расположенном по адресу `http://lokad.github.com/lokad-cqrs/`.

Покажем, как можно определить ПРОЕКЦИЮ для перехвата всех транзакций для каждого экземпляра класса `Customer`.

```
public class CustomerTransactionsProjection
{
    IDocumentWriter<CustomerId, CustomerTransactions> _store;

    public CustomerTransactionsProjection(
       IDocumentWriter<CustomerId, CustomerTransactions> store)
    {
       _store = store;
    }

    public void When(CustomerCreated e)
    {
       _store.Add(e.Id, new CustomerTransactions());
    }

    public void When(CustomerChargeAdded e)
    {
       _store.UpdateOrThrow(e.Id,
       v => v.AddTx(e.ChargeName, -e.Charge, e.NewBalance, e.TimeUtc));
    }

    public void When(CustomerPaymentAdded e)
    {
       _store.UpdateOrThrow(e.Id,
       v => v.AddTx(e.PaymentName, e.Payment, e.NewBalance, e.TimeUtc));
    }
}
```

Этот класс ПРОЕКЦИИ похож на ПРИКЛАДНУЮ СЛУЖБУ, разработанную для шаблона A+ES, в котором используются лямбда-выражения. Однако наша

ПРОЕКЦИЯ реагирует на СОБЫТИЯ, а не на КОМАНДЫ, и обновляет документы с помощью интерфейса IDocumentWriter, а не с помощью обновления экземпляров АГРЕГАТА.

Базовая МОДЕЛЬ ЧТЕНИЯ фактически представляет собой обычный **ОБЪЕКТ ПЕРЕДАЧИ ДАННЫХ (DATA TRANSFER OBJECT — DTO)** [Фаулер], который может быть сериализован и сохранен в базовом хранилище с помощью интерфейса IDocumentWriter.

```
[Serializable]
public class CustomerTransactions
{
    public IList<CustomerTransaction> Transactions =
       new List<CustomerTransaction>();

    public void AddTx(
       string name, CurrencyAmount change,
       CurrencyAmount balance, DateTime timeUtc)
    {
       Transactions.Add(new CustomerTransaction()
       {
            Name = name,
            Balance = balance,
            Change = change,
            TimeUtc = timeUtc
       });
    }
}

[Serializable]
public class CustomerTransaction
{
    public CurrencyAmount Change;
    public CurrencyAmount Balance;
    public string Name;
    public DateTime TimeUtc;
}
```

Как правило, для сохранения МОДЕЛИ ЧТЕНИЯ в документной базе данных могут использоваться другие возможности. Мы можем кешировать МОДЕЛИ ЧТЕНИЯ в памяти, передавать их как документы в сеть доставки контента или сохранять в таблицах реляционной базы данных.

Кроме масштабируемости, одним из главных преимуществ ПРОЕКЦИЙ является их полная доступность. Их можно добавлять, изменять или полностью заменять в любое время на всем протяжении существования приложения. Для замены всей МОДЕЛИ ЧТЕНИЯ отбросьте все существующие данные из МОДЕЛИ ЧТЕНИЯ и сгенерируйте новые данные, выполняя весь ПОТОК СОБЫТИЙ с помощью своих классов ПРОЕКЦИИ. Этот процесс можно автоматизировать. Можно даже предотвратить задержки при полной замене МОДЕЛИ ЧТЕНИЯ.

Комбинация с АГРЕГАТОМ

Такие ПРОЕКЦИИ МОДЕЛИ ЧТЕНИЯ часто используются для предоставления информации разным клиентам (например, настольным системам или пользовательским веб-интерфейсам), но они также часто оказываются полезными для распределения информации между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ и их АГРЕГАТАМИ. Рассмотрим сценарий, в котором АГРЕГАТУ `Invoice` необходима информация об экземпляре класса `Customer` (например, его имя, адрес и индивидуальный номер налогоплательщика), чтобы подготовить объект класса `Invoice`. Мы должны представить эту информацию в удобной для потребления форме с помощью объекта класса `CustomerBillingProjection`, который создаст и будет поддерживать эксклюзивный экземпляр класса `CustomerBillingView`. Эта МОДЕЛЬ ЧТЕНИЯ доступна для АГРЕГАТА `Invoice` благодаря СЛУЖБЕ ПРЕДМЕТНОЙ ОБЛАСТИ `IProvideCustomerBillingInformation`. Эта СЛУЖБА ПРЕДМЕТНОЙ ОБЛАСТИ просто запрашивает в документном хранилище соответствующий экземпляр `CustomerBillingView`.

ПРОЕКЦИИ позволяют также распределять информацию между экземплярами АГРЕГАТА с помощью более удобного способа, обеспечивающего слабую связанность. Если в какой-то момент времени нам потребуется изменить информацию, возвращенную экземпляром класса `IProvideCustomerBillingView`, мы можем сделать это, не модифицируя АГРЕГАТ `Customer`. Для этого достаточно просто изменить реализацию ПРОЕКЦИИ и перестроить МОДЕЛИ ЧТЕНИЯ, заменив все СОБЫТИЯ.

Расширение событий

Одной из основных проблем, связанных с шаблоном A+ES, является его двойственное предназначение. СОБЫТИЯ используются и для хранения АГРЕГАТОВ, и для публикации сообщений о происходящем в предметной области предприятия.

