Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра инженерной кибернетики

ОСНОВЫ ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Конспект лекций
для студентов специальности
5В070200 – Автоматизация и управление

Алматы 2015

СОСТАВИТЕЛЬ: Н.В.Сябина. Основы инженерии программного обеспечения. Конспект лекций для студентов специальности 5В070200 – Автоматизация и управление. – Алматы: АУЭС, 2015. – 78 с.

Настоящий конспект лекций содержит материалы для ознакомления студентов 4 курса с основными понятиями инженерии программного обеспечения и принципами проектирования программных продуктов. Конспект содержит теоретический материал по 15 темам. В приложения включены необходимые иллюстративные и справочные материалы.

Конспект лекций предназначен для студентов всех форм обучения специальности 5В070200 – Автоматизация и управление.

Ил.25, табл.4, библиогр. – 18 назв.

Рецензент: ст. преп., канд. техн. наук Мусапирова Г.Д.

Печатается по плану издания некоммерческого акционерного общества «Алматинский университет энергетики и связи» на 2014 г.

Содержание

Лекция №1. Введение в программную инженерию

Цель: получить представление о программной инженерии, о методах управления программными проектами, о классификации моделей программного обеспечения по «весу», а также о принципах организации эффективной команды разработчиков.

Программная инженерия - это применение определенного систематического измеримого подхода при разработке, эксплуатации и поддержке программного обеспечения (ПО) [1]. Термин *software* (программное обеспечение) ввел в 1958 году всемирно известный статистик Джон Тьюкей. Термин *software engineering* (программная инженерия). Он впервые появился в названии конференции НАТО, состоявшейся в Германии в 1968 году и посвященной так называемому кризису программного обеспечения. С 1990-го по 1995 год велась работа над международным стандартом, который должен был дать единое представление о процессах разработки программного обеспечения. В результате был выпущен стандарт ISO/IEC 12207, а в 2004 году в отрасли был создан основополагающий труд «Руководство к своду знаний по программной инженерии» (*SWEBOK*), в котором были собраны основные теоретические и практические знания, накопленные в этой отрасли.

Программирование — процесс отображения определенного множества целей на множество машинных команд и данных, интерпретация которых на компьютере или вычислительном комплексе обеспечивает достижение поставленных целей [1]. Цели могут быть любые: воспроизведение звука в динамике компьютера, расчет траектории полета космического аппарата на Марс, печать годового балансового отчета и т.д. Важно то, что они должны быть определены.

Профессиональное программирование – это деятельность, направленная на получение доходов при помощи программирования. Принципиальным отличием от просто программирования является то, что имеется или, по крайней мере, предполагается некоторый потребитель, который готов платить за использование программного продукта. Отсюда следует важный вывод о том, что профессиональное производство программ - это всегда коллективная деятельность, в которой участвуют, минимум, два человека: программист и потребитель.

За 50 лет развития программной инженерии накопилось большое количество моделей разработки программного обеспечения. Можно провести аналогию между историей развития методов, применяемых в системах автоматического управления, и эволюцией подходов к управлению программными проектами.

«Как получится» - разомкнутая система управления. Предполагает полное доверие техническим лидерам, представители бизнеса практически не

участвует в проекте. Планирование, если оно и есть, то неформальное и словесное. Время и бюджет, как правило, не контролируются. Аналогия: баллистический полет без обратной связи. Можно, но недалеко и неточно.

«Водопад», или каскадная модель, - жесткое управление с обратной связью. Расчет опорной траектории (план проекта), измерение отклонений, коррекция и возврат на опорную траекторию. Лучше, но не эффективно.

«Гибкое управление». Расчет опорной траектории, измерение отклонений, расчет новой попадающей траектории и коррекция для выхода на нее. «Планы — ничто, планирование — все» (Эйзенхауэр, Дуайт Дэвид).

«Метод частых поставок» - самонаведение. Расчет опорной траектории, измерение отклонений, уточнение цели, расчет новой попадающей траектории и коррекция для выхода на нее.

Классические методы управления перестают работать в случаях, когда структура и свойства управляемого объекта нам не известны и/или изменяются со временем. Эти подходы не помогут, если текущие свойства объекта не позволяют ему двигаться с требуемыми характеристиками. Например, летательный аппарат не может развить требуемое ускорение или разрушается при недопустимой перегрузке. Аналогично, если рабочая группа проекта не может обеспечить требуемую эффективность и поэтому постоянно работает в режиме аврала, то это приводит не к росту производительности, а к уходу профессионалов из проекта.

Когда структура и свойства управляемого объекта нам не известны, необходимо использовать *адаптивное управление*, которое, дополнительно к прямым управляющим воздействиям, направлено на изучение и изменение свойств управляемого объекта. Продолжая аналогию с управлением летательными аппаратами, это расчет опорной траектории, измерение отклонений, уточнение цели, уточнение объекта управления, адаптация (необходимое изменение) объекта управления, расчет новой попадающей траектории и коррекция для выхода на нее. Для того чтобы понять структуру и свойства объекта и воздействовать на него с целью их приведения к желаемому состоянию, в проекте должен быть дополнительный контур обратной связи — *контур адаптации*.

Известно, что производительность разных программистов может отличаться в десятки раз. Поэтому, помимо чисто управленческих задач, руководитель, если он стремится получить наивысшую производительность рабочей группы, должен направлять постоянные усилия на изучение и изменение объекта управления: людей и их взаимодействия.

Модели (*методологии*) процессов разработки программного обеспечения принято классифицировать по «*весу*» - количеству формализованных процессов и детальности их регламентации. Чем больше процессов документировано, чем более детально они описаны, тем больше «вес» модели. Наиболее распространенные современные модели процесса разработки ПО [1] представлены на рисунке А.1.

ГОСТ 19 «Единая система программной документации» и *ГОСТ 34 «Стандарты на разработку и сопровождение автоматизированных систем»* ориентированы на последовательный подход к разработке ПО. Разработка в соответствии с этими стандартами проводится по этапам, каждый из которых предполагает выполнение строго определенных работ, и завершается выпуском достаточно большого числа весьма формализованных и обширных документов. Строгое следование им не только приводит к водопадному подходу, но и требует очень высокой степени формализованности разработки.

В середине 80-х годов минувшего столетия по заказу Министерства обороны США Институт программной инженерии, входящий в состав Университета Карнеги-Меллона, разработал *SW-CMM, Capability Maturity Model for Software* в качестве эталонной модели организации разработки программного обеспечения, которая определяет пять уровней зрелости процесса разработки ПО: *начальный, повторяемый, определенный*, управляемый, оптимизируемый. Документация с полным описанием SW-CMM занимает около 500 страниц и определяет набор из 312 требований, которым должна соответствовать организация, если она планирует аттестоваться по этому стандарту на 5-ый уровень зрелости.

Унифицированный процесс (*Rational Unified Process, RUP*) был разработан компанией «Rational Software» в качестве дополнения к языку моделирования *UML*. Модель RUP описывает абстрактный общий процесс, на основе которого организация или проектная команда должна создать конкретный специализированный процесс, ориентированный на ее потребности. В результате общего построения RUP можно использовать и как основу для самого что ни на есть традиционного водопадного стиля разработки, так и в качестве гибкого процесса.

Microsoft Solutions Framework (*MSF*) — это гибкая и достаточно легковесная модель, построенная на основе итеративной разработки. Особенностью MSF является большое внимание к созданию эффективной и небюрократизированной проектной команды. Для достижения этой цели MSF предлагает достаточно нестандартные подходы к организационной структуре, распределению ответственности и принципам взаимодействия внутри команды.

Одна из последних разработок Института программной инженерии *Personal Software Process / Team Software Process* (*PSP/TSP*). PSP определяет требования к компетенциям разработчика. TSP делает ставку на самоуправляемые команды численностью 3-20 разработчиков. Последовательное применение модели PSP/TSP позволяет сделать нормой в организации пятый уровень CMM.

Основная идея всех гибких моделей заключается в том, что применяемый в разработке ПО процесс должен быть адаптивным. Они декларируют в первую очередь ориентированность на людей и их взаимодействие, а не на процессы и средства. По сути, гибкие методологии это не методологии, а набор практик, которые могут позволить добиваться

эффективной разработки ПО, основываясь на принципах *итеративности*, *инкрементальности*, *самоуправляемости команды* и *адаптивности процесса*.

Алистер Коуберн, один из авторов «Манифеста гибкой разработки ПО», проанализировал разные программные проекты, которые выполнялись на разных моделях от легких до тяжелых, и не обнаружил зависимости между успехом или провалом проектов и моделями процесса разработки. Отсюда он сделал вывод о том, что эффективность разработки ПО не зависит от модели процесса, т.е. не существует единственного правильного процесса разработки ПО. В каждом новом проекте процесс должен определяться каждый раз заново, в зависимости от проекта, продукта и персонала, в соответствие с «*Законом четырех П*» (рисунок А.2). Совершенно разные процессы должны применяться в проектах, имеющих различное назначение, а также в которых участвует разное количество человек с разным опытом ведения разработки.

«Закон четырех П» гласит: процесс должен определяться в зависимости от проекта, продукта и персонала. Команда, которая начинала проект, не остается неизменной, она проходит определенные стадии формирования и, как правило, количественно растет по мере развития проекта. Поэтому процесс должен постоянно адаптироваться к этим изменениям. Главный принцип: не люди должны подстраиваться под выбранную модель процесса, а модель процесса должна подстраиваться под конкретную команду, чтобы обеспечить ее наивысшую эффективность.

Задача программного проекта – это достижение конкретной бизнес-цели, при соблюдении ограничений «*железного треугольника*» (рисунок А.3). Это означает, что ни один из углов треугольника не может быть изменен без оказания влияния на другие. Например, чтобы уменьшить время, потребуется увеличить стоимость и/или сократить содержание.

Согласно текущей редакции стандарта *PMBOK,* проект считается успешным, если удовлетворены все требования заказчика и участников проекта. Поэтому проект разработки оценивают по четырем факторам:

- выполнение в соответствие со спецификациями;
- выполнение в срок;
- выполнение в пределах бюджета;
- удовлетворенность каждого участника команды.

Эффективность - это отношение полученного результата к произведенным затратам. Нельзя рассматривать эффективность, исходя только из результативности: чем больше производишь, чем больше делаешь, тем выше твоя эффективность. Затраты не следует путать с инвестициями.

Цели проекта должны отвечать на вопрос, зачем данный проект нужен. Цели проекта должны описывать бизнес-потребности и задачи, которые решаются в результате исполнения проекта. Целями проекта могут быть: изменения в компании, реализация стратегических планов, выполнение контрактов, разрешение специфических проблем. Цели должны быть значимыми, конкретными, измеримыми и, наконец, реальными. Четкое определение бизнес-целей важно, поскольку существенно влияет на все

процессы и решения в проекте. Проект должен быть закрыт, если признается, что достижение цели невозможно или стало нецелесообразным.

Результаты проекта отвечают на вопрос, что должно быть получено после его завершения. Результаты проекта должны определять:

– какие именно бизнес-выгоды получит заказчик в результате проекта;

– какой конкретно продукт или услуга будут произведены по окончании проекта;

– краткое описание и при необходимости ключевые свойства и/или характеристики продукта/услуги.

Следует помнить, что результаты должны быть *измеримыми*, т.е. при оценке результатов проекта должна иметься возможность сделать заключение достигнуты оговоренные в концепции результаты или нет.

Наконец, немаловажным моментом при разработке успешного проекта является *эффективность команды* разработчиков. Рабочая группа прежде, чем она станет эффективной командой, должна пройти четыре обязательных последовательных стадии:

а) *формирование* - характеризуется избытком энтузиазма, связанного с новизной. Люди должны преодолеть внутренние противоречия, переболеть конфликтами прежде, чем сформируется действительно спаянный коллектив. На этом этапе многое зависит от руководителя. Он должен четко поставить цели членам команды, верно определить роль каждого в проекте;

б) *разногласия и конфликты* - самый сложный и опасный период. Мотивация новизны уже исчезла, а сильные и глубокие стимулы у команды еще не появились. Неизбежные сложности или неудачи порождают конфликты и «поиск виновных». Участники команды методом проб и ошибок вырабатывают наиболее эффективные процессы взаимодействия. Руководителю на этом этапе важно обеспечить открытую коммуникацию в команде. Конфликты не следует прятать, а споры необходимо разрешать спокойно, терпеливо и тщательно;

в) *становление* - в команде растет доверие, люди начинают замечать в коллегах не только проблемные, но и сильные стороны. Закрепляются и оттачиваются наиболее эффективные процессы взаимодействия. На смену битве амбиций приходит продуктивное сотрудничество. Четче становится разделение труда, исчезает дублирование функций. Руководитель перестает находиться в состоянии постоянного аврала, работа по построению команды на этом этапе - скрупулезный труд по отработке общих норм и правил;

г) *отдача* - команда работает эффективно, высок командный дух, люди хорошо знают друг друга и умеют использовать сильные стороны коллег. Все стремятся придерживаться выработанных общих процессов. Высок уровень доверия. Это лучший период для раскрытия индивидуальных талантов. Задача менеджера на этом этапе - поддерживать требуемый уровень мотивации, искать новые пути и открывать новые возможности. Четыре стадии развития команды должны циклически повторяться, чтобы обеспечить непрерывный рост производительности.

Лекция №2. Ресурсы. Управление рисками проекта

Цель: получить представление о ресурсах, сроках разработки, рисках проекта и возможностях их идентификации.

Чтобы понять, сколько будет стоить реализация программного проекта, требуется определить и оценить *ресурсы*, необходимые для его выполнения, а именно: человеческие ресурсы и требования к квалификации персонала; оборудование, услуги, расходные материалы, лицензии на ПО, критические компьютерные ресурсы; бюджет проекта - план расходов и предполагаемых доходов проекта с разбивкой по статьям и фазам/этапам проекта.

Специфика программного проекта заключается в том, что человеческие ресурсы вносят основной вклад в его стоимость. Все остальные затраты, как правило, незначительны, по сравнению с этим расходами [1]. Необходимо помнить, что, помимо непосредственно программирования в проекте разработки ПО, есть много других процессов, которые требуют ресурсы соответствующей квалификации, а само программирование составляет лишь четверть всех затрат. Пример распределения трудозатрат по основным производственным процессам при современном процессе разработки ПО показан на рисунке А.4.

Немаловажной составляющей проекта является оценка *сроков* его разработки. Ф. Брукс в работе [2] приводит эмпирическую формулу оценки срока проекта по его трудоемкости, выведенную Б.Боэмом на основе анализа результатов 63 проектов разработки ПО в аэрокосмической области. Графическое описание закона Боэма показано на рисунке А.5. Согласно этой формуле, для проекта, общая трудоемкость которого составляет N ч.*м. (человеко-месяцев), можно утверждать, что:

– существует оптимальное время выполнения графика для первой поставки: $T = 2{,}5(N\ ч.*м.)^{1/3}$, то есть оптимальное время в месяцах пропорционально кубическому корню предполагаемого объема работ в человеко-месяцах. Следствием является кривая, дающая оптимальную численность проектной команды (рисунок А.5);

– кривая стоимости медленно растет, если запланированный график длиннее оптимального, т.е. работа занимает все отведенное для нее время;

– кривая стоимости резко растет, если запланированный график короче оптимального, т.е. практически ни один проект невозможно завершить быстрее, чем за 3/4 расчетного оптимального графика вне зависимости от количества занятых в нем.

Для серьезного программного проекта недостаточно определить только срок его завершения. Необходимо еще определить его этапы - контрольные точки, в которых будет происходить переоценка проекта на основе реально достигнутых показателей. *Контрольная точка* - важный момент или событие в расписании проекта, отмечающее достижение заданного результата и/или

начало/завершение определенного объема работы. Каждая контрольная точка характеризуется датой и объективными критериями ее достижения.

Риск - это проблема, которая еще не возникла, а *проблема* — это риск, который материализовался. Характеристики риска [1]:

а) *причина* или *источник* – явление (обстоятельство), обусловливающее наступление риска;

б) *симптомы* риска - указание на то, что событие риска произошло или вот-вот произойдет;

в) *последствия* риска - проблема или возможность, которая может реализоваться в проекте в результате произошедшего риска;

г) *влияние* риска - влияние реализовавшегося риска на возможность достижения целей проекта.

Риск - это всегда вероятность и последствия. Пример характеристик риска приведен на рисунке А.6. Выделяют две категории рисков:

а) «*известные неизвестные*» - это те риски, которые можно идентифицировать и подвергнуть анализу, кроме того, в отношении таких рисков можно спланировать ответные действия;

б) «*неизвестные неизвестные*» - риски, которые невозможно идентифицировать и, следовательно, спланировать ответные действия.