В качестве примера рассмотрим следующую ситуацию. Система управления проектом позволяет потребителям создавать новые проекты и архивировать выполненные. Представьте себе, что вы публикуете СОБЫТИЕ `ProjectArchived` каждый раз, когда пользователь архивирует проект. Это СОБЫТИЕ ПРЕДМЕТНОЙ ОБЛАСТИ может иметь следующую структуру.

```
public class ProjectArchived {
    public ProjectId Id { get; set; }
    public UserId ChangeAuthorId { get; set; }
    public DateTime ArchivedUtc { get; set; }
    public string OptionalComment { get; set; }
}
```

Этой информации достаточно для того, чтобы перестроить архивированный объект класса `Project` по шаблону A+ES. Однако в этом случае публикация СОБЫТИЙ становится проблематичной.

Почему? Рассмотрим ПРОЕКЦИЮ для представления `ArchivedProjectsPerCustomer`, как показано на рис. 16. Эта ПРОЕКЦИЯ является подписчиком на СОБЫТИЯ и поддерживает список архивированных проектов для пользователей. Для того чтобы выполнить работу, ПРОЕКЦИИ потребуется самая свежая информация о следующих аспектах.

- Имена проектов
- Имена пользователей
- Назначения проектов пользователям
- Архивные СОБЫТИЯ проекта

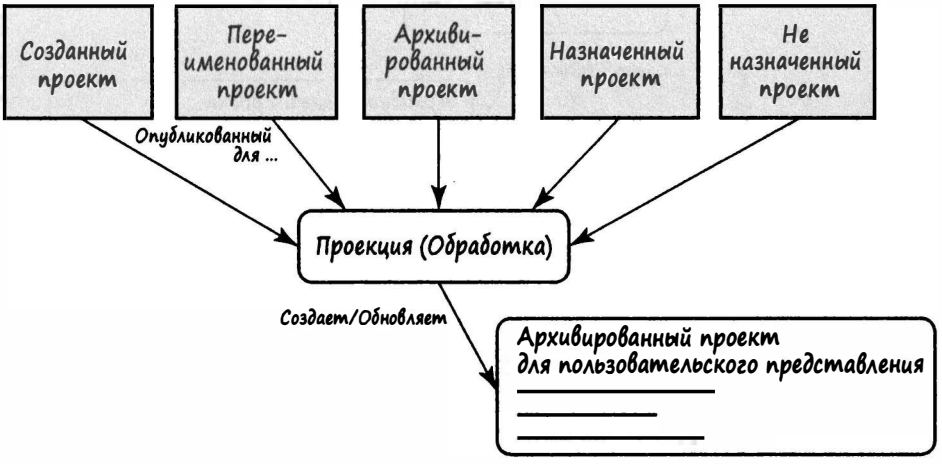


Рис. 16. Многочисленные СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ поступают в ПРОЕКЦИЮ и используются для создания представления МОДЕЛИ ЧТЕНИЯ

Эту ПРОЕКЦИЮ можно значительно упростить, расширив СОБЫТИЕ `ProjectArchived` за счет дополнительных данных-членов, содержащих релевантную информацию. Дополнительные данные-члены могут оказаться не имеющими значения для восстановления состояния соответствующего АГРЕГАТА, но при этом существенно упрощать работу клиентов с СОБЫТИЯМИ. Рассмотрим альтернативный контракт СОБЫТИЯ.

```
public class ProjectArchived {
    public ProjectId Id { get; set; }
    public string ProjectName { get; set; }
    public UserId ChangeAuthorId { get; set; }
```

```
    public DateTime ArchivedUtc { get; set; }
    public string OptionalComment { get; set; }
    public CustomerId Customer { get; set; }
    public string CustomerName { get; set; }
}
```

С помощью расширенного СОБЫТИЯ объект класса `ArchivedProjectsPerCustomerView`, генерируемого ПРОЕКЦИЕЙ, можно упростить так, как показано на рис. 17.

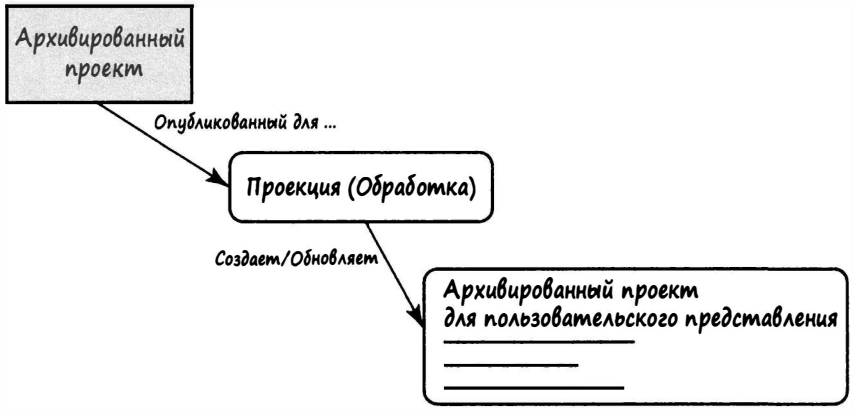


Рис. 17. СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ, **такие как** `ProjectArchived`, **могут обрабатываться процессорами** ПРОЕКЦИИ **для генерации** МОДЕЛЕЙ ЧТЕНИЯ, **предназначенных для представлений и отчетов**

Эмпирическое правило разработки СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ утверждает, что они должны содержать информацию, удовлетворяющую 80% подписчиков, даже если для этого СОБЫТИЯ должны будут содержать больше информации, чем требуется значительному количеству подписчиков. Напомним, что мы хотим гарантировать, чтобы процессоры ПРОЕКЦИИ имели большой набор данных о СОБЫТИИ, в который входят следующие данные.

- Идентификаторы СОБЫТИЙ, которые являются их владельцами, как, например, класс `CustomerId` по отношению к классу `Customer`.
- Имена и другие свойства, которые обычно используются для отображения информации на дисплее, как, например, классы `ProjectName`, `CustomerName` и им подобные.

Это рекомендации, а не правила. Обычно они вполне оправданны на предприятиях с большим количеством ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Монолитные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ меньше соответствуют этим правилам, поскольку они, как правило, поддерживают вторичные таблицы поиска и карты СОБЫТИЙ.

Разумеется, вам лучше знать, какие свойства следует включать в ваши СОБЫТИЯ. Иногда совершенно очевидно, какие свойства должны включаться в свойства СОБЫТИЙ практически без рефакторинга.

Вспомогательные средства и шаблоны

Для разработки, создания, развертывания и обслуживания систем на основе подхода A+ES необходим набор шаблонов, которые могут несколько отличаться от шаблонов для традиционных систем. В этом разделе представлены шаблоны, инструменты и методы, которые оказались довольно полезными при использовании подхода A+ES.

Механизмы сериализации событий

Правильно выбранный метод сериализации должен поддерживать изменение номеров версий и имен СОБЫТИЙ. Это особенно важно в тех проектах A+ES, в которых модель предметной области может быстро эволюционировать. Рассмотрим СОБЫТИЕ, объявленное с помощью реализации аннотаций на языке Protocol Buffers[1] на платформе .NET.

```
[DataContract]
public class ProjectClosed {
    [DataMember(Order=1)] public long ProjectId { get; set; }
    [DataMember(Order=2)] public DateTime Closed { get; set; }
}
```

Теперь, если мы захотим сериализовать объект класса `ProjectClosed` с помощью объектов классов `DataContractSerializer` или `JsonSerializer`, а не языка Protocol Buffers, любой переименованный член может легко нарушить работоспособность зависящих от него клиентов. Например, допустим, что мы меняем имя свойства `Closed` на `ClosedUtc`. Если не предпринять специальных мер для отображения переименованного свойства в клиентском ОГРАНИЧЕННОМ КОНТЕКСТЕ, возникнет ошибка.

```
[DataContract]
public class ProjectClosed {
    [DataMember] public long ProjectId { get; set; }
    [DataMember(Name="Closed")] public DateTime ClosedUtc { get; set; }
}
```

[1] Язык Protocol Buffers был изобретен компанией Google. Остальные создали его реализацию на платформе .NET.

С помощью языка Protocol Buffers можно справляться с изменяющейся сериализацией, потому что он отслеживает элементы контракта по целочисленным дескрипторам, а не именам. Как показано в следующем коде, клиенты могут успешно использовать свойство как с именем `CloseUtc`, так и с именем `Close`. Благодаря этому сериализация объектов в очень компактное двоичное представление происходит чрезвычайно быстро. Используя язык Protocol Buffers, мы можем переименовывать свойства СОБЫТИЙ, не беспокоясь об обратной совместимости и уменьшая сложность разработки в развивающейся модели предметной области.

```
[DataContract]
public class ProjectClosed {
    [DataMember(Order=1)] public long ProjectId { get; set; }
    [DataMember(Order=2)] public DateTime ClosedUtc { get; set; }
}
```

К дополнительным кроссплатформенным средствам сериализации относятся Apache Thrift, Avro и MessagePack, обеспечивающие дополнительные возможности.

Неизменяемость события

ПОТОКИ СОБЫТИЙ по своей природе считаются неизменяемыми. Для того чтобы модель разработки оставалась согласованной с этой концепцией (и чтобы избежать нежелательных побочных эффектов), контракты СОБЫТИЙ следует реализовывать как неизменяемые. Для того чтобы сделать это на языке C# на платформе .NET, необходимо пометить поля как предназначенные только для чтения и задавать значения только с помощью конструктора. Вернемся к предыдущему СОБЫТИЮ `ProjectClosed` и напишем его реализацию как неизменяемого СОБЫТИЯ.

```
[DataContract]
public class ProjectClosed {
   [DataMember(Order=1)] public long ProjectId { get; private set }
   [DataMember(Order=2)] public DateTime ClosedUtc { get; private set; }
   public ProjectClosed(long projectId, DateTime closedUtc)
   {
       ProjectId = projectId;
       ClosedUtc = closedUtc;
    }
}
```

Объекты-значения

Как уже говорилось в главе, посвященной **ОБЪЕКТАМ-ЗНАЧЕНИЯМ (6)**, этот шаблон может значительно упростить разработку и эволюцию сложных моделей

предметной области. Используя ОБЪЕКТЫ-ЗНАЧЕНИЯ, мы связываем элементарные типы в явно именованный неизменяемый тип. Например, вместо объявления идентификатора проекта как `long` мы моделировали бы явный тип `ProjectId`.

```
public struct ProjectId
{
    public readonly long Id { get; private set; }
    public ProjectId(long id)
    {
       Id = id
    }

    public override ToString() {
       return string.Format("Project-{0}", Id);
    }
}
```

Тип `long` по-прежнему используется для хранения фактического идентификационного номера, а тип `ProjectId` позволяет отличить его от всех остальных. Очевидно, что типы значений не ограничиваются уникальными идентификаторами. К другим соответствующим типам ЗНАЧЕНИЙ относятся деньги (особенно в многовалютных системах), почтовые адреса, адреса электронной почты, единицы измерения и др.