Управление рисками - это определенная деятельность, которая выполняется в проекте от его начала до завершения. Как и любая другая работа, в проекте управление рисками требует времени и затрат ресурсов. Поэтому эта работа обязательно должна планироваться. *Планирование управления рисками* - это процесс определения подходов и планирования операций по управлению рисками проекта. Тщательное и подробное планирование управления рисками позволяет: выделить достаточное количество времени и ресурсов для выполнения операций по управлению рисками, определить общие основания для оценки рисков, а также повысить вероятность успешного достижения результатов проекта.

Планирование управления рисками должен быть завершено на ранней стадии планирования проекта, поскольку оно крайне важно для успешного выполнения других процессов.

Шкала оценки воздействия отражает значимость риска в случае его возникновения. Шкала оценки воздействия может различаться в зависимости от потенциально затронутой риском цели, типа и размера проекта, принятыми в организации стратегиями и его финансовым состоянием, а также от чувствительности организации к конкретному виду воздействий.

Еще одной важной характеристикой риска является *близость* его *наступления*.

Идентификация рисков - это выявление рисков, способных повлиять на проект, и документальное оформление их характеристик. Это итеративный процесс, который периодически повторяется на всем протяжении проекта, поскольку в рамках его жизненного цикла могут обнаруживаться новые

риски. Исходные данные для выявления и описания характеристик рисков могут браться из разных источников.

Каждый проект задумывается и разрабатывается на основании ряда гипотез, сценариев и допущений. Как правило, в описании содержания проекта перечисляются принятые допущения - факторы, которые для целей планирования считаются верными, реальными или определенными без привлечения доказательств. Неопределенность в допущениях проекта следует также обязательно рассматривать в качестве потенциального источника возникновения рисков проекта. Анализ допущения позволяет идентифицировать риски проекта, происходящие от неточности, несовместимости или неполноты допущений.

Для сбора информации о рисках могут применяться различные подходы. Среди этих подходов наиболее распространены *опрос экспертов*, *мозговой штурм*, *метод Дельфи*, *карточки Кроуфорда*. В качестве источника информации при выявлении рисков могут служить различные доступные контрольные списки рисков проектов разработки ПО, которые следует проанализировать на применимость к данному конкретному проекту [1, 3]. Не существует исчерпывающих контрольных списков рисков программного проекта, поэтому необходимо внимательно анализировать особенности каждого конкретного проекта. Результатом идентификации рисков должен стать список рисков с описанием их основных характеристик: причины, условия, последствий и ущерба.

Качественный анализ рисков включает в себя расстановку рангов для идентифицированных рисков. При анализе вероятности и влияния предполагается, что никаких мер по предупреждению рисков не производится. Качественный анализ рисков включает:

– определение вероятности реализации рисков;
– определение тяжести последствий реализации рисков;
– определения ранга риска по матрице «вероятность - последствия»;
– определение близости наступления риска;
– оценку качества использованной информации.

Результаты качественного анализа используются в ходе последующего количественного анализа рисков и планирования реагирования на риски.

Количественный анализ производится в отношении тех рисков, которые в процессе качественного анализа были квалифицированы как имеющие высокий и средний ранг. Для количественного анализа рисков могут быть использованы следующие методы:

– *анализ чувствительности* помогает определить, какие риски обладают наибольшим потенциальным влиянием на проект;

– *анализ дерева решений*, описывающего ситуацию с учетом каждой из имеющихся возможностей выбора и возможного сценария;

– *моделирование и имитация* - используется модель для определения последствий от воздействия подробно описанных неопределенностей на результаты проекта в целом.

После проведения качественного и количественного анализа рисков начинается процесс разработки путей и определения действий по увеличению возможностей и снижению угроз для целей проекта - *планирование реагирования на риски*.

Возможны четыре метода реагирования на риски [3]:

– *уклонение от риска*;

– *передача риска*;

– *снижение рисков*;

– *принятие риска*.

Важно помнить о *вторичных* рисках, возникающих в результате применения реагирования на риски, которые тоже должны быть идентифицированы, проанализированы и при необходимости включены в список управляемых рисков.

Управление рисками должно осуществляться на протяжении всего проекта. *Мониторинг и управление рисками* - это процесс идентификации, анализа и планирования реагирования на новые риски, отслеживания ранее идентифицированных рисков, а также проверки и исполнения операций реагирования на риски и оценка эффективности этих операций. Мониторинг и управление рисками включает в себя задачи *пересмотра рисков, аудита рисков, анализа отклонений и трендов*.

В разработке ПО часто приходится формулировать риски в виде *допущений*. Например, оценивая проект разработки и внедрения по схеме с фиксированной ценой, следует записать в допущения предположение о том, что стоимость лицензий на стороннее ПО не изменится, до завершения проекта. *Ограничения*, как правило, сокращают возможности проектной команды в выборе решений. В частности, они могут содержать специфические нормативные требования, специфические технические требования, а также специфические требования к защите информации. Кроме того, уместно сформулировать те требования к системе, которые могут ожидаться заказчиком по умолчанию, но не включаются в рамки данного проекта.

Лекция №3. Модели жизненного цикла программного обеспечения

Цель: получить представление об основных моделях жизненного цикла программного обеспечения и их использования.

На протяжении последних тридцати лет в программировании сменились несколько *моделей жизненного цикла* программного обеспечения [4]. Каждой из моделей соответствует определенная стратегия конструирования ПО [5]. Модели в процессе эволюции усложнялись и совершенствовались.

В 1970 году Уинстон Ройс предложил схему разработки, которая активно использовалась в течение последующих 15 лет (1970-1985). *Каскадная схема* разработки ПО предполагала, что переход на следующую стадию осуществляется только после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии (см. рисунок А.7).

Эту модель в разных источниках часто называют *классическим жизненным циклом* [5] или *водопадной* моделью [4]. Достоинствами такой схемы являются:

- получение плана и временного графика по всем этапам проекта;
- простота планирования процесса разработки;
- получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности.

Именно эту схему используют обычно при блочно-иерархическом подходе к разработке сложных технических объектов, обеспечивая очень высокие параметры эффективности разработки. Однако каскадная схема имеет недостатки:

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;
- цикл основан на точной формулировке исходных требований к ПО;
- результаты проекта доступны заказчику только в конце работы.

Схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования, что уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречается крайне редко. Каскадной (водопадной) схеме соответствует стратегия конструирования называемая *однократным проходом* или *водопадной стратегией*.

Реальный же процесс носит *итерационный* характер. Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в возможности приспособления продукта под операционную систему, в форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях используется *макетирование*, основная цель которого - снять неопределенности в требованиях заказчика. При макетировании модель может принимать одну из трех форм:

- *бумажный макет* или *макет на основе компьютера* (изображается человеко-машинный диалог);
- *работающий макет* (выполняется некоторая часть требуемых функций);
- *существующая программа* (впоследствии характеристики программы должны быть улучшены).

Макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик (рисунок А.8). Макетирование начинается со сбора и уточнения требований к программе. Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить. Затем выполняется быстрое проектирование, причем внимание сосредотачивается на тех характеристиках программного обеспечения, которые должны быть видимы пользователю. Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к программному обеспечению. Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и тем самым не даст возможность разработчику понять, что должно быть сделано.

Несомненным достоинством макетирования является то, что оно обеспечивает определение полных требований к программному продукту.

Однако, когда заказчик видит работающую версию программного продукта, он перестает осознавать, что это всего лишь макет, который далек от законченного программного продукта. При этом в погоне за работающим вариантом остаются нерешенными вопросы качества и удобства сопровождения программного обеспечения. Кроме того, разработчик так же, как и заказчик может принять макет за продукт. Желаемое принимается за действительное.

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время при разработке финишной версии разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Аналогичная схема, поддерживающая итерационный характер процесса разработки, была названа *моделью с промежуточным контролем* (рисунок А.9).

Контроль, выполняемый по данной схеме после завершения каждого этапа, позволяет вернуться на любой уровень и внести изменения. Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Сочетание элементов последовательной водопадной модели с итерационной философией макетирования позволило получить *инкрементную* модель [5], которая соответствует *инкрементной стратегии* конструирования.

Инкрементная модель в отличие от макетирования позволяет получать на каждой итерации работающий продукт. Примерами современной реализации инкрементного подхода могут служить *технология быстрой разработки приложений* (*RAD-технология*) и *экстремальное программирование XP*, предложенное Кентом Беком в 1999 году.

В конце 1980-х годов Германией и США независимо друг от друга была разработана концепция *V-образной модели* - вариации каскадной модели, в которой задачи разработки идут сверху вниз по левой стороне буквы V, а задачи тестирования — вверх по правой стороне буквы V (рисунок А.10). Внутри V проводятся горизонтальные линии, показывающие, как результаты каждой из фаз разработки влияют на развитие системы тестирования на каждой из фаз тестирования.

Основной принцип V-образной модели заключается в том, что детализация проекта возрастает при движении слева направо, одновременно с течением времени, и ни то, ни другое не может повернуть вспять. Итерации в проекте производятся по горизонтали, между левой и правой сторонами буквы.

Немецкая V-модель была разработана аэрокосмической компанией IABG в Оттобрунне для Министерства обороны Германии и летом 1992 была принята немецкой федеральной администрацией для гражданских нужд. Американская V-Model (VEE) была разработана национальным советом по системной инженерии для спутниковых систем, включая оборудование, программное обеспечение и взаимодействие с пользователями. Современной версией V-Model является V-Model XT, которая была утверждена в феврале 2005 года и используется для управления процессом разработки программного обеспечения для немецкой федеральной администрации. Сейчас она является стандартом для немецких правительственных и оборонных проектов, а также для производителей программного обеспечения в Германии. V-Model представляет собой скорее набор стандартов в области проектов, касающихся разработки новых продуктов.

Несомненным достоинством V-модели является то, что пользователи сами участвуют в процессах разработки и поддержки. Комитет по контролю за изменениями поддерживает проект и собирается раз в год для обработки всех полученных запросов на внесение изменений в V-Model. На старте любого проекта V-образная модель может быть адаптирована под этот проект. В модели особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта на ранних стадиях его разработки. В V-образной модели определение требований выполняется перед разработкой проекта системы, а проектирование программного обеспечения - перед разработкой компонентов. Модель определяет продукты, которые должны быть получены в результате процесса разработки, причем каждые полученные данные должны подвергаться тестированию.

Несмотря на указанные достоинства, V-модель не предусматривает работу с параллельными событиями; не позволяет вносить требования динамических изменений на разных этапах жизненного цикла; тестирование требований в жизненном цикле происходит слишком поздно, а некоторый результат можно посмотреть только при достижении низа буквы V.

Для преодоления проблем моделей итеративной разработки в 1988 году Барри Боэмом была предложена *спиральная схема* (рисунок А.11).

Спиральная модель базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент — анализ риска. В соответствии с приведенной выше схемой программное обеспечение создается не сразу, а итерационно с использованием *метода прототипирования*.

Метод базируется на создании *прототипов* - действующих программных продуктов, реализующих отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения. Создание программного обеспечения выполняется в несколько итераций:

а) *I итерация*: специфицируется, проектируется, реализуется и тестируется интерфейс пользователя;

б) *II итерация*: добавляется некоторый ограниченный набор функций;

в) *III ... N- итерации*: расширяется функционал продукта.

На каждом витке спирали проводится анализ риска. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое на этапе проектирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации. Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

Достоинство схемы: начиная с некоторой итерации, на которой обеспечена определенная функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

– сократить время до появления первых версий программного продукта;

– заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;

– ускорить формирование и уточнение спецификаций за счет появления практики использования продукта;

– уменьшить вероятность морального устаревания системы за время разработки.

Основной проблемой использования схемы является определение моментов перехода на следующие стадии, для чего ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

Спиральная модель является классическим примером применения *эволюционной стратегии* конструирования.

Развитием спиральной модели является *компонентно-ориентированная модель*, которая также основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание этапа проектирования: в современных условиях новая разработка должна

основываться на повторном использовании существующих *программных компонентов*.

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

- уменьшает на 30% время разработки программного продукта;
- уменьшает стоимость программной разработки до 70%;
- увеличивает в 1,5 раза производительность разработки.

Альтернативой спиральной и каскадной моделям жизненного цикла является *модель хаоса* (Raccoon, 1995), которая предполагает, что фазы жизненного цикла распространяются на все уровни проекта: от всего проекта в целом до отдельной строки кода. Это означает, что проект, все его системы, модули, функции и строки кода должны быть определены, реализованы и интегрированы. Главное правило - всегда решать наиболее важную задачу первой.

Лекция №4. Качество программного обеспечения

Цель: получить представление о технологичности программного обеспечения, ее критериях, а также модулях и их характеристках.

Для заказчика одной из важнейших характеристик программного обеспечения является качество программного продукта, поскольку именно это определяет насколько оправдаются вложенные в разработку ПО финансовые затраты. С точки же зрения разработчика немаловажно, чтобы разрабатываемый проект быстро и легко кодировался, тестировался, отлаживался и модифицировался.

Таким образом, качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации, называют *технологичностью* [6]. Высокая технологичность проекта особенно важна, если необходимо обеспечить повышенные требования к его качеству или разрабатывается программный продукт, рассчитанный на многолетнее интенсивное использование.

В настоящее время *критериями качества программного обеспечения* (*criteria of software quality*) принято считать [4]:

а) *функциональность* – это способность ПО выполнять набор функций, удовлетворяющих заданным или подразумеваемым потребностям пользователей;

б) *надежность* – это способность программного обеспечения безотказно выполнять определенные функции при заданных условиях в течение заданного периода времени;

в) *легкость применения* – это характеристики программного обеспечения, которые позволяют минимизировать усилия пользователя по подготовке исходных данных, использованию программы и оценке полученных результатов, а также вызывать положительные эмоции у пользователя;

г) *эффективность* – это отношение уровня услуг, предоставляемых программным обеспечением пользователю при заданных условиях, к объему используемых ресурсов;

д) *сопровождаемость* – это характеристика программного обеспечения, которая позволяет минимизировать усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с потребностями пользователей;

е) *мобильность* – это способность программного обеспечения быть перенесенным из одной среды в другую, например, с одного компьютера на другой.

Функциональность и надежность являются обязательными критериями качества ПО. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПО.

Если рассматривать технологичность с позиций устройства общей структуры программного обеспечения, то в первую очередь следует обратить внимание на *проработанность моделей*, *уровень независимости модулей*, *стиль программирования* и *степень повторного использования кодов*.

При проектировании программного обеспечения выполняется разбиение сложной системы на простые части – *декомпозиция*. При этом в зависимости от используемого подхода к разработке применяются *процедурный* (*структурный*) или *объектный* способы декомпозиции. Результатом *процедурной декомпозиции* является *иерархия подпрограмм* (процедур), где функции принятия решения реализуются подпрограммами верхних уровней, а обработка - подпрограммами нижних уровней. Результатом *объектной декомпозиции* является *совокупность объектов*, которые реализуются как переменные некоторых специально разрабатываемых типов.

При любом способе декомпозиции получают набор связанных с соответствующими данными подпрограмм, которые в процессе реализации организуют в *модули* - автономно компилируемые программные единицы. Каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Поэтому программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании [5].

Первоначально, когда размер программ был невелик, и подпрограммы компилировались отдельно, под модулем понималась последовательность

связанных фрагментов программы, обращение к которой выполняется по имени. Со временем, когда размер программ значительно вырос и появилась возможность создавать *библиотеки* ресурсов, под модулем стали понимать автономно компилируемый набор программных ресурсов. Данные модуль получает и/или возвращает через общие области памяти или параметры.

Однако не всякий программный модуль способствует упрощению программы. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля Р. Хольтом были предложены следующие критерии:

- хороший модуль снаружи проще, чем внутри;
- хороший модуль проще использовать, чем построить.

Сложную проблему легче решить, разделив ее на управляемые части. Однако с увеличением количества модулей и уменьшением их размера растут и затраты времени на разработку программного обеспечения. Следовательно, должно существовать оптимальное количество модулей, которое приведет к минимальной стоимости разработки. Критерии Хольта точно отражают свойства, которыми должен обладать оптимальный модуль.

Принцип *информационной закрытости* (рисунок А.12), предложенный Д. Парнасом в 1972 году, утверждает: «*Содержание модулей должно быть скрыто друг от друга*». Модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (*клиентам*).

Информационная закрытость означает следующее:

- все модули независимы, обмениваются только информацией, необходимой для работы;
- доступ к операциям и структурам данных модуля ограничен.

Информационная закрытость позволяет обеспечить возможность разработки модулей независимыми коллективами, а также легкость модификация системы.

Идеальный модуль играет роль «*черного ящика*», содержимое которого невидимо клиентам. Он прост в использовании, поскольку количество «*ручек*» и «*органов управления*» им невелико. Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям.