Кроме содержательности и выразительности контрактов СОБЫТИЙ и КОМАНД, ОБЪЕКТЫ-ЗНАЧЕНИЯ предметной области дают практические преимущества для реализации подхода A+ES, такие как статическая проверка типов и поддержка интегрированной среды разработки. Рассмотрим следующий сценарий, в котором разработчик может случайно перепутать параметры конструктора простого события, передав их в неправильном порядке.

```
long customerId = ...;
long projectId = ...;
var event = new ProjectAssignedToCustomer(customerId, projectId);
```

Эта ошибка может остаться незамеченной компилятором и быть обнаружена только после долгой отладки. Однако если в качестве идентификаторов используются ОБЪЕКТЫ-ЗНАЧЕНИЯ, то компилятор (а значит, и редактор интегрированной среды разработки) сможет выявить ошибку, которая заключается в том, что объект класса `CustomerId` передается первым, а объект класса `ProjectId` — вторым.

```
CustomerId customerId = ...;
ProjectId projectId = ...;
var event = new ProjectAssignedToCustomer(customerId, projectId);
```

Преимущества этого решения становятся еще более очевидными для простых контрактов классов с большим количеством полей. Например, рассмотрим следующее СОБЫТИЕ (это упрощенный вариант СОБЫТИЯ из реального проекта).

```
public class CustomerInvoiceWritten {
    public InvoiceId Id { get; private set; }
    public DateTime CreatedUtc { get; private set; }
    public CurrencyType Currency { get; private set; }
    public InvoiceLine[] Lines { get; private set; }
    public decimal SubTotal { get; private set; }

    public CustomerId Customer { get; private set; }
    public string CustomerName { get; private set; }
    public string CustomerBillingAddress { get; private set; }
    public float OptionalVatRatio { get; private set; }
    public string OptionalVatName { get; private set; }
    public decimal VatTax { get; private set; }
    public decimal Total { get; private set; }
}
```

Легко видеть, что работа с классом, имеющим так много свойств[2], может оказаться затруднительной. Мы можем выполнить рефакторинг этого крупного СОБЫТИЯ, чтобы оно стало более ясным и читабельным, уточнив его модель с помощью существующей концепции предметной области.

```
public class CustomerInvoiceWritten {
    public InvoiceId Id { get; private set; }
    public InvoiceHeader Header { get; private set; }
    public InvoiceLine[] Lines { get; private set; }
    public InvoiceFooter Footer { get; private set; }
}
```

Классы `InvoiceHeader` и `InvoiceFooter` связывают свойства друг с другом.

```
public class InvoiceHeader {
    public DateTime CreatedUtc { get; private set; }
    public CustomerId Customer { get; private set; }
    public string CustomerName { get; private set; }
    public string CustomerBillingAddress { get; private set; }
}

public class InvoiceFooter {
    public CurrencyAmount SubTotal { get; private set; }
    public VatInformation OptionalVat { get; private set; }
    public CurrencyAmount VarAmount { get; private set; }
    public CurrencyAmount Total { get; private set; }
}
```

[2] Опыт подтверждает эмпирическое правило: в классе не должно быть больше 5–7 членов.

Мы заменили отдельные свойства `CurrencyType Currency` и `decimal SubTotal` ОБЪЕКТОМ-ЗНАЧЕНИЕМ `CurrencyAmount`. Дополнительное преимущество состоит в том, что этот класс можно усилить функциями проверки работоспособности, которая предотвращает операции между суммами, выраженными в различных валютах, и другие некорректные операции. То же самое относится и к объединению информации о VAT в отдельном ОБЪЕКТЕ-ЗНАЧЕНИИ внутри класса `InvoiceFooter` с другими суммами счетов.

Везде, где возможно, следует стремиться использовать ОБЪЕКТЫ-ЗНАЧЕНИЯ для объектов КОМАНД, СОБЫТИЙ или компонентов АГРЕГАТОВ.

Очевидно, что использование ОБЪЕКТОВ-ЗНАЧЕНИЙ в КОМАНДАХ и/или СОБЫТИЯХ потребовало бы совместного развертывания или даже создания **ОБЩЕГО ЯДРА (3)**. Для некоторых очень сложных предметных областей может потребоваться разработка ОБЪЕКТОВ-ЗНАЧЕНИЙ с разветвленной бизнес-логикой. Включение таких ОБЪЕКТОВ-ЗНАЧЕНИЙ в ОБЩЕЕ ЯДРО просто для обеспечения десериализации, безопасной с точки зрения типов, вероятно, привело бы к ненадежному проекту. Благодаря этому мы могли бы различать простые совместно используемые классы для десериализации КОМАНД и данные о СОБЫТИИ от более сложных классов, необходимых для **СМЫСЛОВОГО ЯДРА (2)**, безопасным с точки зрения типов способом. Для этого пришлось бы создать два набора классов ОБЪЕКТОВ-ЗНАЧЕНИЙ — классов, используемых исключительно СМЫСЛОВЫМ ЯДРОМ, и классов, развернутых вместе с классами КОМАНД и СОБЫТИЙ. Данные, сохраненные этими двумя наборами классов по мере необходимости могут преобразовываться друг в друга.

В зависимости от вашего вкуса копирование классов может показаться слишком сложным. В таком случае, возможно, целесообразно рассмотреть другой подход. В частности, можно стандартизовать сериализованные СОБЫТИЯ как **ОБЩЕДОСТУПНЫЙ ЯЗЫК (3)**. Как объяснялось в главе, посвященной **ИНТЕГРАЦИИ ОГРАНИЧЕННЫХ КОНТЕКСТОВ (13)**, вы можете принять решение использовать уведомления о СОБЫТИИ, используя подход, основанный на динамическом контроле типов. Это избавило бы вас от необходимости развертывания типов СОБЫТИЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ у подписчиков. Как у всех подходов, у этого решения есть компромиссы, которые должны быть тщательно взвешены.

Генерация контрактов

Поддерживать сотни контрактов СОБЫТИЙ (и КОМАНД) вручную очень утомительно и ненадежно. Лучше выразить их определения с помощью компактного предметно-ориентированного языка (domain-specific language — DSL), позволяющего генерировать простой код и строить правильные классы. Существует

несколько способов сформулировать синтаксис языка DSL, в частности — формат `.proto` языка Protocol Buffer и его аналоги. Например, может оказаться полезным следующий подход.

```
CustomerInvoiceWritten!(InvoiceId Id, InvoiceHeader header,
    InvoiceLine[] lines, InvoiceFooter footer)
```

Простой генератор кода может использовать лексический анализатор DSL для генерации кода, соответствующего каждой строке исходного кода. Рассмотрим пример, в котором генерируется класс `CustomerInvoiceWritten`.

```
[DataContract]
public sealed class CustomerInvoiceWritten : IDomainEvent {
    [DataMember(Order=1) public InvoiceId Id
       { get; private set; }
    [DataMember(Order=2) public InvoiceHeader Header
       { get; private set; }
    [DataMember(Order=3) public InvoiceLine[] Lines
       { get; private set; }
    [DataMember(Order=4) public InvoiceFooter Footer
       { get; private set; }
    public CustomerInvoiceWriter(
       InvoiceId id, InvoiceHeader header, InvoiceLine[] lines,
       InvoiceFooter footer)
       {
          Id = id;
          Header = header;
          Lines = lines;
          Footer = footer;
       }

       // Требуется механизм сериализации
       ProjectClosed() {
          Lines = new InvoiceLine[0];
    }
}
```

Перечислим практические выгоды, которые приносит этот подход.

- Он уменьшает сложность разработки, ускоряя итерации моделирования предметной области.
- Он уменьшает вероятность человеческих ошибок.
- Компактное представление позволяет удерживать все определения СОБЫТИЙ на одном экране, обеспечивая широкий обзор для улучшенного понимания. Это может даже служить кратким глоссарием ЕДИНОГО ЯЗЫКА.
- Мы можем поддерживать версии и распределять контракты СОБЫТИЙ в виде компактных определений вместо того, чтобы требовать исходный или

двоичный код. Это может даже усилить сотрудничество между различными командами.

То же самое можно сказать о контрактах КОМАНД. Открытая реализация инструмента для генерации кода на основе языка DSL доступна вместе с остальными примерами на сайте, где размещен учебный проект.

Модульное тестирование и спецификации

Рассмотрим преимущества, которые можно получить с помощью ИСТОЧНИКА СОБЫТИЙ при создании модульных тестов. Наши тесты можно легко сформулировать в формате *Дано-Условие-Ожидаемый результат.*

1. ***Дано:*** прошлые СОБЫТИЯ.
2. ***Условие:*** вызван метод АГРЕГАТА.
3. ***Ожидаемый результат:*** следующие СОБЫТИЯ *или* исключение.

Изучим работу этой схемы. Прошлые СОБЫТИЯ используются для настройки состояния АГРЕГАТА в начале модульного теста. Затем мы выполняем тестируемый метод АГРЕГАТА, передавая ему, если потребуется, тестовые аргументы и имитации СЛУЖБ ПРЕДМЕТНОЙ ОБЛАСТИ. В заключение мы формулируем ожидаемые результаты, сравнивая СОБЫТИЯ, порожденные агрегатом, с ожидаемыми СОБЫТИЯМИ.

Этот подход позволяет нам выявлять и проверять поведение каждого АГРЕГАТА. В то же время мы не вмешиваемся во внутреннее состояние АГРЕГАТА. Это позволяет уменьшить *хрупкость* теста, потому что группы разработчиков могут как угодно изменять и оптимизировать реализации агрегата до тех пор, пока выполняются контракты поведенческих функций, что подтверждается тестовыми модулями.

Можно продвинуться еще на один шаг вперед, непосредственно выразив пункт *Условие* с помощью КОМАНДЫ, которая передается надлежащей ПРИКЛАДНОЙ СЛУЖБЕ, содержащей тестируемый АГРЕГАТ. В таком случае модульный тест представляет собой спецификацию, выраженную полностью в терминах нашего ЕДИНОГО ЯЗЫКА, либо в виде кода, либо в виде созданного языка DSL.

Добавив немного кода, эти спецификации можно автоматически распечатать для использования экспертами в предметной области. Эти определения сценариев использования могут помочь проектным командам разбираться в предметных областях со сложным поведением и повысить эффективность их моделирования.

Рассмотрим простую спецификацию в виде текстового документа.

```
[Passed] Use case 'Add Customer Payment - Unlock On Payment'.

Given:
   1. Created customer 7 Eur 'Northwind' with key c67b30 ...
   2. Customer locked

When:
   Add 'unlock' payment 10 EUR via unlock

Expectations:
   [ok] Tx 1: payment 10 EUR 'unlock' (none)
   [ok] Customer unlocked
```

Читатели, заинтересовавшиеся этим подходом, могут поискать в Интернете ключевые слова "Event Sourcing Specifications" и найти подробное руководство на эту тему.

ИСТОЧНИКИ СОБЫТИЙ в функциональных языках

Шаблоны реализации, обрисованные в общих чертах ранее, фокусировались на объектно-ориентированном подходе, который является подходящим вариантом для таких языков программирования, как Java и C#. Однако шаблон ИСТОЧНИК СОБЫТИЯ по своей природе является функциональным. Таким образом, его можно успешно реализовать с помощью таких функциональных языков, как F# и Clojure. В этом случае можно получить более лаконичный код с оптимальной производительностью.