Г. Майерс [7] предложил для оценки приемлемости программного модуля использовать более конструктивные его характеристики:

- *размер* модуля;
- *прочность* модуля;
- *сцепление* с другими модулями;
- *рутинность* модуля.

Размер модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и

могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений. Кроме того, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов. Это вполне соответствует принципам Хольта.

Чем выше *степень независимости модулей*, тем

- легче разобраться в отдельном модуле и всей программе, тестировать, отлаживать и модифицировать ее**;**

- меньше вероятность появления новых ошибок при исправлении старых или внесении изменений в программу;

- проще организовать разработку программы группой программистов и легче ее сопровождать.

Степень независимости модулей оценивается двумя критериями: *связностью* (*прочностью* по Майерсу) и *сцеплением*.

Связность - мера прочности соединения функциональных и информационных объектов внутри одного модуля, т.е. степень взаимосвязи элементов, реализуемых одним модулем. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи и взаимовлияние модулей. В то же время размещение сильно связанных элементов в разных модулях не только усиливает межмодульные связи, но и усложняет понимание их взаимодействия. Объединение слабо связанных элементов уменьшает технологичность модулей, так как такими элементами сложнее мысленно манипулировать. Различают следующие типы связности:

а) *функциональная* - все части модуля предназначены для выполнения одной функции;

б) *информационная* (*последовательная*) – выходные данные одной функции служат исходными данными для другой функции;

в) *коммуникативная* - функции, обрабатывают одни и те же данные;

г) *процедурная* - части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения;

д) *временная* – части модуля не связаны, но функции выполняются одновременно (параллельно);

е) *логическая* - базируется на объединении данных или функций в одну логическую группу;

ж) *случайная* (*по совпадению*) - связь между элементами мала или отсутствует.

Анализ таблицы А.1 показывает, что целесообразно использовать функциональную, информационную (последовательную) и коммуникативную связности, поскольку именно они показывают наилучшие характеристики.

Сцепление - это мера взаимозависимости модулей, определяющая насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит никакой информации о другом модуле. Различают шесть типов сцепления модулей:

а) *сцепление по данным* - модули обмениваются данными в скалярных значениях. При небольшом количестве передаваемых параметров этот тип обеспечивает наилучшие технологические характеристики ПО;

б) *сцепление по образцу* - модули обмениваются данными, объединенными в структуры. Однако уменьшается «прозрачность» связи между модулями, так как передаваемые данные «спрятаны» в структуры, а при изменении структуры необходимо модифицировать все использующие ее модули. Дает неплохие характеристики, но хуже, чем у предыдущего типа;

в) *сцепление по управлению* - один модуль посылает другому некоторый информационный объект (*флаг* или *переключатель*), предназначенный для управления внутренней логикой модуля. Такие настройки снижают наглядность взаимодействия модулей и обеспечивают худшие характеристики технологичности программ по сравнению с предыдущими типами;

г) *сцепление по внешним ссылкам* - модули ссылаются на один и тот же глобальный элемент данных;

д) *сцепление по общей области данных* - модули работают с общей областью данных. Считается недопустимым, поскольку программы, использующие данный тип сцепления, сложны для понимания при сопровождении ПО. Ошибка одного модуля, приводящая к изменению общих данных, может проявиться при выполнении другого модуля. При ссылке к данным в общей области модули используют конкретные имена, что уменьшает гибкость разрабатываемого ПО;

е) *сцепление по содержимому* - один модуль прямо ссылается на содержание другого модуля (не через его точку входа), что полностью противоречит блочно-иерархическому подходу. Отдельный модуль в этом случае уже не является блоком («черным ящиком»): его содержимое должно учитываться в процессе разработки другого модуля. Такой вид сцепления остается возможным для языков низкого уровня, например, Ассемблера.

Допустимыми считают первые три типа сцепления, использование остальных приводит к резкому ухудшению технологичности программ. Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество принято определять по типу сцепления с худшими характеристиками. Например, при использовании сцепления по данным и сцепления по управлению, определяющим считают сцепление по управлению.

И наконец, последняя характеристика по Майерсу - *рутинность* модуля, т.е. его независимость от предыстории обращений к нему.

Модуль называется *рутинным*, если результат обращения к нему зависит только от значений его параметров и не зависит от предыстории обращений к нему.

Модуль называется *зависящим от предыстории*, если результат обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему.

Майерс не рекомендует использовать зависящие от предыстории (*непредсказуемые*) модули, так как они провоцируют появление в программах

«хитрых» (неуловимых) ошибок. Однако во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому корректнее будет поступать согласно следующим рекомендациям [5]:

- рутинный модуль допускается использовать, если это не приводит к плохим сцеплениям модулей;
- зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
- зависимость в спецификации зависящего от предыстории модуля должна быть четко сформулирована так, чтобы поведение этого модуля было возможно прогнозировать при разных последующих обращениях к нему.

При рассмотрении модуля как библиотеки ресурсов выделяют *библиотеки подпрограмм*, реализующие функции, близкие по назначению, и *библиотеки классов*, реализующие близкие по назначению классы.

В качестве средства улучшения технологических характеристик библиотек ресурсов используют разделение тела модуля на *интерфейсную часть* и *область реализации* (в Паскале –Interface и Implementation, в C++ - h и cpp-файлы).

Интерфейсная часть содержит совокупность объявлений ресурсов (заголовков функций, имен переменных, типов, классов), предоставляемых другим модулям. Необъявленные ресурсы извне не доступны.

Область реализации содержит тела подпрограмм и внутренние ресурсы (подпрограммы, переменные, типы), используемые этими подпрограммами. При такой организации любые изменения реализации библиотеки, не затрагивающие ее интерфейс, не требуют пересмотра модулей, связанных с библиотекой, что улучшает технологичность модулей-библиотек.

Лекция №5. Парадигмы программирования

Цель: получить представление о парадигмах программирования и их особенностях.

В процессе эволюции технологии программирования в рамках традиционных подходов сформировались определенные исходные концептуальные схемы постановки проблем и их решения – *парадигмы* [8], определяющие стиль написания программ, а также образ мышления программиста. Каждая парадигма программирования имеет свой круг приверженцев и класс успешно решаемых задач. Кроме того, в каждой из них приняты разные приоритеты при оценке качества программирования, отличаются инструменты и методы работы

Впервые термин «*парадигма программирования*» применил в своих лекциях Р.Флойд - лауреат премии Тьюринга. По мнению Флойда, парадигмы программирования могут сочетаться, обогащая инструментарий программиста.

Точное определение термину дать проблематично, поскольку в литературных источниках можно встретить отличающиеся друг от друга мнения авторов. Наиболее емкое определение принадлежит Д. Спинеллису: «Понятие «*парадигма*» используется в программировании для определения семейства обозначений (нотаций), разделяющих общий способ (методику) реализации программ».

Общие парадигмы программирования, сложившиеся в самом начале эры компьютерного программирования, - парадигмы *прикладного*, *теоретического* и *функционального* программирования сохраняют свое лидерство до сих пор.

Более пятидесяти лет назад в сфере по организации вычислительных и информационных процессов получила популярность ведущая парадигма *прикладного программирования* на основе *императивного управления и процедурно-операторного стиля* построения программ.

По одной из существующих классификаций языки программирования можно разделить на следующие группы:

- *императивные* (imperative), называемые также *процедурными* (procedural) или *директивными* (directive);
- *декларативные* (declarative);
- *объектно-ориентированные* (object-oriented).

Императивное программирование - это парадигма программирования, которая описывает *процесс* вычисления в виде *инструкций*, изменяющих состояние программы [9]. Императивная программа очень похожа на приказы, то есть это - последовательность команд, которые должен выполнить компьютер. Первыми императивными языками были машинные коды, инструкции которых были крайне просты, что снижало нагрузку на компьютеры, однако затрудняло написание крупных программ. В 1954 появился первый «человеческий» язык программирования, максимально приближенный к естественному, — FORTRAN (Дж. Бэкус, IBM), который является компилируемым и позволяет использовать именованные переменные, составные выражения, подпрограммы и многие другие элементы распространённых сейчас императивных языков. С тех пор семейство императивных языков пополнили:

- ALGOL (конец 1950-х), разработанный с целью упростить выражение математических алгоритмов и в дальнейшем послуживший базой для написания операционных систем для некоторых моделей компьютеров;

- COBOL (1960) и BASIC (1964), появившиеся в результате попыток сделать программирование более похожим на обычный английский язык;

- а также Pascal, (Н.Вирт, начало 1970-х), C (Д. Ритчи), Ada (Honeywell, 1978), C++ (Б.Страуструп, 1985), Perl (Л.Уолл, 1987), Python (Гвидо ван Россум, 1990), PHP (Р.Лердорф, 1994), Java (Sun Microsystems, 1994), C# (1998).

Прикладное программирование подчинено проблемной направленности, отражающей компьютеризацию информационных и вычислительных

процессов численной обработки, исследованных задолго до появления компьютеров. Именно здесь быстро проявился явный практический результат. В практике прикладного программирования принято доверять проверенным шаблонам и библиотекам процедур, избегать рискованных экспериментов. Ценится точность и устойчивость научных расчетов.

Теоретическое (доказательное) программирование - технология разработки программ с доказательствами правильности - доказательствами отсутствия ошибок в программах (несоответствий между программой и реализуемым ею алгоритмом), использовавшаяся в 1980-х годах в академических кругах. Программирование пытается выразить свои формальные модели, показать их значимость и фундаментальность. Эти модели унаследовали основные черты родственных математических понятий и утвердились как алгоритмический подход в информатикеhttp://localhost:3232/department/se/paradigms/popup.lit.html - 38.

Стремление к доказательности построений и оценка их эффективности, правдоподобия, правильности, корректности и других формализуемых отношений на схемах и текстах программ послужили основой *структурного программирования* и других методик достижения надежности процесса разработки программ, например, *литературное* (*грамотное*) *программирование*. Стандартные подмножества Алгола и Паскаля, послужившие рабочим материалом для теории программирования, сменились более удобными для экспериментирования *аппликативными* языками такими, как ML, Miranda, Scheme и другие диалекты Lisp. В настоящее время к ним присоединяются подмножества C и Java.

Структурное программирование (или *программирование без goto*) - методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры блоков. Предложена в 70-х годах XX века Э. Дейкстрой, разработана и дополнена Н. Виртом. Методология структурной разработки программного обеспечения была признана «самой сильной формализацией 1970-х годов». Является одним из способов обеспечения высокого уровня технологичности разрабатываемого программного обеспечения. Различают три вида вычислительного процесса, реализуемого программами: *линейный*, *разветвленный* и *циклический*.

Линейная структура процесса вычислений предполагает, что для получения результата необходимо выполнить операции в определенной последовательности. При *разветвленной структуре* процесса вычислений конкретная последовательность операций зависит от значений одной или нескольких переменных. Для получения результата при *циклической структуре* некоторые действия необходимо выполнить несколько раз.

Декларативное (*логическое*) *программирование* - парадигма программирования, основанная на теории и аппарате математической логики. Возникло как упрощение функционального программирования для математиков и лингвистов, решающих задачи символьной обработки.

Программа «декларативна», если она описывает, *что* представляет собой нечто, а не *как* его создать. Например, веб-страницы на HTML декларативны, так как они описывают, *что* именно должна содержать страница, а не *как* ее отображать на экране. Декларативные программы пишутся на исключительно функциональном, логическом языках или языке программирования с ограничениями. Самым известным языком логического программирования является Prolog. Примеры использования подобных технологий можно увидеть в Microsoft ASP.NET и Microsoft Windows Communication Foundation.

Функциональное программирование - в основу функциональной парадигмы и функционального стиля программирования положен простой принцип: функция — вполне подходящее и адекватное средство описания вычислений. Функциональная программа представляет собой набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают параметры, вычисляют и возвращают результат, при необходимости вычисляя значения других функций. На функциональном языке программист не должен описывать порядок вычислений. Нужно просто описать желаемый результат как систему функций.

Функциональное программирование нашло большое применение в теории искусственного интеллекта и её приложениях.

Продуманность и методическая обоснованность первых реализаций основного языка функционального программирования Lisp (List processor) позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач и виды технических средств, в том числе язык-прототип ISWIM, Scheme, ML, Caml, Miranda, Haskell, Clean – диалекты Lisp.

Развитием функциональной парадигмы стало *аппликативное программирование*. Аппликативный подход к написанию программы состоит в систематическом осуществлении применения одного объекта к другому. Результатом такого применения вновь является объект, который может участвовать в применениях как в роли функции, так и в роли аргумента и т. д. Это делает запись программы математически ясной. Конструкции аппликативных языков программирования, в целом отличает простота исходных посылок. Их базовыми строительными блоками являются представления о выражении и функции, а все прочие понятия являются производными и вводятся шаг за шагом, что придаёт конструктивный стиль самому процессу программирования.

Основные средства и методы программирования сформировались по мере возрастания сложности решаемых задач. Произошло расслоение парадигм программирования в зависимости от глубины и общности проработки технических деталей организации процессов компьютерной

обработки информации. Выделились разные стили программирования, наиболее зрелые из которых:

- *низкоуровневое* (*машинно-ориентированное*) *программирование*;
- *системное программирование*;
- *декларативно-логическое программирование*;
- *оптимизационно-трансформационное программирование*;
- *высокопроизводительное* (*параллельное*) *программирование*.

Низкоуровневое программирование характеризуется аппаратным подходом к организации работы компьютера, нацеленным на доступ к любым возможностям оборудования. В центре внимания - конфигурация оборудования, состояние памяти, команды, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность реагирования. Ассемблер в качестве предпочтительного средства программирования на некоторое время уступил языкам Паскаль и Си даже в области микропрограммирования, но усовершенствование пользовательского интерфейса может восстановить его позиции.

Системное (*стандартное*) *программирование* долгое время развивалось под прессом сервисных и заказных работ. Свойственный таким работам производственный подход опирается на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. В этой области доминирует *императивно-процедурный* стиль прикладного программирования. Он допускает использование модульного программирования, но сопровождается сложными построениями, спецификациями, методами тестирования и средствами интеграции программ.

При построении трансляторов используется *автоматное программирование* - это парадигма программирования, при использовании которой программа или её фрагмент осмысливается как модель какого-либо формального автомата. В зависимости от конкретной задачи в автоматном программировании могут использоваться как конечные автоматы, так и автоматы более сложной структуры. Автоматное программирование широко применяется при построении лексических анализаторов (классические конечные автоматы) и синтаксических анализаторов (автоматы с магазинной памятью). К языкам автоматного программирования относятся:

а) *язык последовательных функциональных схем SFC* (Sequential Function Chart) - графический язык программирования широко используется для программирования промышленных логических контроллеров PLC;

б) *дракон-схемы* - графический язык программирования, используется для программирования в ракетно-космической технике;

с) *Рефлекс* - Си-подобный язык программирования, ориентированный на описание сложных алгоритмов управления в задачах промышленной автоматизации.

Оптимизационно-трансформационное программирование объединило технику оптимизации программ, макрогенерации и частичных вычислений. Парадигма базируется на *эквивалентности* информации, которая проявляется

в определении преобразований программ и процессов, в поиске критериев применимости преобразований, в выборе стратегии их использования. В качестве методов повышения эффективности информационной обработки использует смешанные вычисления, отложенные действия, «ленивое» программирование, задержанные процессы и т.п.

Высокопроизводительное (*параллельное*) *программирование* является ветвью системного программирования и нацелено на достижение предельно возможных характеристик при решении особо важных задач. Естественный резерв производительности компьютеров - параллельные процессы. Их организация требует детального учета временных характеристик и *неимперативного* стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, потребовали особой техники. *Графово-сетевой* подход к представлению систем и процессов для параллельных архитектур получил выражение в специализированных языках параллельного программирования и суперкомпиляторах.

В том или ином виде разными языками программирования поддерживается *обобщённое программирование* - парадигма программирования, заключающаяся в таком описании данных и алгоритмов, которое можно применять к различным типам данных, не меняя само это описание. Возможности обобщённого программирования впервые появились в 1970-х годах в языках Клу и Ада, а затем во многих объектно-ориентированных языках таких, как C++, Java, Object Pascal, D, Eiffel, языках для платформы .NET и других.

В результате эволюции *объектно-ориентированной парадигмы программирования* появились:

- *аспектно-ориентированное программирование* (АОП) - парадигма программирования, основанная на идее разделения функциональности для улучшения разбиения программы на модули. Методология была предложена группой инженеров исследовательского центра Xerox PARC под руководством Грегора Кичалеса (Gregor Kiczales). В 2001 г. ими же было разработано аспектно-ориентированное расширение для языка Java, получившее название AspectJ. АОП дополняет объектно-ориентированное программирование, обогащая его другим типом модульности – аспектами. Аспекты в системе могут изменяться, вставляться, удаляться на этапе компиляции и, более того, повторно использоваться;

- *событийно-ориентированное программирование* (СОП) - парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета). Как правило, в реальных задачах оказывается недопустимым длительное выполнение обработчика события, поскольку при этом программа не может реагировать на другие события. В связи с этим при написании событийно-ориентированных программ часто применяют автоматное программирование;

- *компонентно-ориентированное программирование* - парадигма программирования, ключевой фигурой которой является *компонент* - независимый модуль программного кода, предназначенный для повторного использования и развертывания. Компонентно-ориентированное программирование включает в себя набор ограничений, налагаемых на механизм объектно-ориентированного программирования. Среди языков компонентно-ориентированного программирования можно выделить Оберон (1987, Н.Вирт), компонентный Паскаль, PHP.

- *прототипное программирование* - стиль объектно-ориентированного программирования, при котором отсутствует понятие класса, а повторное использование (наследование) производится путём клонирования существующего экземпляра объекта (прототипа). Примером прототип-ориентированного языка является язык Self. В дальнейшем этот стиль программирования начал обретать популярность и был положен в основу таких языков программирования, как JavaScript, Lua, Io, REBOL.

Лекция №6. Программирование без ошибок

Цель: получить представление о возможных ошибках программирования и способах защиты от них.

Любая из ошибок программирования, которая не обнаруживается на этапах компиляции и компоновки программы, в итоге может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.

Однако до того, как результат работы программы становится фатальным, ошибки много раз проявляются в виде неверных типах данных, неверных промежуточных результатах, неверных управляющих переменных, индексах структур данных и т. п. (рисунок А.13).

Часть ошибок можно обнаружить и нейтрализовать, пока они не привели к тяжелым последствиям. Программирование, при котором применяют специальные приемы раннего обнаружения и нейтрализации ошибок, названо *защитным* или *программированием «с защитой от ошибок»*.

Детальный анализ ошибок и их возможных ранних проявлений показывает, что целесообразно проверять правильность выполнения операций ввода-вывода, а также допустимость промежуточных результатов (значений управляющих переменных, индексов, типов данных, числовых аргументов).

Причинами неверного определения исходных данных могут являться как *внутренние* ошибки (ошибки устройств ввода-вывода или программного обеспечения), так и *внешние* ошибки (ошибки пользователя). Различают:

а) *ошибки передачи* - аппаратные средства искажают данные;

б) *ошибки преобразования* - программа неверно преобразует исходные данные из входного формата во внутренний;

в) *ошибки перезаписи* - пользователь ошибается при вводе данных;

г) *ошибки данных* — пользователь вводит неверные данные.

Ошибки передачи обычно контролируются аппаратно.

Для защиты от ошибок преобразования данные после ввода сразу демонстрируют пользователю («эхо»), выполняя преобразование сначала во внутренний формат и обратно. Предотвратить все ошибки преобразования на этом этапе крайне сложно, поэтому фрагменты программы тестируют [10], используя методы эквивалентного разбиения и граничных значений.

Обнаружить и устранить ошибки перезаписи можно только при вводе избыточных данных, например, контрольных сумм. Если ввод избыточных данных нежелателен, то, по возможности, проверяют входные данные и контролируют интервалы возможных значений, определенных в техническом задании, а также выводят введенные данные для проверки пользователю.

Неверные данные обычно может обнаружить только пользователь.

Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых ошибок кодирования и проектирования. Для этого необходимо, чтобы в программе использовались переменные с ограничениями любого происхождения, например, связанные с сущностью моделируемых процессов.

Любые дополнительные операции в программе требуют использования дополнительных ресурсов (времени, памяти) и могут содержать ошибки. Имеет смысл проверять только те промежуточные результаты, проверка которых целесообразна и не сложна, например, допустимость индекса.

Для снижения погрешности результатов вычислений следует:

а) избегать вычитания близких чисел (машинный ноль);

б) избегать деления больших чисел на малые;

в) сложение длинной последовательности чисел начинать с меньших по абсолютной величине;

г) стремиться по возможности уменьшать количество операций;

д) использовать методы с известными оценками погрешностей;

е) не использовать условие равенства вещественных чисел;

ж) вычислять с двойной точностью, а результат выдавать - с одинарной.

Поскольку полный контроль данных на входе и в процессе вычислений невозможен, следует предусматривать перехват и обработку аварийных ситуаций. Для перехвата и обработки аппаратно и программно фиксируемых ошибок в некоторых языках программирования (Delphi Pascal, C++ и Java) предусмотрены средства обработки исключений. Использование этих средств позволяет не допустить выдачи пользователю сообщения об аварийном завершении программы, а программист получает возможность предусмотреть действия, позволяющие исправить эту ошибку или выдать пользователю сообщение с точным описанием ситуации и продолжить работу.

С целью устранения ошибок программирования необходимо организовать контроль структуры программы. Г.Майерс предлагает использовать следующие методы [7]:

а) *статический контроль* состоит в оценке структуры программы: насколько хорошо программа разбита на модули с учетом значений рассмотренных выше основных характеристик модуля;

б) *смежный контроль* – это контроль со стороны разработчиков архитектуры, внешнего описания программной системы или спецификации модулей со стороны разработчиков этих модулей;

в) *сквозной контроль* – это проверка структуры программы при выполнении заранее разработанных тестов.

Контроль структуры программы производится в рамках классического подхода. При конструктивном и архитектурном подходах контроль структуры программы осуществляется в процессе кодирования модулей в подходящие моменты времени.

Лекция №7. Основные подходы к разработке программного обеспечения. Требования и исходные данные к разработке

Цель: получить представление о подходах к проектированию программного обеспечения, а также основных требованиях и этапах разработки.

При проектировании, реализации и тестировании компонентов структуры программного обеспечения используются следующие основные подходы: *восходящий*, *нисходящий* и *расширение ядра* [11].

При использовании *восходящего подхода* сначала проектируются и реализуются компоненты нижнего уровня, затем предыдущего и т.д. По мере завершения тестирования и отладки компонентов осуществляется их сборка, причем компоненты нижнего уровня помещают в библиотеки компонентов. Для тестирования и отладки разрабатывают специальные тестирующие программы. При промышленном изготовлении программного обеспечения восходящий подход практически не используют, поскольку он имеет существенные недостатки:

- увеличивается вероятность несогласованности компонентов;
- появляются дополнительные расходы на проектирование и реализацию тестирующих программ, которые впоследствии будет невозможно использовать;
- интерфейс проектируется в последнюю очередь, что долго не позволяет продемонстрировать его заказчику.

Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз»: вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При программировании компоненты нижних не реализованных уровней заменяют специально разработанными отладочными модулями - «*заглушками*», что позволяет тестировать и отлаживать уже

реализованную часть. При использовании нисходящего подхода применяют *иерархический*, *операционный* и *комбинированный* методы определения последовательности проектирования и реализации компонентов, характерные особенности которых приведены в таблице А.2.

Расширение ядра – это подход, который предполагает в первую очередь проектирование и реализацию некоторой основы – *ядра программного обеспечения* (например, структуры данных и связанные с ними функции), а затем – его наращивание с использованием методов нисходящего и восходящего подходов или их комбинации.

Методы восходящей и нисходящей разработок относятся к *классическим.* Их особенностью является требование, чтобы модульная структура программы была разработана до начала кодирования модулей, что полностью соответствует водопадному подходу к разработке программного обеспечения. Однако эти методы вызывают ряд возражений [4]: представляется сомнительным, чтобы до программирования модулей можно было разработать структуру программы достаточно точно и содержательно. С целью устранения этого недостатка используются *конструктивный* и *архитектурный* подходы к разработке программ, в которых модульная структура формируется в процессе кодирования модулей.

Конструктивный подход представляет собой модификацию нисходящей разработки, при которой модульная структура программы формируется в процессе программирования модулей. Разработка программы при конструктивном подходе начинается с программирования головного модуля, исходя из спецификации программы в целом.

Архитектурный подход представляет собой модификацию восходящей разработки, при которой модульная структура программы также формируется в процессе программирования модуля. Но при этом цель разработки существенно изменяется: повышается уровень используемого языка программирования, а не разрабатывается конкретная программа. Для заданной предметной области выделяются типичные функции, каждая из которых может использоваться при решении разных задач в этой области. Обычно сначала выделяются и реализуются отдельными модулями более простые функции, а постепенно затем появляются модули, использующие ранее выделенные функции.

Одним из важнейших и требующих особого внимания считается этап *постановки задачи*, в процессе которого формулируется назначение и определяются требования к программному обеспечению. Каждое требование представляет собой описание необходимого или желаемого свойства программного обеспечения. Различают *функциональные требования*, определяющие функции, которые должно выполнять разрабатываемое программное обеспечение, и *эксплуатационные требования*, определяющие особенности его функционирования. Требования к программному обеспечению, имеющему *прототипы*, обычно определяют по аналогии,

учитывая структуру и характеристики уже существующего программного обеспечения.

Для формулирования требований к программному обеспечению, не имеющему аналогов, иногда необходимо провести специальные исследования, называемые *предпроектными*. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы ее решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого программного обеспечения. Для их выполнения, как правило, используют все доступные источники информации, в том числе результаты научно-исследовательских работ, привлекают консультантов-специалистов. В любом случае этап постановки задачи заканчивается разработкой *технического задания*, фиксирующего принципиальные требования, и принятием основных проектных решений.

Сложность многих программных систем не позволяет сразу сформулировать четкие требования к ним. Обычно для перехода от идеи создания программного обеспечения к четкой формулировке требований, которые могут быть занесены в техническое задание, необходимо выполнить *предпроектные* исследования в области разработки. Их целью является преобразование общих нечетких знаний о предназначении будущего программного обеспечения в сравнительно точные требования к нему.

Существуют два варианта неопределенности:

а) неизвестны методы решения формулируемой задачи - такого типа неопределенности обычно возникают при решении научно-технических задач, поэтому во время предпроектных исследований определяют возможность решения поставленной задачи и методы, позволяющие получить требуемый результат, что может потребовать соответствующих научных исследований как фундаментального, так и прикладного характера, разработки и исследования новых моделей объектов реального мира;

б) неизвестна структура автоматизируемых информационных процессов - обычно встречается при построении автоматизированных систем управления предприятиями, поэтому в этом случае определяют:

– структуру и взаимосвязи информационных процессов;

– распределение функций между человеком и системой, аппаратурой и программным обеспечением;

– функции программного обеспечения, условия его функционирования, особенности аппаратных и пользовательских интерфейсов;

– требования к программным и информационным компонентам, необходимые аппаратные ресурсы, требования к базам данных и физические характеристики программных компонент.

На практике для проведения предпроектных исследований привлекают консультантов - специалистов в автоматизируемой области, используют результаты научных исследований, научного прогнозирования, а также новейших разработок в области техники и технологий. Результаты

предпроектных исследований предметной области используются в процессе разработки *технического задания*.

Техническое задание представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и регламентирован процесс приемно-сдаточных испытаний. В разработке технического задания участвуют как представители заказчика, так и представители исполнителя. В основе этого документа лежат исходные требования заказчика, результаты выполнения предпроектных исследований, научного прогнозирования и т. п. Основными факторами, определяющими характеристики разрабатываемого программного обеспечения, являются:

а) исходные данные и требуемые результаты, определяющие функции программы или системы;

б) среда функционирования (программная и аппаратная) - задается или выбирается для обеспечения параметров, указанных в техническом задании;

в) возможное взаимодействие с другим программным обеспечением и/или специальными техническими средствами - задается или выбирается, исходя из набора выполняемых функций.

Разработка технического задания - процесс трудоемкий, требующий определенных навыков, выполняется в следующей последовательности:

а) устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных;

б) определяют перечень результатов, их характеристики и способы представления;

в) уточняют среду функционирования программного обеспечения: конкретную комплектацию и параметры технических средств, версию используемой операционной системы, версии и параметры другого установленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту;

г) если разрабатываемое программное обеспечение собирает и хранит некоторую информацию или включается в управление каким-либо техническим процессом, то регламентируют действия программы в случае сбоев оборудования и энергоснабжения.

Образец выполнения технического задания на создание системы «Учет успеваемости студентов» и пример оформления титульного листа приведены в Приложении Б.

Система предназначена для оперативного учета успеваемости студентов в сессию деканом, заместителями декана по курсам и сотрудниками деканата. Сведения об успеваемости студентов должны храниться в течение всего срока обучения и использоваться при составлении справок о прослушанных курсах и приложений к диплому. Данный пример составлен по сокращенной схеме, несущественные для данной разработки разделы пропущены.

Лекция №8. Принятие принципиальных решений

Цель: ознакомиться с особенностями принятия принципиальных решений начальных этапов проектирования.

После утверждения технического задания разработчик непосредственно приступает к созданию программного обеспечения. Однако переход к этапу *уточнения спецификаций* требует принятия некоторых принципиальных решений, от которых во многом зависят характеристики, возможности разрабатываемого программного обеспечения, особенности разработки. Решения, которые определяют, что проектируется, с какими потребительскими характеристиками, как и какими средствами. К таким решениям относят:

- выбор архитектуры программного обеспечения;
- типа пользовательского интерфейса;
- технологии работы с документами;
- выбор подхода к разработке;
- выбор языка и среды программирования.

Часть решений может быть определена в техническом задании, образовав группу *технологических требований,* остальные должны быть приняты как можно раньше, так как представляют собой исходные данные для процесса проектирования. Рассмотрим их более подробно.

Архитектура программного обеспечения – это совокупность базовых принципов его построения. Она определяется сложностью решаемых задач, степенью универсальности разрабатываемого программного обеспечения и числом пользователей, одновременно работающих с одной его копией.

Различают *однопользовательскую* архитектуру, при которой программное обеспечение рассчитано на одного пользователя, и *многопользовательскую* архитектуру, которая рассчитана на работу в локальной или глобальной сети.

В рамках однопользовательской архитектуры различают *программы, пакеты программ*, *программные комплексы*, *программные системы*.

Многопользовательскую архитектуру реализуют системы, построенные по принципу «клиент-сервер». *Многопользовательские программные системы* должны организовывать сетевое взаимодействие отдельных компонентов ПО, что усложняет процесс его разработки, поэтому для разработки используют специальные технологии или платформы (технологии CORBA, COM, Java).

Тип пользовательского интерфейса во многом определяет сложность и трудоемкость разработки. По последним данным до 80 % программного кода может реализовывать именно пользовательский интерфейс [12]. Различают четыре типа пользовательских интерфейсов:

а) *примитивные* - реализуют единственный сценарий работы, используя операции ввода данных, их обработки и вывода результатов;

б) *меню* - реализуют множество сценариев работы, операции которых организованы в иерархические структуры;

в) *со свободной навигацией* - реализуют множество сценариев, операции которых не привязаны к уровням иерархии, и предполагают определение множества возможных операций на конкретном шаге работы; интерфейсы данной формы в основном используют Windows-приложения;

г) *прямого манипулирования* - реализуют множество сценариев, представленных в операциях над объектами, инициируемых перемещением пиктограмм объектов мышью; данная форма реализована в интерфейсе операционной системы Windows.

Появление объектно-ориентированных визуальных сред разработки программного обеспечения, использующих событийный подход к программированию, существенно снизило трудоемкость разработки интерфейсов и упростило реализацию интерфейсов прямого манипулирования.

Выбор типа интерфейса включает выбор *технологии работы с документами.* Различают две основные технологии:

а) *однодокументная*, которая предполагает однодокументный интерфейс (*SDI - Single Document Interface*) и используется, если одновременная работа с несколькими документами не обязательна;

б) *многодокументная*, которая предполагает многодокументный интерфейс (*MDI - Multiple Document Interface*) и используется, если программное обеспечение должно работать с несколькими документами одновременно, например, с несколькими текстами или несколькими изображениями.

Трудоемкость реализации многодокументных интерфейсов с использованием современных библиотек примерно на 3...5 % выше, чем однодокументных, однако тип влияет на трудоемкость более существенно.

При выборе интерфейса прямого манипулирования или со свободной навигацией практически однозначно предполагается использование объектного подхода и событийного программирования, так как современные среды визуального программирования такие, как Visual C++, Delphi, Builder C++, предоставляют интерфейсные компоненты именно в виде объектов библиотечных классов. При этом в зависимости от сложности предметной области программное обеспечение может реализовываться как с использованием объектов и, соответственно, классов, так и чисто процедурно. Исключение составляют случаи использования специализированных языков разработки Интернет-приложений (Perl), построенных по совершенно другому принципу.