Перечислим некоторые особенности перехода от объектно-ориентированного подхода к функциональному при реализации АГРЕГАТОВ.

- Мы должны перейти от разработки изменяемого объектно-ориентированного объекта состояния АГРЕГАТА к проектированию простой неизменяемой записи состояния с набором модифицирующих функций. Эти функции получают запись состояния и параметры СОБЫТИЯ, возвращая как результат новую запись состояния. Это напоминает проектирование неизменяемого ОБЪЕКТА-ЗНАЧЕНИЯ, при котором ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ лишь производят новые ЗНАЧЕНИЯ на основе его собственного состояния и аргументов функции. Такие функции принимают форму `Func<State, Event, State>`.
- Текущее состояние АГРЕГАТА может быть определено как левая свертка (left fold) всех прошлых СОБЫТИЙ, которые были переданы модифицирующим функциям.

- Методы АГРЕГАТА могут также быть преобразованы в набор функций, не сохраняющих состояние, которые получают параметры КОМАНДЫ, СЛУЖБЫ ПРЕДМЕТНОЙ ОБЛАСТИ и состояние. Такие функции возвращают одно или несколько СОБЫТИЙ или не возвращают ни одного и принимают форму `Func<TArg1, TArg2..., State, Event[]>`.
- ХРАНИЛИЩЕ СОБЫТИЙ может рассматриваться как *функциональная база данных*, потому что оно сохраняет параметры для функций, которые изменяют состояние АГРЕГАТОВ. Поддержка снимков в функциональном ХРАНИЛИЩЕ СОБЫТИЙ знакома функциональным программистам под именем *запоминание результатов* (memoization).

Отражая основные бизнес-концепции с помощью шаблона A+ES и функционального языка программирования, можно значительно ускорить моделирование предметной области. Кроме того, это вынуждает нас перенести фокус внимания при исследовании предметной области со структуры АГРЕГАТА на выражение поведения в терминах ЕДИНОГО ЯЗЫКА предметной области. Присвоение СМЫСЛОВОМУ ЯДРУ более высокого приоритета по сравнению с технологией, скорее всего, окажется более ценным для бизнеса и даст более сильное конкурентное преимущество.

Библиография

- [Appleton, LoD] Appleton, Brad. n.d. "Introducing Demeter and Its Laws." www.bradapp.com/docs/demeter-intro.html.
- [Bentley] Bentley, Jon. 2000. *Programming Pearls, Second Edition*. Boston, MA: Addison-Wesley. http://cs.bell-labs.com/cm/cs/pearls/bote.html.
- [Brandolini] Brandolini, Alberto. 2009. "Strategic Domain-Driven Design with Context Mapping." www.infoq.com/articles/ddd-contextmapping.
- [Buschmann et al.] Buschmann, Frank, et al. 1996. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. New York: Wiley.*
- [Cockburn] Cockburn, Alastair. 2012. "Hexagonal Architecture." http://alistair.cockburn.us/Hexagonal+architecture.
- [Crupi et al.] Crupi, John, et al. n.d. "Core J2EE Patterns." http://corej2eepatterns.com/Patterns2ndEd/DataAccessObject.htm.
- [Cunningham, Checks] Cunningham, Ward. 1994. "The CHECKS Pattern Language of Information Integrity." http://c2.com/ppr/checks.html.
- [Cunningham, Whole Value] Cunningham, Ward. 1994. "1. Whole Value." http://c2.com/ppr/checks.html#1.
- [Cunningham, Whole Value aka Value Object] Cunningham, Ward. 2005. "Whole Value." http://fit.c2.com/wiki.cgi?WholeValue.
- [Dahan, CQRS] Dahan, Udi. 2009. "Clarified CQRS." www.udidahan.com/2009/12/09/clarified-cqrs/.
- [Dahan, Roles] Dahan, Udi. 2009. "Making Roles Explicit." www.infoq.com/presentations/Making-Roles-Explicit-Udi-Dahan.
- [Deutsch] Deutsch, Peter. 2012. "Fallacies of Distributed Computing." http://en.wikipedia.org/wiki/Fallacies_of_Distributed_Computing.
- [Dolphin] Object Arts. 2000. "Dolphin Smalltalk; Twisting the Triad." www.object-arts.com/downloads/papers/TwistingTheTriad.PDF.
- [Erl] Erl, Thomas. 2012. "SOA Principles: An Introduction to the Service-Oriented Paradigm." http://serviceorientation.com/index.php/serviceorientation/index.
- [Evans] Evans, Eric. 2004. *Domain-Driven Design: Tackling the Complexity in the Heart of Software.* Boston, MA: Addison-Wesley.
- [Evans, Ref] Evans, Eric. 2012. "Domain-Driven Design Reference." http://domainlanguage.com/ddd/patterns/DDD_Reference_2011-01-31.pdf.
- [Evans & Fowler, Spec] Evans, Eric, and Martin Fowler. 2012. "Specifications." http://martinfowler.com/apsupp/spec.pdf.