Примитивный интерфейс и интерфейс типа меню совместимы как со структурным, так и с объектным подходами к разработке. Поэтому выбор подхода осуществляют с использованием дополнительной информации.

Объектный подход эффективен для разработки очень больших программных систем (более 100000 операторов) и в случаях, когда объектная структура предметной области ярко выражена. Следует осторожно использовать объектный подход при жестких ограничениях на эффективность

программного обеспечения, например, при разработке систем реального времени. Во всех прочих случаях выбор подхода остается за разработчиком.

В большинстве случаев проблемы выбора языка программирования реально не существует. Язык может быть определен:

- организацией, ведущей разработку;
- программистом, который, по возможности, всегда будет использовать хорошо знакомый язык;
- устоявшимся мнением.

В общем случае все существующие языки программирования можно разделить на четыре группы [13]:

а) универсальные языки высокого уровня;
б) специализированные языки разработчика программного обеспечения;
в) специализированные языки пользователя;
г) языки низкого уровня.

В группе *универсальных языков высокого уровня* безусловными лидерами сегодня являются *C*-подобные языки, возможности которых позволяют их использовать для создания операционных систем. Альтернативой им среди универсальных языков программирования, используемых для создания прикладного программного обеспечения, на сегодня является *Pascal*, компиляторы которого в силу четкого синтаксиса обнаруживают, помимо синтаксических, и большое количество семантических ошибок. Версия *Object Pascal*, использованная в среде *Delphi*, сопровождается профессиональными библиотеками классов, что делает *Delphi* достаточно эффективной средой для создания приложений *Windows*. Кроме этих языков, к группе универсальных принадлежат также *Basic, Modula, Ada* и некоторые другие. Каждый из указанных языков так же, как *C++* и *Pascal*, имеет свои особенности и, соответственно, свою область применения.

Специализированные языки разработчика используют для создания конкретных типов программного обеспечения. К ним относят языки баз данных, языки создания сетевых приложений, языки создания систем искусственного интеллекта и т. д.

Специализированные языки пользователя обычно являются частью профессиональных сред, характеризуются узкой направленностью и разработчиками программного обеспечения не используются.

Языки низкого уровня позволяют осуществлять программирование практически на уровне машинных команд. При этом получают самые оптимальные как с точки зрения времени выполнения, так и с точки зрения объема необходимой памяти программы. Но эти языки совершенно не годятся для создания больших программ и, тем более, программных систем. Основная причина - низкий уровень абстракций, которыми должен оперировать разработчик, откуда недопустимо большое время разработки. Существенно и то, что сами языки низкого уровня не поддерживают принципов структурного программирования, что значительно ухудшает технологичность разрабатываемых программ. В настоящее время языки типа *Ассемблера*

обычно используют при написании сравнительно простых программ, взаимодействующих непосредственно с техническими средствами (например, драйверов), а также в виде вставок в программы на языках высокого уровня.

Таким образом, при выборе языка программирования следует руководствоваться следующими соображениями:

а) язык должен быть удобен для программиста;

б) язык должен быть пригоден для данного компьютера;

в) язык должен быть пригоден для решения данной задачи.

Среда программирования - программный комплекс, включающий текстовый редактор, встроенные компилятор, компоновщик, отладчик, справочную систему и другие программы, использование которых упрощает процесс написания и отладки программ.

Последнее время широкое распространение получили среды визуального программирования, позволяющие визуально подключать к программе коды из специальных библиотек компонентов. Наиболее часто используют визуальные среды Delphi, C++ Builder фирмы Borland (Inprise Corporation), Visual C++**,** Visual Basic фирмы Microsoft, Visual Ada фирмы IBM. Причем, среды фирмы Microsoft обеспечивают более низкий уровень программирования «под Windows». Это является их достоинством (уменьшается вероятность возникновения «нестандартной» ситуации) и недостатком (существенно загружает программиста «рутинной» работой, от которой избавлен программист, работающий с Delphi или C++ Builder). В общем случае, выбор среды в значительной степени должен определяться характером разработки.

Описанные принципиальные решения существенно влияют на трудоемкость и сложность разработки. Только после их принятия переходят к анализу требований и разработке спецификаций проектируемого программного обеспечения.

Лекция №9. Анализ требований и определение спецификаций

Цель: ознакомиться с особенностями структурного подхода к определению спецификаций, структур данных, моделей и методов решения задач.

Разработка любого программного обеспечения начинается с анализа требований к будущему программному продукту. В результате анализа получают *спецификации* разрабатываемого программного обеспечения.

Спецификации – это полное и точное описание функций и ограничений разрабатываемого программного обеспечения. Различают:

- *функциональные* спецификации, которые описывают функции разрабатываемого программного продукта и состав обрабатываемых данных;

- *эксплуатационные*, которые определяют требования к техническим средствам, надежности, информационной безопасности и т. д.

Естественный язык для описания спецификаций не подходит, поскольку не обеспечивает необходимой точности. Точные спецификации можно определить, лишь разработав некоторую *формальную модель*. Формальные модели можно разделить на две группы: модели, зависящие от подхода к разработке, и модели, не зависящие от него. При любом подходе к разработке используются *диаграммы переходов состояний* и *математические модели предметной области*. Поскольку разные модели описывают проектируемое программное обеспечение с разных сторон, используют сразу несколько моделей, которые сопровождают текстами. В состав *обобщенной* модели, как правило, входят следующие элементы: спецификации процессов; словарь терминов; диаграммы переходов состояний; функциональные диаграммы; диаграммы потоков данных.

Спецификации процессов обычно представляют в виде краткого текстового описания, схем алгоритмов, псевдокодов, Flow-форм или диаграмм Насси-Шнейдермана. Для краткости и понятности описания не только разработчику, но и заказчику, чаще всего используют псевдокоды.

Словарь терминов – краткое описание основных понятий, используемых при составлении спецификаций, который включает определение основных понятий предметной области, описание структур элементов данных, их типов и форматов, а также всех сокращений и условных обозначений. Словарь предназначен для однозначного понимания предметной области и исключения риска возникновения разногласий при обсуждении моделей между заказчиками и разработчиками. Описание термина в словаре выполняется по схеме: *«термин – категория– краткое описание»*.

Диаграмма переходов состояний демонстрирует поведение разрабатываемой программной системы при получении управляющих воздействий (управляющей информации, поступающей извне). Для ее построения в соответствии с *теорией конечных автоматов* необходимо определить: основные состояния, управляющие воздействия (или условия перехода), выполняемые действия и возможные варианты переходов из одного состояния в другое.

Если программная система в процессе функционирования активно не взаимодействует с окружающей средой (пользователем или датчиками), например, использует примитивный интерфейс и выполняет некоторые вычисления по заданным исходным данным, то диаграмма переходов состояний обычно интереса не представляет (рисунок А.14). Представленная диаграмма демонстрирует только последовательно выполняемые переходы: из исходного состояния - в состояние ввода данных, после выполнения вычислений - в состояние вывода и, наконец, в состояние завершения работы.

Для интерактивного программного обеспечения с развитым пользовательским интерфейсом характерно получение команд различных типов (рисунок А.15), а для программного обеспечения реального времени - однотипных сигналов (либо от многих датчиков, либо требующих продолжительной обработки). Общим для них является наличие состояния

ожидания, когда программное обеспечение приостанавливает работу до получения очередного управляющего воздействия.

В отличие от интерактивных систем для систем реального времени обычно установлено более жесткое ограничение на время обработки полученного сигнала программного обеспечения. Такое ограничение часто требует выполнения дополнительных исследований поведения системы во времени. В любом случае полученную диаграмму переходов состояний обязательно согласовывают с заказчиком.

Функциональными называют диаграммы, отражающие взаимосвязи функций разрабатываемого программного обеспечения. Для отображения взаимосвязи функций строится *иерархии функциональных диаграмм,* схематически представляющих взаимосвязи нескольких функций. Каждый блок такой диаграммы соответствует некоторой функции (рисунок А.16), для которой должны быть определены: исходные данные, результаты, управляющая информация и механизмы ее осуществления - человек или технические средства. Все связи функции представляются дугами. Дуги, изображающие каждый тип связей, должны подходить к блоку с определенной стороны, а направление связи должно указываться стрелкой в конце дуги. Физически дуги исходных данных, результатов и управления представляют собой наборы данных, передаваемые между функциями. Дуги, определяющие механизм выполнения функции, используются при описании сложных информационных систем. Блоки и дуги сопровождаются текстами на естественном языке. Блоки на диаграмме размещают по «ступенчатой» схеме в соответствии с последовательностью их работы или *доминированием* - влиянием, которое оказывает один блок на другие. Дуги могут разветвляться и соединяться различными способами. Разветвление означает, что часть или вся информация может использоваться в каждом ответвлении дуги.

В процессе построения иерархии диаграмм фиксируют всю уточняющую информацию и строят словарь данных, в котором определяют структуры и элементы данных, показанных на диаграммах. В результате получают спецификацию, которая состоит из иерархии функциональных диаграмм, спецификаций функций нижнего уровня и словаря, имеющих ссылки друг на друга. Функциональную модель имеет смысл применять для определения спецификаций программного обеспечения, которое не предусматривает работу со сложными структурами данных.

Диаграммы потоков данных позволяют специфицировать функции разрабатываемого программного обеспечения и обрабатываемые им данные. При использовании этой модели систему представляют в виде иерархии диаграмм потоков данных, описывающих асинхронный процесс преобразования информации с момента ввода в систему до выдачи пользователю. На каждом следующем уровне иерархии происходит уточнение процессов, пока очередной процесс не будет признан элементарным. В основе модели лежат понятия:

а) *внешняя сущность* - объект или физической лицо, которое является источником или приемником информации (банк, персонал, клиент и т.д.);

б) *процесс* – преобразование входных данных в выходные в соответствии с алгоритмом;

в) *хранилище* (накопителя) *данных* – абстрактное устройство для хранения информации (картотека, база данных, файл и т.д.);

г) *поток данных* – процесс передачи информации от источника к приемнику (автоматически ил вручную).

Для отображения связей, существующих между отдельными элементами данных, используют совокупности правил и ограничений - *структуры данных* (рисунок А.17). Различают:

- *абстрактные структуры* данных, используемые для уточнения связей между элементами;

- *конкретные структуры,* используемые для представления данных в программах.

Для задач, алгоритм решения которых не очевиден, используют разного рода *математические модели*. Процесс построения такой модели включает:

а) анализ условия задачи;

б) выбор математических абстракций, *адекватно* (с требуемой *точностью* и *полнотой*) представляющих исходные данные и результаты;

в) формальную постановку задачи;

г) определение метода преобразования исходных данных в результат (*метода решения задачи*).

Для многих задач, которые часто встречаются на практике, в математике определены как модели, так и методы решения. В ряде случаев формальная постановка задачи однозначно определяет метод ее решения, но, как правило, методов решения существует несколько, и тогда для выбора метода решения может потребоваться специальное исследование. При выборе метода учитывают:

- особенности данных конкретной задачи, связанные с предметной областью (погрешность, возможные особые случаи и т. п.);

- требования к результатам (допустимую погрешность);

- характеристики метода (точный или приближенный, погрешности результатов, вычислительную сложность, сложность реализации).

Определив методы решения, целесообразно вручную подсчитать ожидаемые результаты для некоторых вариантов исходных данных. Эти данные позже могут быть использованы при тестировании программного обеспечения. Выполнение операций вручную позволяет точно уяснить последовательность действий, что упростит разработку алгоритмов. Имеет смысл продумать, для каких сочетаний исходных данных результат не существует или не может быть получен данным методом, что тоже необходимо учесть при разработке программного обеспечения.

Лекция №10. Особенности проектирования программного обеспечения при структурном подходе

Цель: ознакомиться с особенностями проектирования программного обеспечения при структурном подходе, а также с приемами разработки структурной и функциональной схем.

Для того чтобы иметь четкое представление о компонентах сложного программного обеспечения и их взаимосвязях, необходимо уточнить его структуру. Результат уточнения может быть представлен в виде *структурной* и/или *функциональной* схем и описания (спецификаций) компонентов. Схему, отражающую состав и взаимодействие по управлению частей разрабатываемого программного обеспечения, называют *структурной*.

Поскольку организация программ в пакеты не предусматривает передачи управления между ними, структурные схемы пакетов программ не информативны. Поэтому структурные схемы разрабатывают для каждой программы пакета, а список программ пакета определяют, анализируя функции, указанные в техническом задании.

Разработку структурной схемы программы, включающей в качестве структурных компонентов только подпрограммы и библиотеки ресурсов, выполняют методом пошаговой детализации.

Структурными компонентами программной системы/комплекса служат программы, библиотеки ресурсов, подсистемы, базы данных. Структурная схема программного комплекса демонстрирует передачу управления от программы-диспетчера соответствующей программе (рисунок А.18а).

Структурная схема программной системы показывает наличие подсистем или других структурных компонентов. В отличие от программного комплекса отдельные части (подсистемы) программной системы интенсивно обмениваются данными между собой и с основной программой. Структурная схема программной системы этого не показывает (рисунок А.19а).

С точки зрения взаимодействия его компонентов между собой и с внешней более полное представление о проектируемом программном обеспечении средой дает *функциональная* схема. *Функциональная схема* (*схема данных*, ГОСТ 19.701-90) - схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств.

Функциональные схемы более информативны, чем структурные. На рисунках А.18б и А.19б приведены функциональные схемы программных комплексов и систем. Все компоненты структурных и функциональных схем должны быть описаны. Следует тщательно прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок, к которым относятся ошибки, обнаруживаемые при комплексном тестировании.

Для изображения функциональных схем используют специальные обозначения, установленные стандартом. Основные обозначения схем данных приведены в таблице А.3.

Таким образом, структурная схема программы – это результат декомпозиции программы методом пошаговой детализации, т.е. многоуровневая иерархическая схема взаимодействия подпрограмм по управлению. В простейшем случае такая схема показывает общую структуру программы, т.е. отображает два уровня иерархии. Тот же метод позволяет получить структурные схемы с большим количеством уровней.

Разбиение на модули выполняется эвристически, исходя из рекомендуемых размеров модулей (20-60 строк) и сложности структуры (2-3 вложенных управляющих конструкции). Для анализа технологичности иерархии модулей используют методики *Константайна* или *Джексона* [14].

Практически одновременно появились *методики* проектирования программного обеспечения *Джексона* и *Варнье-Орра*, также основанные на декомпозиции данных.

Обе методики предназначены для создания «простых» программ, работающих со сложными, но иерархически организованными структурами данных. При разработке программных систем предлагается вначале разбить систему на отдельные программы, а затем использовать эти методики. Они могут использоваться только в том случае, если данные разрабатываемых программ могут быть представлены в виде иерархии или совокупности иерархий.

Методика Джексона основана на поиске соответствий структур исходных данных и результатов. Однако при ее применении возможны ситуации, когда на каких-то уровнях соответствия отсутствуют. Например, записи исходного файла сортированы не в том порядке, в котором соответствующие строки должны появляться в отчете. Такие ситуации были названы «*столкновениями*».

Методика Варнье-Орра базируется на том же положении, что и методика Джексона, но основными при построении программы считаются структуры выходных данных и, если структуры входных данных не соответствуют структурам выходных, то их допускается менять.

Таким образом, ликвидируется основная причина столкновений. Однако на практике не всегда существует возможность пересмотра структур входных данных: эти структуры уже могут быть строго заданы, например, если данные получены при выполнении других программ, поэтому эту методику применяют реже.

Лекция №11. Особенности проектирования программного обеспечения при объектном подходе

Цель: получить представление об особенностях проектирования программного обеспечения при объектном подходе, а также об использовании моделей и диаграмм UML при разработке.

Как показывает практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Когда в середине 80-х годов XX века для разработки программного обеспечения большого объема было предложено использовать *объектный* подход, в качестве основной технологии выбрано *объектно-ориентированное программирование* (ООП) - методика разработки программ, в основе которой лежит понятие *объекта* [15].

Объект - это некоторая структура, соответствующая объекту реального мира, его поведению. Каждый объект имеет *состояние*, обладает четко определенным *поведением* и *уникальной идентичностью*. Совокупность *атрибутов* (свойств) и их значений характеризует объект. Объекты, описываемые одинаковыми наборами атрибутов, объединяются в *классы*. Данные класса называются *полями*, процедуры и функции - *методами*. Все *экземпляры* одного класса (объекты, порожденные от одного класса) имеют один и тот же набор свойств и общее поведение, то есть одинаково реагируют на одинаковые сообщения. Каждый объект имеет определенное время жизни. В процессе выполнения программы или функционирования реальной системы, могут создаваться новые объекты и уничтожаться уже существующие.

Поскольку объект - это динамическая структура, *переменная-объект* содержит не данные, а *ссылку* на данные объекта. Поэтому программист должен позаботиться о выделении памяти для этих данных. Выделение памяти при создании объекта осуществляется при помощи специального метода класса – *конструктора* (*constructor*), а освобождение памяти при его уничтожении - при помощи *деструктора* (*destructor*).

Важнейшими понятиями ООП являются *инкапсуляция*, *наследование*, *полиморфизм*, которые позволяют конструировать сложные объекты из сравнительно простых. Программа, написанная с использованием ООП, состоит из множества объектов, взаимодействующих между собой путем передачи *сообщений*.

В основе объектного подхода к разработке программного обеспечения лежит объектная декомпозиция, т. е. представление разрабатываемого программного продукта в виде совокупности объектов, в процессе взаимодействия которых через передачу сообщений и происходит выполнение требуемых функций. Объектно-ориентированный подход имеет следующие преимущества:

а) уменьшение сложности программного обеспечения;

б) повышение надежности программного обеспечения;

в) обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;

г) обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Систематическое применение объектного подхода позволяет разрабатывать хорошо структурированные, надежные в эксплуатации, достаточно просто модифицируемые программные системы, поэтому ООП является одним из наиболее интенсивно развивающихся направлений теоретического и прикладного программирования. Однако при объектном подходе сразу можно выполнить декомпозицию только очень простого программного обеспечения.

На заре эпохи ООП были предложены методы анализа и проектирования в рамках объектного подхода, использующие различные модели и нотации. Спорить о достоинствах и недостатках этих методов и моделей можно было бесконечно. Эта ситуация получила название «*войны методов*». Конец «войне методов» положило появление в 1995 г. первой версии языка *UML* (*Unified Modeling Language* - унифицированный язык моделирования), который был создан ведущими специалистами в этой области (Гради Бучем, Иваром Якобсоном и Джеймсом Рамбо) и в настоящее время фактически признан стандартным средством описания проектов, создаваемых с использованием объектно-ориентированного подхода [16,17,18]. UML объединил лучшие современные технические приемы моделирования и разработки программного обеспечения. По сути, язык UML был задуман так, чтобы его можно было реализовать посредством его же инструментальных средств. Фактически это признание того, что большие современные программные системы, как правило, нуждаются в инструментальной поддержке. UML-диаграммы легко воспринимаются и при этом без труда генерируются компьютерами. UML объединил лучшие идеи «доисторических» методов, отказываясь от их наиболее специфических деталей.

Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей [6]: *использования*, *логическую*, *реализации*, *процессов* и *развертывания* (рисунок А.20).

Логическая модель описывает ключевые абстракции ПО (классы, интерфейсы), т. е. средства, обеспечивающие требуемую функциональность.

Модель реализации определяет реальную организацию программных модулей в среде разработки.

Модель использования представляет собой описание функциональности программного продукта с точки зрения пользователя.

Модель процессов отображает организацию вычислений и оперирует понятиями «процессы» и «нити». Она позволяет оценить производительность, масштабируемость и надежность программного обеспечения.

Модель развертывания показывает особенности размещения программных компонентов на конкретном оборудовании.

Каждая из указанных моделей характеризует определенный аспект проектируемой системы, а все вместе они составляют относительно полную модель разрабатываемого программного продукта.

В процессе анализа требований необходимо понять, что именно ожидают от системы пользователи программного обеспечения. Для этого в распоряжении имеется множество приемов UML. В общей сложности *UML2* предлагает тринадцать дополняющих друг друга диаграмм [16,18], входящих в различные модели (рисунок А.22). Все диаграммы используют единую графическую нотацию, что существенно облегчает их понимание. В общем случае можно выделить два вида диаграмм *UML*: *диаграмма структуры* и *диаграмма поведения.* Каждый из этих видов включает набор диаграмм, которые более детально с разных точек зрения описывают модели. Рассмотрим более подробно эти диаграммы.

В состав *диаграммы структуры* входят шесть диаграмм:

1) *Диаграмма классов* описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. Диаграмма классов строится с точки зрения концептуальной перспективы и может служить хорошим инструментом для построения точного словаря предметной области. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

2) *Диаграмма пакетов* демонстрирует взаимосвязи наборов классов, объединенных в пакеты (логические блоки системы, представляющие группы времени компиляции), между собой и помогает держать их под контролем. Фактически - это хорошая логическая маршрутная карта системы. Диаграммы пакетов исключительно удобны в больших по размерам системах для представления картины зависимостей между основными элементами системы.

3) *Диаграмма объектов* - это снимок объектов системы в какой-то момент времени. Поскольку она показывает экземпляры, а не классы, то диаграмму объектов часто называют *диаграммой экземпляров*. Диаграмму объектов можно использовать для отображения одного из вариантов конфигурации объектов.

4) *Диаграмма компонентов* показывает, из каких программных компонентов (элементов, которые можно независимо друг от друга купить и обновить) состоит программное обеспечение и как эти компоненты связаны между собой.

5) *Диаграмма развертывания* (или *диаграмма размещения*) отражает физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения.

6) *Диаграмма составных структур* – новый вид диаграмм, появившийся в языке *UML2*. Одной из наиболее значимых новых черт языка

UML2 является возможность превращать класс в иерархию внутренних структур. Это позволяет разбить сложный объект на составляющие - составные структуры, которые представляют группы времени выполнения.

В состав *диаграммы поведения* входят четыре основные диаграммы:

1) *Диаграмма прецедентов* описывает типичные взаимодействия между пользователями системы и самой системой. В первой версии *UML* позиционировалась как *диаграмма вариантов использования,* которая демонстрировала основные функции системы для каждого типа пользователей. Фактически *прецеденты* – это способ записи требований или вариант использования системы одним из *актеров* (пользователей). Пример диаграммы прецедентов показан на рисунке А.21.

2) *Диаграмма деятельности* представляет собой схему потоков управления для решения некоторой задачи по отдельным действиям, допускают наличие параллельных и/или альтернативных действий. Это - технология, позволяющая описывать логику поведения системы, бизнес-процессы и потоки работ. Во многих случаях они напоминают блок-схемы, но в отличие от них поддерживают параллельное процессы.

3) *Диаграмма конечных автоматов* (или *диаграмма состояний*) демонстрирует состояния одного объекта и условия переходов из одного состояния в другое. Средством моделирования динамического поведения объекта является конечный автомат.

4) *Диаграмма взаимодействия* служит для уточнения основных аналитических диаграмм взаимодействий с дополнением деталей реализации, а также для иллюстрации технических вопросов, возникших при проектировании. К диаграммам взаимодействия относятся:

а) *диаграмма последовательностей* отображает упорядоченное по времени взаимодействие объектов в процессе выполнения варианта использования, акцент делается именно на последовательность взаимодействия;

б) *коммуникационная диаграмма* (в *UML1* называлась *диаграммой кооперации*) - особый вид диаграмм взаимодействия, в которых внимание акцентируется на обмене данными между различными участниками взаимодействия;

в) *диаграмма обзора взаимодействий* - комбинация диаграмм деятельности и диаграмм последовательности, впервые появилась в *UML2*;

г) *временная диаграмма* - разновидность диаграммы взаимодействий, основное внимание в которой направлено на моделирование временных ограничений. Временная диаграмма идеально подходит для моделирования систем реального времени, а также полезна для обозначения временных интервалов между изменениями состояний различных объектов.

Решение о том, какие именно диаграммы следует построить при анализе требований и в процессе проектирования программного обеспечения принимается непосредственно разработчиками применительно к конкретному проекту. Дополнениями к диаграммам служат формализованные и

неформализованные текстовые описания, комментарии и словари. При построении этих и других диаграмм используют унифицированную систему обозначений. *UML* и предлагаемая теми же авторами методика *Rational Unified Process* поддерживаются пакетом *Rational Rose* фирмы *Rational Software Corporation*. Ряд диаграмм *UML* можно построить также средствами программы *Microsoft Visual Modeler* и других *CASE*-средств.

Лекция №12. Пользовательские интерфейсы

Цель: ознакомиться с типами пользовательских интерфейсов и основными принципами их разработки.

Еще не так давно программисты использовали весьма ограниченный арсенал средств взаимодействия пользователей с создаваемыми программами. Как правило, такое взаимодействие заключалось в обмене текстовыми сообщениями либо псевдографическими изображениями. *Пользовательский интерфейс* рассматривался как средство общения человека с операционной системой и был достаточно примитивным. При этом «промышленное» программное изделие было ориентировано на людей, если не профессионалов, то достаточно хорошо знакомых с вычислительной техникой. Однако в последние годы ситуация коренным образом изменилась.

С распространением персональных компьютеров невероятно возросло число их пользователей, в том числе не имеющих даже начальных знаний в области вычислительной техники. Значительно увеличилось и число программирующих пользователей компьютеров как имеющих соответствующую базовую подготовку, так и «самоучек», в распоряжении которых теперь имеются мощные средства разработки, позволяющие создавать программы с практически неограниченными интерактивными возможностями. Наконец наличие такого уникального средства общения, как Интернет, позволяет всем желающим выставлять на всеобщее обозрение плоды своего творчества, вне зависимости от их качества и предназначения.

В результате на свет стало появляться все больше программных продуктов-однодневок, которые снабжены ярким и впечатляющим интерфейсом в виде многочисленных и разнообразных кнопок, пиктограмм, переключателей и т.п., однако никакой пользы не приносят. С другой стороны, действительно полезные программы, снабженные неудачным интерфейсом, остаются невостребованными. Известны случаи, когда достаточно крупные проекты были отклонены заказчиком только из-за того, что не были своевременно и качественно решены вопросы, связанные с пользовательским интерфейсом. Кроме того, полнота использования потенциальных возможностей программного продукта зависит от качества пользовательского интерфейса. Важность грамотной разработки пользовательского интерфейса давно осознана ведущими производителями программного обеспечения, а его качество регламентируются целым рядом

соответствующих стандартов. Те программные продукты, которые им не удовлетворяют, практически не имеют шансов на «выживание». В настоящее время основной проблемой является разработка интерактивных интерфейсов к сложным программным продуктам для непрофессиональных пользователей.

Сложность понятия «*пользовательский интерфейс*» породила множество определений, которые менялись в процессе развития самого пользовательского интерфейса. Пользовательский интерфейс часто понимают только как внешний вид программы, однако такое понимание является слишком узким. В действительности он объединяет в себе все элементы и компоненты программы, которые способны оказывать влияние на взаимодействие пользователя с программным обеспечением, а это уже не только экран, который видит пользователь. К элементам пользовательского интерфейса относятся:

- набор задач пользователя, которые он решает при помощи системы;
- используемая системой метафора (например, рабочий стол в *Windows*);
- элементы управления системой;
- навигация между блоками системы;
- визуальный (и не только) дизайн экранов программы;
- средства отображения информации, отображаемая информация и форматы;
- устройства и технологии ввода данных;
- диалоги, взаимодействие и транзакции между пользователем и компьютером;
- обратная связь с пользователем;
- поддержка принятия решений в конкретной предметной области;
- порядок использования программы и документация на нее.

Пользовательский интерфейс представляет собой совокупность программных и аппаратных средств, обеспечивающих взаимодействие пользователя с компьютером. Основу такого взаимодействия составляют *диалоги*.

Под *диалогом* понимают регламентированный обмен информацией между человеком и компьютером, осуществляемый в реальном масштабе времени и направленный на совместное решение конкретной задачи: обмен информацией и координация действий. Каждый диалог состоит из отдельных процессов ввода-вывода, которые физически обеспечивают связь пользователя и компьютера.

Обмен информацией осуществляется передачей *сообщений* и управляющих сигналов. *Сообщение* - порция информации, участвующая в диалоговом обмене. Различают:

а) *входные сообщения*, которые генерируются человеком с помощью средств ввода: клавиатуры, манипуляторов, например мыши;

б) *выходные сообщения*, которые генерируются компьютером в виде текстов, звуковых сигналов и/или изображений и выводятся пользователю на экран монитора или другие устройства вывода информации (рисунок А.23).

В основном пользователь генерирует сообщения следующих типов: запрос информации; запрос помощи; запрос операции или функции; ввод или изменение информации; выбор поля кадра и т. д. В ответ пользователь получает от компьютера: подсказки или справки; информационные сообщения, не требующие ответа; приказы, требующие действий; сообщения об ошибках; изменение формата кадра и т. д.

Интерфейс - это набор правил, которые можно обобщить, сгруппировать по общему признаку. Тогда *вид интерфейса* - это объединение по схожести способов взаимодействия человека и компьютеров. Среди различных интерфейсов общения человека и компьютера можно выделить:

- *командный интерфейс*;
- *графический интерфейс*;
- *речевой интерфейс*;
- *мимический интерфейс*;
- *семантический интерфейс*.

Командный интерфейс получил свое название благодаря тому, что в этом виде интерфейса пользователь подает команды компьютеру, а компьютер их выполняет и выдает результат. Он реализован в виде комбинации *пакетной технологии* и *технологии командной строки*. Преобладающим видом файлов при работе с командным интерфейсом стали текстовые файлы - только их можно было создать при помощи клавиатуры. Интерфейс командной строки активно использовался в *UNIX*.

Идея появления *графического интерфейса* зародилась в середине 70-х годов XX века, когда в исследовательском центре Xerox Palo Alto Research Center (PARC) была разработана концепция визуального интерфейса. Его реализация стала возможной благодаря развитию технической базы компьютеров, уменьшению времени реакции компьютера на команду, а также увеличению объема оперативной памяти. Характерной особенностью *графических интерфейсов* является то, что диалог с пользователем ведется не явно с помощью команд, а через графические образы. Этот вид интерфейса реализован на технологиях *простого графического интерфейса* и *«чистого» WIMP - интерфейса*. Простой графический интерфейс использовался в файловой оболочке *Nortron Commander*, редакторах *Multi-Edit, Лексикон, ChiWriter* и текстовом процессоре *Microsoft Word for Dos*. Ярким примером программ с графическим интерфейсом является операционная система *Microsoft Windows*.

В середине 90-х годов XX века, после появления недорогих звуковых карт и широкого распространения технологий распознавания речи, была разработана так называемая «*речевая технология*» - простейшая реализация *SILK - интерфейса*. *SILK - интерфейс* (*Speech* - речь, *Image* - образ, *Language* - язык, *Knowlege* - знание) наиболее приближен к обычной, человеческой форме общения. Согласно технологии команды подаются голосом путем произнесения специальных зарезервированных слов – команд, т.е. ведется обычный «разговор» пользователя и компьютера. При этом компьютер

находит для себя команды, анализируя человеческую речь и находя в ней ключевые фразы. Результат выполнения команд он также преобразует в понятную пользователю форму. Этот вид интерфейса наиболее требователен к аппаратным ресурсам компьютера, и поэтому его применяют в основном для военных целей. Слова должны выговариваться четко, в одном темпе. Между словами обязательна пауза. Из-за неразвитости алгоритма распознавания речи такие системы требует индивидуальной предварительной настройки на каждого конкретного пользователя.

В основу *мимического интерфейса* положена биометрическая технология, которая возникла в конце 90-х годов XX века. Для управления компьютером используется выражение лица человека, направление его взгляда, размер зрачка и другие признаки. Для идентификации пользователя используется рисунок радужной оболочки его глаз, отпечатки пальцев и другая уникальная информация. Изображения считываются с цифровой видеокамеры, а затем с помощью специальных программ распознавания образов из этого изображения выделяются команды. Эта технология актуальна для программных продуктов и приложений, где важно точно идентифицировать пользователя компьютера.

В конце 70-х годов XX века, с развитием искусственного интеллекта возник *семантический* (*общественный*) *интерфейс*. Его трудно назвать самостоятельным видом интерфейса, поскольку он включает и интерфейс командной строки, и графический, и речевой, и мимический интерфейс. Его основная отличительная черта - это отсутствие команд при общении с компьютером. Запрос формируется на естественном языке, в виде связанного текста и образов. Фактически - это моделирование «общения» пользователя с компьютером. С середины 90-х годов XX века в связи с важным военным значением этих разработок это направление было засекречены.

Лекция №13. Типы пользовательских интерфейсов

Цель: ознакомиться с основными требованиями к разработке интерфейсов с учетом психофизических особенностей человека, а также принципами построения пользовательской и программной моделей интерфейса.

Различают два основных подхода к разработке интерфейсов: процедурно-ориентированный и объектно-ориентированный подходы (рисунок А.24).

Процедурно-ориентированные модели интерфейсов используют традиционную модель взаимодействия с пользователем, основанную на понятиях «*процедура*» и «*операция*». В рамках этой модели программное обеспечение предоставляет пользователю возможность выполнения некоторых действий, для которых пользователь определяет соответствующие данные и следствием выполнения которых является получение желаемых

результатов. Реализация современного процедурно-ориентированного пользовательского интерфейса на базе структурного подхода является очень сложной и трудоемкой задачей.