- [Fairbanks] Fairbanks, George. 2011. *Just Enough Software Architecture.* Marshall & Brainerd.
- [Fowler, Anemic] Fowler, Martin. 2003. "AnemicDomainModel." `http://martinfowler.com/bliki/AnemicDomainModel.html`.
- [Fowler, CQS] Fowler, Martin. 2005. "CommandQuerySeparation." `http://martinfowler.com/bliki/CommandQuerySeparation.html`.
- [Fowler, DI] Fowler, Martin. 2004. "Inversion of Control Containers and the Dependency Injection Pattern." `http://martinfowler.com/articles/injection.html`.
- [Fowler, P of EAA] Fowler, Martin. 2003. *Patterns of Enterprise Application Architecture.* Boston, MA: Addison-Wesley.
- [Fowler, PM] Fowler, Martin. 2004. "Presentation Model." `http://martinfowler.com/eaaDev/PresentationModel.html`.
- [Fowler, Self Encap] Fowler, Martin. 2012. "SelfEncapsulation." `http://martinfowler.com/bliki/SelfEncapsulation.html`.
- [Fowler, SOA] Fowler, Martin. 2005. "ServiceOrientedAmbiguity." `http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html`.
- [Freeman et al.] Freeman, Eric, Elisabeth Robson, Bert Bates, and Kathy Sierra. 2004. *Head First Design Patterns.* Sebastopol, CA: O'Reilly Media.
- [Gamma et al.] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns.* Reading, MA: Addison-Wesley.
- [Garcia-Molina & Salem] Garcia-Molina, Hector, and Kenneth Salem. 1987. "Sagas." ACM, Department of Computer Science, Princeton University, Prince ton, NJ. `www.amundsen.com/downloads/sagas.pdf`.
- [GemFire Functions] 2012. VMware vFabric 5 Documentation Center. `http://pubs.vmware.com/vfabric5/index.jsp?topic=/com.vmware.vfabric.gemfire.6.6/developing/function_exec/chapter_overview.html`.
- [Gson] 2012. A Java JSON library hosted on Google Code. `http://code.google.com/p/google-gson/`.
- [Helland] Helland, Pat. 2007. "Life beyond Distributed Transactions: An Apostate's Opinion." Third Biennial Conference on Innovative DataSystems Research (CIDR), January 7–10, Asilomar, CA. `www.ics.uci.edu/~cs223/papers/cidr07p15.pdf`.
- [Hohpe & Woolf] Hohpe, Gregor, and Bobby Woolf. 2004. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Systems. Boston, MA: Addison-Wesley.
- [Inductive UI] 2001. Microsoft Inductive User Interface Guidelines. `http://msdn.microsoft.com/en-us/library/ms997506.aspx`.

- [Jezequel et al.] Jezequel, Jean-Marc, Michael Train, and Christine Mingins. 2000. *Design Patterns and Contract*. Reading, MA: Addison-Wesley.
- [Keith & Stafford] Keith, Michael, and Randy Stafford. 2008. "Exposing the ORM Cache." ACM, May 1.
  `http://queue.acm.org/detail.cfm?id=1394141`.
- [Liskov] Liskov, Barbara. 1987. Conference Keynote: "Data Abstraction and Hierarchy."
  `http://en.wikipedia.org/wiki/Liskov_substitution_principle`
- "The Liskov Substitution Principle."
  `www.objectmentor.com/resources/articles/lsp.pdf`.
- [Martin, DIP] Martin, Robert. 1996. "The Dependency Inversion Principle."
  `www.objectmentor.com/resources/articles/dip.pdf`.
- [Martin, SRP] Martin, Robert. 2012. "SRP: The Single Responsibility Principle."
  `www.objectmentor.com/resources/articles/srp.pdf`.
- [MassTransit] Patterson, Chris. 2008. "Managing Long-Lived Transactions with MassTransit.Saga."
  `http://lostechies.com/chrispatterson/2008/08/29/managing-long-lived- transactions-with-masstransit-saga/`.
- [MSDN Assemblies] 2012.
  `http://msdn.microsoft.com/en-us/library/51ket42z%28v=vs.71%29.aspx`.
- [Nilsson] Nilsson, Jimmy. 2006. *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET*. Boston, MA: Addison-Wesley.
- [Nijof, CQRS] Nijof, Mark. 2009. "CQRS a la Greg Young."
  `http://cre8ivethought.com/blog/2009/11/12/cqrs--la-greg-young`.
- [NServiceBus] 2012.
  `www.nservicebus.com/`.
- [Öberg] Öberg, Rickard. 2012. "What Is Qi4j™?"
  `http://qi4j.org/`.
- [Parastatidis et al., RiP] Webber, Jim, Savas Parastatidis, and Ian Robinson. 2011. *REST in Practice*. Sebastopol, CA: O'Reilly Media.
- [PragProg, TDA] The Pragmatic Programmer. "Tell, Don't Ask."
  `http://pragprog.com/articles/tell-dont-ask`.
- [Quartz] 2012. Terracotta Quartz Scheduler.
  `http://terracotta.org/products/quartz-scheduler`.
- [Seoviæ] Seoviæ, Aleksandar, Mark Falco, and Patrick Peralta. 2010. *Oracle Coherence 3.5: Creating Internet-Scale Applications Using Oracle's High-Performance Data Grid*. Birmingham, England: Packt Publishing.
- [SOA Manifesto] 2009. SOA Manifesto.
  `www.soa-manifesto.org/`.