Объектно-ориентированные модели интерфейсов используют несколько иную модель взаимодействия с пользователем, ориентированную на манипулирование *объектами* предметной области. В рамках этой модели пользователю предоставляется возможность напрямую взаимодействовать с каждым объектом и инициировать выполнение операций, в процессе которых взаимодействуют несколько объектов. Пользователю предоставляется возможность создавать объекты, изменять их параметры и связи с другими объектами, а также инициировать взаимодействие этих объектов. Элементы интерфейсов данного типа включены в пользовательский интерфейс *Windows*.

В таблице А.4 перечислены основные отличия пользовательских моделей интерфейсов процедурного и объектно-ориентированного типов.

Среди процедурно-ориентированных интерфейсов различают три типа: *примитивные, меню* и *со свободной навигацией* [12].

Примитивным называют интерфейс, который организует взаимодействие с пользователем в консольном режиме. Обычно такой интерфейс реализует конкретный сценарий работы программного обеспечения (ввод данных - решение задачи - вывод результата). Единственное отклонение от последовательного процесса, которое обеспечивается данным интерфейсом, заключается в организации цикла для обработки нескольких наборов данных. Подобные интерфейсы в настоящее время используют только в процессе обучения программированию или в тех случаях, когда вся программа реализует одну функцию, например, в некоторых системных утилитах.

Интерфейс-меню позволяет пользователю выбирать необходимые операции из специального списка, выводимого ему программой. Эти интерфейсы предполагают реализацию множества сценариев работы, последовательность действий в которых определяется пользователем. Различают одноуровневые и иерархические меню. Интерфейсы данного типа несложно реализовать в рамках структурного подхода к программированию. Интерфейсы-меню в настоящее время используют редко и только для сравнительно простого программного обеспечения или в разработках, которые должны быть выполнены по структурной технологии и без использования специальных библиотек.

Интерфейсы *со свободной навигацией* также называют *графическими пользовательскими интерфейсами* (*GUI - Graphic User Interface*) или интерфейсами *WYSIWYG* (*What You See Is What You Get* - что видишь, то и получишь, т. е. *что* пользователь видит на экране, *то* он и получит при печати). Эти названия подчеркивают, что интерфейсы данного типа ориентированы на использование экрана в графическом режиме с высокой разрешающей способностью. Графические интерфейсы поддерживают концепцию интерактивного взаимодействия с программным обеспечением, осуществляя визуальную обратную связь с пользователем и возможность прямого

манипулирования объектами и информацией на экране. Кроме того, интерфейсы данного типа поддерживают концепцию совместимости программ, позволяя перемещать между ними информацию.

Интерфейс со свободной навигацией обеспечивает возможность осуществления любых допустимых в конкретном состоянии операций, доступ к которым возможен через различные интерфейсные компоненты. Например, окна программ, реализующих интерфейс Windows, обычно содержат меню различных типов (ниспадающее, кнопочное, контекстное), разного рода компоненты ввода данных. Причем выбор следующей операции в меню осуществляется как мышью, так и с помощью клавиатуры. Существенной особенностью интерфейсов данного типа является способность изменяться в процессе взаимодействия с пользователем, предлагая выбор только тех операций, которые имеют смысл в конкретной ситуации. Реализуют интерфейсы со свободной навигацией, используя событийное программирование и объектно-ориентированные библиотеки, что предполагает применение визуальных сред разработки программного обеспечения.

Объектно-ориентированные интерфейсы пока представлены только *интерфейсом прямого манипулирования*. Этот тип интерфейса предполагает, что взаимодействие пользователя с программным обеспечением осуществляется посредством выбора и перемещения *пиктограмм,* соответствующих объектам предметной области. Для его реализации используют событийное программирование и объектно-ориентированные библиотеки.

При проектировании пользовательских интерфейсов необходимо учитывать психофизические особенности человека, связанные с восприятием, запоминанием и обработкой информации. Исследованием принципов работы мозга человека занимается *когнитивная психология* [6].

Информация о внешнем мире поступает в мозг в огромных количествах. При этом *краткосрочная память* является своего рода оперативной памятью мозга, невостребованная информация хранится в ней не более 30 секунд. Ее емкость приблизительно равна 7 ± 2 несвязанных объектов. Чтобы не забыть важную информацию, мы обычно повторяем ее «про себя», «обновляя» информацию в краткосрочной памяти. Поэтому при проектировании интерфейсов следует иметь в виду, что большинству людей сложно, например, запомнить и ввести на другом экране число, содержащее более 5 цифр (7 - 2), или некоторое сочетание букв.

Каждый человек вносит в деятельность свое понимание того, как она должна выполняться. Это понимание - *модель деятельности* - базируется на его прошлом опыте. Множество таких моделей хранится в *долговременной памяти* человека - хранилище информации с неограниченной емкостью и временем хранения. В нее записываются постоянно повторяемые сведения или информация, связанная с сильными эмоциями. Однако механизмы извлечения информации из памяти имеют ассоциативный характер. Специальная методика запоминания информации (*мнемоника*) использует именно это свойство памяти: для запоминания информации ее «привязывают» к тем

данным, которые память уже хранит и позволяет легко получить. Поскольку доступ к долговременной памяти затруднен, целесообразно рассчитывать не на то, что пользователь вспомнит нужную информацию, а на то, что он ее узнает. Кроме того, важную роль при разработке дизайна интерфейсов играют *цвет, звук, восприятие времени, использование анимации*.

Цвет является для человека сильным раздражителем, поэтому применять цвета в интерфейсе необходимо крайне осторожно. Следует иметь в виду, что обилие оттенков привлекает внимание, но быстро утомляет. Не стоит ярко раскрашивать окна, с которыми пользователь будет долго работать. Необходимо учитывать и индивидуальные особенности восприятия цветов человеком: примерно каждый десятый плохо различает какие-либо цвета, поэтому в ответственных случаях необходимо предоставить пользователю возможность их настройки.

В интерфейсах *звук* используют с разными целями: для привлечения внимания; как фон, обеспечивающий некоторое состояние пользователя; как источник дополнительной информации. Применяя звук, следует учитывать, что большинство людей очень чувствительны к звуковым сигналам, особенно, если они указывают на наличие ошибки. Поэтому при создании звукового сопровождения целесообразно предусматривать возможность его отключения.

Человеку свойственно *субъективное восприятие времени*. Считают, что внутреннее время связано со скоростью и количеством воспринимаемой и обрабатываемой информации. Занятый человек времени не замечает. Зато в состоянии ожидания время тянется бесконечно: в это время мозг оказывается в состоянии информационного вакуума. Доказано, что при ожидании более 1-2 секунд пользователь может отвлечься, «потерять мысль», что увеличивает усталость и неблагоприятно сказывается на результатах работы. Сократить время ожидания можно, заняв пользователя, но, не отвлекая его от работы. Например, предоставить ему какую-либо информацию для обдумывания или выводить промежуточные результаты.

Известны попытки использования для «развлечения» пользователя *анимации*. Однако интересно смотреть анимацию первый раз, многократный просмотр одного и того же начинает раздражать. Поэтому необходимо соблюдать основное правило: информировать пользователя, что заказанные операции потребуют некоторого времени выполнения, используя индикаторы оставшегося времени, анимацию или изменение формы курсора мыши на песочные часы. Важно точно обозначить момент, когда система готова продолжать работу, для чего значительно изменяют внешний вид экрана.

В итоге взаимодействие пользователя с интерфейсом определяется не только его физическими возможностями и особенностями по восприятию информации, но и *пользовательской моделью* интерфейса. Существуют три модели пользовательского интерфейса: *модель программиста, модель пользователя, программная модель*.

Программист при разработке интерфейса исходит из того, управление какими операциями нужно реализовать в нем и как это осуществить, не

затрачивая существенных ресурсов компьютера, своих сил и времени. Его интересуют эффективность, функциональность, технологичность, внутренняя стройность и другие, не связанные с удобством пользователя характеристики программного обеспечения. Именно поэтому большинство интерфейсов существующих программ вызывают серьезные нарекания пользователей.

С точки зрения здравого смысла хорошим считается интерфейс, при работе с которым пользователь получает именно то, что он ожидал. Представление пользователя о функциях интерфейса можно описать в виде *пользовательской модели интерфейса* - совокупности обобщенных представлений конкретного пользователя или группы пользователей о процессах, происходящих во время работы программной системы. Модель базируется на особенностях опыта конкретных пользователей, который характеризуется уровнем подготовки в предметной области разрабатываемого программного обеспечения, *интуитивными моделями* выполнения операций в этой области, уровнем подготовки в области владения компьютером, а также устоявшимися стереотипами работы с компьютером. Для ее построения необходимо изучить особенности опыта предполагаемых пользователей, для чего используют опросы, тесты и фиксируют на пленку последовательность выполнения операций в реальном процессе. Приведение в соответствие моделей пользователя и программиста, а также построение на их базе *программной модели* интерфейса (рисунок А.25) - задача не простая.

Чем сложнее автоматизируемая предметная область, тем сложнее строить программную модель интерфейса, учитывающую особенности модели программиста и пользовательской модели. С этой точки зрения наиболее перспективны объектные интерфейсы, так как в их основе лежит отображение объектов предметной области, которыми оперируют пользователи. Основой для разработки интерфейса должны стать *интуитивные модели* выполнения операций в предметной области. Нежелание или невозможность следования интуитивным моделям приводит к созданию искусственных надуманных интерфейсов, которые негативно воспринимаются пользователями.

Основное достоинство *хорошего* интерфейса пользователя заключается в том, что *пользователь всегда чувствует, что он управляет программным обеспечением, а не программное обеспечение управляет им*.

Лекция №14. Тестирование и отладка программных продуктов

Цель: ознакомиться с видами и способами контроля и тестирования программного обеспечения, а также методами и средствами отладки программ.

Недостаточно выполнить проектирование и кодирование программного продукта, также необходимо обеспечить его соответствие требованиям и спецификациям. Многократно проводимые исследования показали, что чем

раньше обнаруживаются те или иные несоответствия или ошибки, тем больше вероятность их исправления и ниже его стоимость [14]. Современные технологии разработки программного обеспечения предусматривают раннее обнаружение ошибок за счет выполнения контроля результатов всех этапов разработки. На начальных этапах контроль осуществляют вручную или с использованием *CASE*-средств, затем он принимает форму тестирования.

Тестирование - это процесс выполнения программы, целью которого является выявление ошибок. Никакое тестирование не может доказать отсутствие ошибок в сложном программном обеспечении, поскольку выполнение полного тестирования становится невозможным и имеется вероятность, что остались невыявленные ошибки. Соблюдение основных правил тестирования и научно обоснованный подбор тестов может уменьшить их количество.

Процесс разработки согласно современной модели жизненного цикла программного обеспечения предполагает три стадии тестирования:

- *автономное* тестирование компонентов программного обеспечения;
- *комплексное* тестирование программного обеспечения;
- *системное* или *оценочное* тестирование на соответствие основным критериям качества.

Доля стоимости тестирования в общей стоимости разработки возрастает при увеличении сложности программного обеспечения и повышении требований к его качеству. Для повышения качества тестирования рекомендуется соблюдать следующие принципы:

а) предполагаемые результаты должны быть известны до тестирования;
б) следует избегать тестирования программы автором;
в) необходимо досконально изучать результаты каждого теста;
г) необходимо проверять действия программы на неверных данных;
д) необходимо проверять программу на неожиданные побочные эффекты на неверных данных.

Вероятность наличия необнаруженных ошибок в части программы пропорциональна количеству ошибок уже найденных в этой части. Удачным считают тест, который обнаруживает хотя бы одну ошибку. Формирование набора тестов имеет большое значение, поскольку тестирование является одним из наиболее трудоемких этапов создания программного обеспечения.

Существуют два принципиально различных подхода к формированию тестовых наборов: *структурный* и *функциональный*.

Структурный подход базируется на том, что известна структура тестируемого программного обеспечения, в том числе его алгоритмы («*стеклянный ящик*»). Тесты строятся для проверки правильности реализации заданной логики в коде программы. В основе *структурного тестирования* лежит концепция максимально полного тестирования всех *маршрутов*, предусмотренных алгоритмом (последовательности операторов программы, выполняемых при конкретном варианте исходных данных). Недостатки: построенные тестовые наборы не обнаруживают пропущенные маршруты и ошибки, зависящие от данных; не дают гарантии, что программа правильна.

Функциональный подход основывается на том, что структура программного обеспечения неизвестна. В этом случае тесты строят, опираясь на функциональные спецификации. Этот подход называют также *подходом, управляемым данными,* так как при его использовании тесты строят на базе различных способов декомпозиции множества данных. Наборы тестов, полученные в соответствии с методами этих подходов, объединяют, обеспечивая всестороннее тестирование программного обеспечения.

При *функциональном тестировании* программа рассматривается как «*черный ящик*», а целью тестирования является выяснение обстоятельств, когда поведение программы не соответствует спецификации. Для обнаружения всех ошибок необходимо выполнить *исчерпывающее* тестирование (при всех возможных наборах данных), что для большинства случаев невозможно. Поэтому обычно выполняют «*разумное*» или «*приемлемое*» тестирование, ограничивающееся прогонами программы на небольшом подмножестве всех возможных входных данных. При функциональном тестировании различают следующие методы формирования тестовых наборов: *эквивалентное разбиение*; *анализ граничных значений*; *анализ причинно-следственных связей*; *предположение об ошибке*.

На ранних этапах разработки используют *ручной контроль*. Все проектные решения анализируются с точки зрения их правильности и целесообразности как можно раньше, пока их можно легко пересмотреть.

Различают *статический* и *динамический* подходы к ручному контролю. При *статическом* подходе анализируют структуру, управляющие и информационные связи программы, ее входные и выходные данные. При *динамическом* - выполняют *ручное тестирование* (вручную моделируют процесс выполнения программы на заданных исходных данных). Исходными данными для таких проверок являются: техническое задание, спецификации, структурная и функциональная схемы программного продукта, схемы отдельных компонентов, а для более поздних этапов - алгоритмы и тексты программ, а также тестовые наборы. Доказано, что ручной контроль способствует существенному увеличению производительности и повышению надежности программ и с его помощью можно находить от 30 до 70 % ошибок логического проектирования и кодирования.

Основными методами ручного контроля являются: *инспекции исходного текста*, *сквозные просмотры*, *проверка за столом*, *оценки программ*.

При *комплексном тестировании* используют тесты, построенные по методам эквивалентных классов, граничных условий и предположении об ошибках, поскольку структурное тестирование для него не применимо.

Одним из самых сложных является вопрос о завершении тестирования, так как невозможно гарантировать, что в программе не осталось ошибок. Часто тестирование завершают потому, что закончилось время, отведенное на его выполнение. Его сворачивают, обходясь *минимальным тестированием* [19], которое предполагает: тестирование граничных значений, тщательную проверку руководства, тестирование минимальных конфигураций

технических средств, возможности редактирования команд и повторения их в любой последовательности, устойчивости к ошибкам пользователя.

После завершения комплексного тестирования приступают к *оценочному тестированию*, целью которого является поиск несоответствий техническому заданию.

Таким образом, при разработке программного обеспечения нужно придерживаться следующих принципов:

- главное правило тестирования - не избегайте тестирования;
- практически доказано: чем лучше изначально написан код, тем меньше ошибок он будет содержать;
- тестирование граничных условий непосредственно при написании кода - эффективный способ удалить множество мелких глупых ошибок;
- необходимо активно использовать автоматизированное тестирование, поскольку машины не делают ошибок, не устают и не занимаются самообманом;
- проведение тестирования после внесения каждого, даже небольшого, изменения - это верный способ локализации источника проблем, поскольку новые ошибки появляются, как правило, именно в новом коде.

Отладка - это процесс *локализации* и *исправления* ошибок, обнаруженных при тестировании программного обеспечения. Под *локализацией* ошибки понимают определение оператора программы, выполнение которого вызвало нарушение вычислительного процесса. Для *исправления* ошибки необходимо определить ее причину. Отладка требует от программиста глубоких знаний специфики управления используемыми техническими средствами, операционной системы, среды и языка программирования, реализуемых процессов, природы и специфики ошибок, методик отладки и соответствующих программных средств. По своей сути, отладка психологически дискомфортна, поскольку нужно искать собственные ошибки в условиях ограниченного времени. Кроме того, отладка оставляет возможность взаимовлияния ошибок в разных частях программы. Четко сформулированные методики отладки отсутствуют. В зависимости от этапа обработки, на котором могут проявляться ошибки, различают:

а) *синтаксические ошибки* – сопровождаются комментарием с указанием их местоположения, фиксируются компилятором (транслятором) при выполнении синтаксического и частично семантического анализа;

б) *ошибки компоновки* - обнаруживаются компоновщиком (редактором связей) при объединении модулей программы;

в) *ошибки выполнения* - обнаруживаются аппаратными средствами, операционной системой или пользователем при выполнении программы, проявляются разными способами и в свою очередь делятся на группы:

– *ошибки определения исходных данных* (ошибки передачи, ошибки преобразования, ошибки перезаписи и ошибки данных);

– *логические ошибки проектирования* (неприменимый метод, неверный алгоритм, неверная структура данных, другие) и кодирования (ошибки

некорректного использования переменных, вычислений, межмодульного интерфейса, реализации алгоритма, другие);

– *ошибки накопления погрешностей* результатов вычислений (игнорирование ограничений разрядной сетки и способов уменьшения погрешности).