- [Sutherland] Sutherland, Jeff. 2010. "Story Points: Why Are They Better than Hours?" http://scrum.jeffsutherland.com/2010/04/story-points-why-are-they-better-than.html.
- [Tilkov, Manifesto] Tilkov, Stefan. 2009. "Comments on the SOA Manifesto." www.innoq.com/blog/st/2009/10/comments_on_the_soa_manifesto.html.
- [Tilkov, RESTful Doubts] Tilkov, Stefan. 2012. "Addressing Doubts about REST." www.infoq.com/articles/tilkov-rest-doubts.
- [Vernon, DDR] Vernon, Vaughn. n.d. "Architecture and Domain-Driven Design." http://vaughnvernon.co/?page_id=38.
- [Vernon, DPO] Vernon, Vaughn. n.d. "Architecture and Domain-Driven Design." http://vaughnvernon.co/?page_id=40.
- [Vernon, RESTful DDD] Vernon, Vaughn. 2010. "RESTful SOA or Domain-Driven Design—A Compromise?" QCon SF 2010. www.infoq.com/presentations/RESTful-SOA-DDD.
- [Webber, REST & DDD] Webber, Jim. "REST and DDD." http://skillsmatter.com/podcast/design-architecture/rest-and-ddd.
- [Wiegers] Wiegers, Karl E. 2012. "First Things First: Prioritizing Requirements." www.processimpact.com/articles/prioritizing.html.
- [Wikipedia, CQS] 2012. "Command-Query Separation." http://en.wikipedia.org/wiki/Command-query_separation.
- [Wikipedia, EDA] 2012. "Event-Driven Architecture." http://en.wikipedia.org/wiki/Event-driven_architecture.
- [Young, ES] Young, Greg. 2010. "Why Use Event Sourcing?" http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/.
- [Гамма и др.] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* — СПб.: Питер, 2013. — 368 с.
- [Фаулер и др.] Фаулер М. (при участии Дейвида Райса, Мэттью Фоммела, Эдварда Хайета, Роберта Ми и Рэнди Стаффорда). *Шаблоны корпоративных приложений.* — М.: ООО "И.Д. Вильямс", 2012. — 448 с.
- [Эванс] Эванс Э. *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем.* : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2012. — 448 с.

Предметный указатель

Программная инженерия

"Для разработчиков программного обеспечения любой квалификации, стремящихся повысить свой уровень в области проектирования и реализации предметно-ориентированных промышленных приложений с учетом лучших достижений профессиональной практики, книга *Реализация методов предметно-ориентированного проектирования* станет кладезем знаний, ценой больших усилий добытых специалистами в области DDD и архитектуры промышленных приложений за последние десятилетия".

Рэнди Стаффорд (Randy Stafford), архитектор больших проектов,разработчик Oracle Coherence

"Эта книга должна входить в круг чтения всех, кто ищет способы внедрения принципов DDD в практику".

Уди Дахан (Udi Dahan), создатель каркаса NServiceBus

Книга посвящена методам предметно-ориентированного проектирования (DDD). Автор придерживается принципа "от общего к частному", плавно переходя от стратегических шаблонов к средствам тактического программирования. Вон Вернон описывает специализированные подходы к реализации систем на основе современной архитектуры, подчеркивая важность ориентации на предметную область с учетом технических ограничений.

Опираясь на знаменитую книгу Эрика Эванса *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем*, автор описывает методы DDD на примерах известных предметных областей. Каждый принцип подкреплен реалистичными примерами на языке Java, которые представляют интерес и для программистов, работающих на языке C#. Все примеры объединены в рамках единого сценария разработки системы SaaS для многоарендной среды на основе методологии Scrum.

Автор выходит далеко за пределы "облегченного подхода DDD", в котором принципы DDD рассматриваются только с технической точки зрения, и показывает, как извлечь максимальную пользу из стратегических шаблонов предметно-ориентированного проектирования с помощью **ОГРАНИЧЕННЫХ КОНТЕКСТОВ**, **КАРТ КОНТЕКСТОВ** и **ЕДИНОГО ЯЗЫКА**. Используя описанные методы и приведенные примеры, разработчики смогут сократить время выхода на рынок и повысить качество проектов, создавая более гибкое, масштабируемое и ориентированное на достижение бизнес-целей программное обеспечение.

В книге изложены следующие темы.

- Правильная трактовка подхода DDD, позволяющая быстро получить выгоду от его применения.
- Использование подхода DDD в сочетании с разными архитектурными стилями, включая **ГЕКСАГОНАЛЬНУЮ АРХИТЕКТУРУ**, **SOA**, **REST**, **CQRS**, **СОБЫТИЙНО-ОРИЕНТИРОВАННУЮ АРХИТЕКТУРУ**, а также **ФАБРИКИ ДАННЫХ** и **РАСПРЕДЕЛЕННЫЕ ВЫЧИСЛЕНИЯ**.
- Правильное проектирование и применение **СОБЫТИЙ**, а также распознавание ситуаций, в которых вместо них следует использовать **ОБЪЕКТЫ-ЗНАЧЕНИЯ**.
- Новый метод предметно-ориентированного проектирования — **СОБЫТИЯ ПРЕДМЕТНОЙ ОБЛАСТИ**.
- Проектирование **ХРАНИЛИЩ** на основе ORM, NoSQL и других баз данных.

Вон Вернон — ветеран программирования, имеющий более чем 25-летний опыт разработки программ, проектов и архитектурных стилей. Он является признанным лидером в области упрощения проектирования и реализации программного обеспечения с помощью инновационных методов. С 1980-х годов он разрабатывает программы с помощью объектно-ориентированных языков программирования, а с начала 1990-х годов применяет методы предметно-ориентированного программирования, опираясь на свой опыт работы на языке предметно-ориентированного моделирования Simula. Он дает консультации, выступает на конференциях и проводит мастер-классы по предметно-ориентированному проектированию на многих континентах.

ВИЛЬЯМС
Издательский дом "Вильямс"
http://www.williamspublishing.com

Addison-Wesley
Pearson Education

ISBN 978-5-8459-1881-9
14006
9 785845 918819

ALWAYS LEARNING
PEARSON