Отладка программы в любом случае предполагает обдумывание и логическое осмысление всей имеющейся информации об ошибке. При правильном подходе отладка может превратиться в удовольствие типа разгадывания головоломки. Хорошо написанный код сразу содержит меньше ошибок, а те, что все же имеются, гораздо легче обнаружить. Большинство ошибок можно обнаружить по косвенным признакам посредством тщательного анализа текстов программ и результатов тестирования без получения дополнительной информации с помощью следующих методов:

а) *ручного тестирования* (при обнаружении ошибки нужно выполнить тестируемую программу вручную, используя исходный тестовый набор);

б) *индукции* (метод основан на тщательном анализе симптомов ошибки, которые могут проявляться как неверные результаты вычислений или как сообщение об ошибке);

в) *дедукции* (вначале формируют множество причин, которые могли бы вызвать данное проявление ошибки, а затем, анализируя причины, исключают те, которые противоречат имеющимся данным);

г) *обратного прослеживания* (для точки вывода неверного результата строится гипотеза о значениях основных переменных, которые могли бы привести к получению данного результата, а затем, исходя из этой гипотезы, делают предположения о значениях переменных в предыдущей точке).

Для получения дополнительной информации об ошибке выполняют *добавочные* тесты и используют специальные методы и средства: *отладочный вывод, интегрированные средства отладки, независимые отладчики*.

Общая методика отладки программных продуктов, написанных для выполнения в операционных системах *MS DOS* и *Win32*, предполагает:

1 этап - изучение проявления ошибки;

2 этап – определение локализации ошибки;

3 этап - определение причины ошибки;

4 этап — исправление ошибки;

5 этап - повторное тестирование.

Процесс отладки можно существенно упростить, если следовать основным рекомендациям структурного подхода к программированию:

а) программу наращивать «сверху-вниз», от интерфейса к обрабатывающим подпрограммам, тестируя ее по ходу добавления подпрограмм;

б) выводить пользователю вводимые им данные для контроля и проверять их на допустимость сразу после ввода;

в) предусматривать вывод основных данных во всех узловых точках алгоритма (ветвлениях, вызовах подпрограмм).

Лекция №15. Составление программной документации

Цель: ознакомиться с видами программных документов и основными правилами их оформления.

Одним из главных отличий программы от программного продукта является наличие разнообразной, хорошо подготовленной *документации*. Составление программной документации - важный процесс. На каждый программный продукт должна разрабатываться документация двух типов [16]: для пользователей различных групп и для разработчиков. Отсутствие документации любого типа для конкретного программного обеспечения недопустимо. К *программным* относят документы, содержащие сведения, необходимые для разработки, сопровождения и эксплуатации программного обеспечения. Документирование программного обеспечения осуществляется в соответствии с Единой системой программной документации (*ГОСТ 19.XXX*).

В Республике Казахстан с этой целью разработаны и используются стандарты: СТ РК ИСО/МЭК 6592 «Руководство по документированию компьютерных прикладных систем» и СТ РК 34.010-2002 «Информационная технология. Порядок проведения экспертизы программной документации». Оба стандарта базируются на документе *ГОСТ 19.101-77*, который устанавливает виды программных документов для ПО различных типов. К основным программным документам по этому стандарту относятся: *спецификация, ведомость держателей подлинников, текст программы, описание программы, ведомость эксплуатационных документов, формуляр, описание применения, руководство системного программиста, руководство программиста, руководство оператора, описание языка, руководство по техническому обслуживанию, программа и методика испытаний, пояснительная записка.*

Допускается объединять отдельные виды эксплуатационных документов. Необходимость объединения указывается в *техническом задании*, а имя берут у одного из объединяемых документов. При оформлении текстовых и графических материалов, входящих в программную документацию, также следует придерживаться действующих стандартов. Рассмотрим подробнее некоторые из перечисленных документов.

Самым главным документом для разработчиков является *техническое задание* (ТЗ), в котором описываются цели и задачи работы, заказчик и исполнители, технические требования, сроки и этапы, требования секретности, форс-мажорные обстоятельства и правила предъявления результатов. Техническое задание должно быть составлено таким образом, чтобы исключить возможные разночтения, все требования должны быть сформулированы так, чтобы их можно было проверить однозначным образом.

Следующим по важности документом является *программа и методика испытаний* (ПМИ). Структурно она подобна ТЗ: практически для каждого пункта ТЗ в ПМИ говорится, как этот пункт будет проверяться. Способы проверки могут быть самыми разными – от пропуска специального теста до

изучения исходных текстов программы, но они должны быть предусмотрены заранее, а не придумываться в момент испытаний. Новички приступают к составлению ПМИ непосредственно перед завершением работ, а опытные руководители составляют и согласовывают с заказчиками одновременно с ТЗ. Хорошо составленная ПМИ является гарантией успешной сдачи работ.

Руководство системного программиста, на современном языке называется *руководством по инсталляции*. В нем описывается порядок установки системы, как проверить корректность поставленной системы, как вносить изменения и т.п. Обычно это простой короткий документ.

Руководство оператора (пользователя) – это основной документ, описывающий, как пользоваться системой. В хорошем руководстве сначала описывается идея системы, основные функции и как ими пользоваться, а уже потом идет описание всех клавиш и меню.

Руководство программиста - это самый объемный документ, описывающий внутреннюю организацию программы. Обычно этот документ идет в паре с документом «*Текст программы*» – одностраничным документом с оглавлением дискеты или CD. Руководство программиста дает заказчику возможность дописать новые фрагменты программы или переделать старые. В современной литературе этот документ называется *SDK* (*Software Development Kit*). Продукт, снабженный *SDK*, может стоить на порядок дороже, чем такой же продукт без него.

Сейчас не принято продавать исходные тексты программ – проблемы с интеллектуальной собственностью. Даже при наличии *SDK* трудно «влезть» в чужую программу, как говорится, себе дороже. Поэтому большое распространение получили *API* (*Application Program Interface*). Программа передается только в виде *DLL* (*библиотека двоичных кодов*), при этом известно, как обратиться к каждой функции из других программ, т.е. известно имя точки входа, количество, типы и значения параметров. Наличие множества *API*, конечно, хуже, чем наличие исходных текстов (например, нельзя переделать что-то в середине функции), зато много проще в использовании.

С другой стороны, все большую популярность приобретает *FSF* (*Free Software Foundation*). Основателем этого движения был Ричард Столман [14], который забил тревогу по поводу попыток крупных фирм запатентовать многие основные алгоритмы и программы: «Дойдет до того, что они запатентуют понятия *цикл* и *подпрограмма*, что мы будем тогда делать?». *FSF* представляет собой собрание программ в исходных текстах. Любой программист может использовать их в своих целях, но все добавления и улучшения, которые он сделал, следует положить в *FSF*. Таким образом, *FSF* представляет собой одно из самых больших доступных хранилищ программ.

В любом случае, при подготовке документации не следует забывать, что она разрабатывается для того, чтобы ее использовали, и потому она должна содержать все необходимые сведения.

Приложение А

Инженерия программного обеспечения

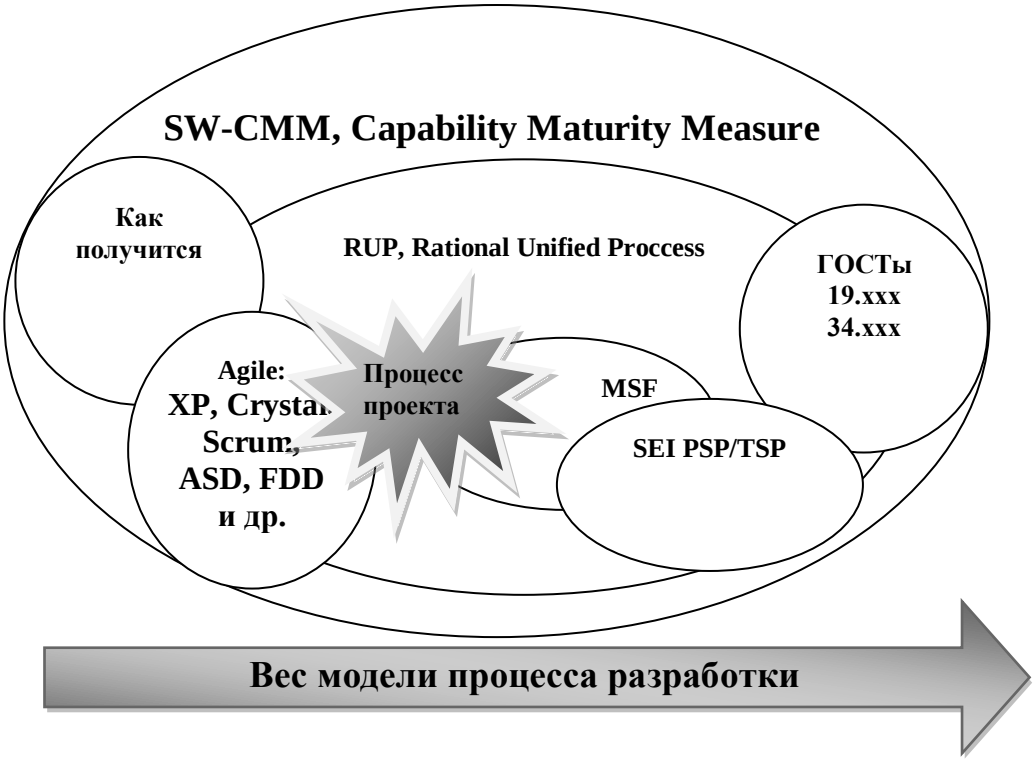


Рисунок А.1 - Модели процесса разработки ПО и их распределение по «весу»

Рисунок А.2 - «Закон четырех П»

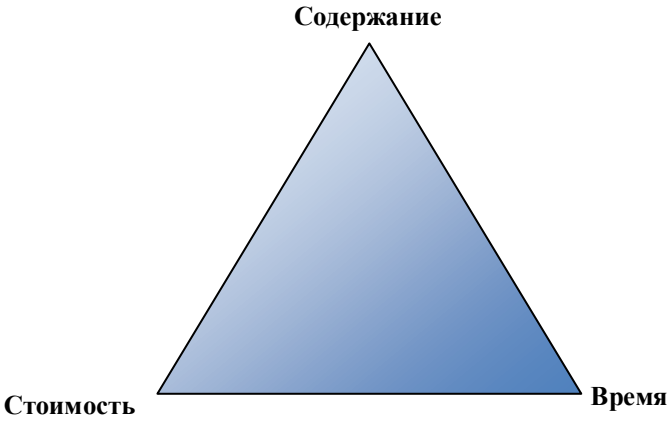


Рисунок А.3 - «Железный треугольник» ограничений проекта

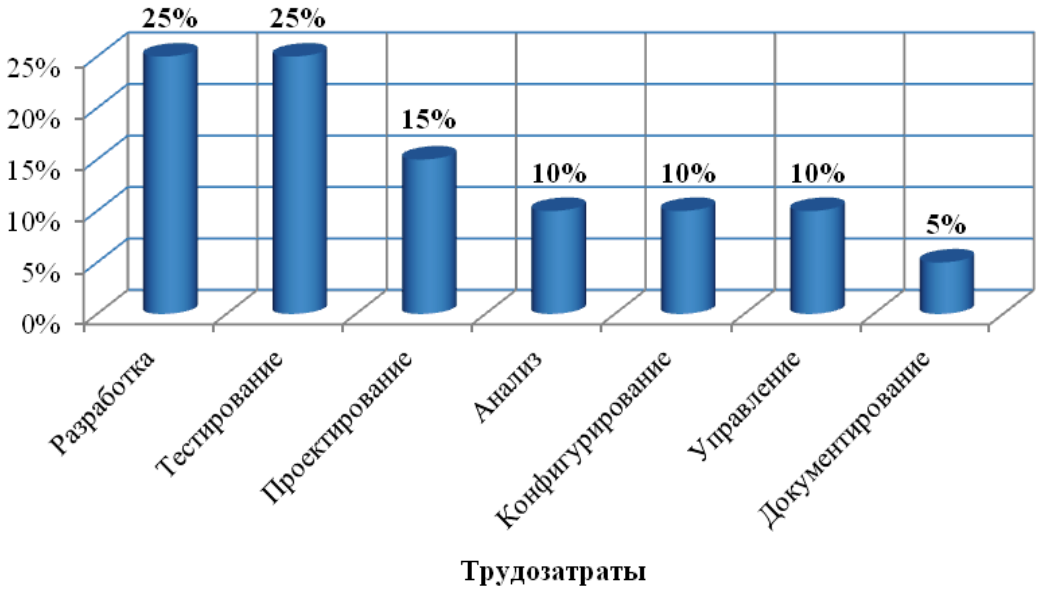


Рисунок А.4 - Распределение трудозатрат по основным производственным процессам при разработке программного обеспечения

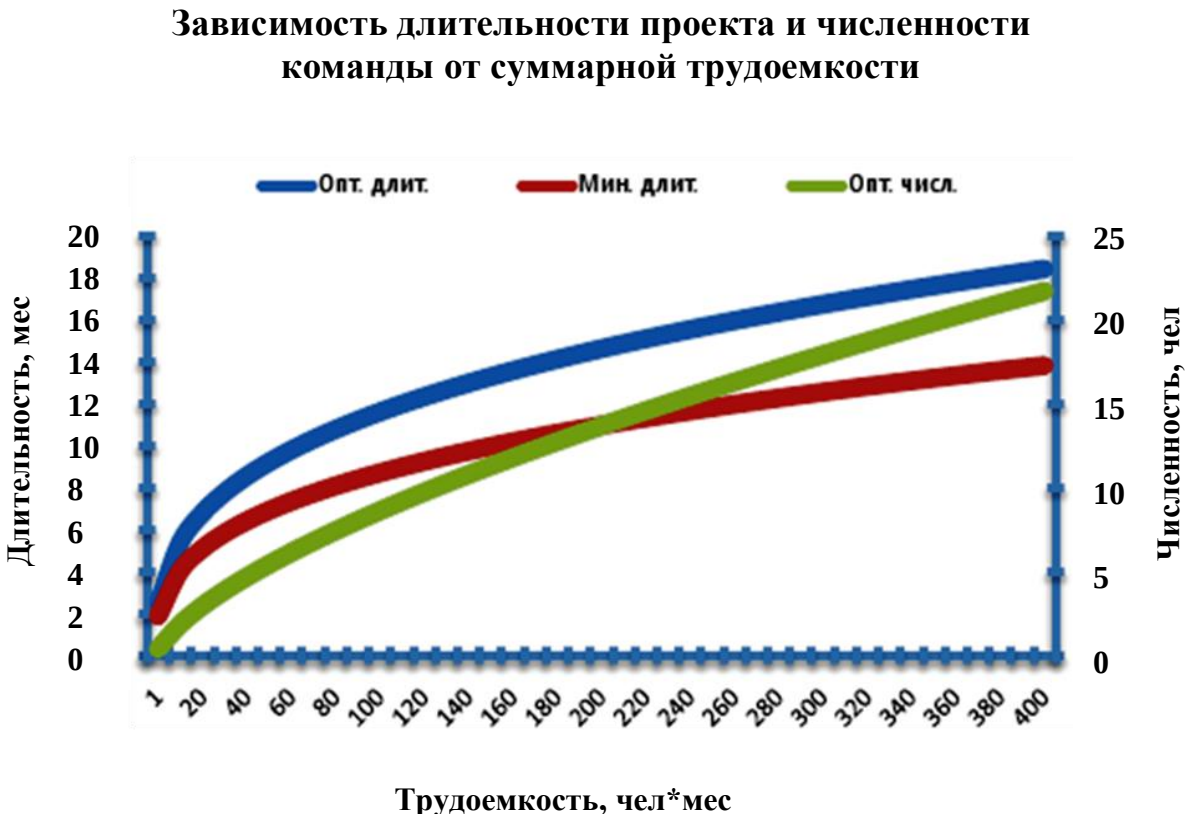


Рисунок А.5 - Закон Б.Боэма

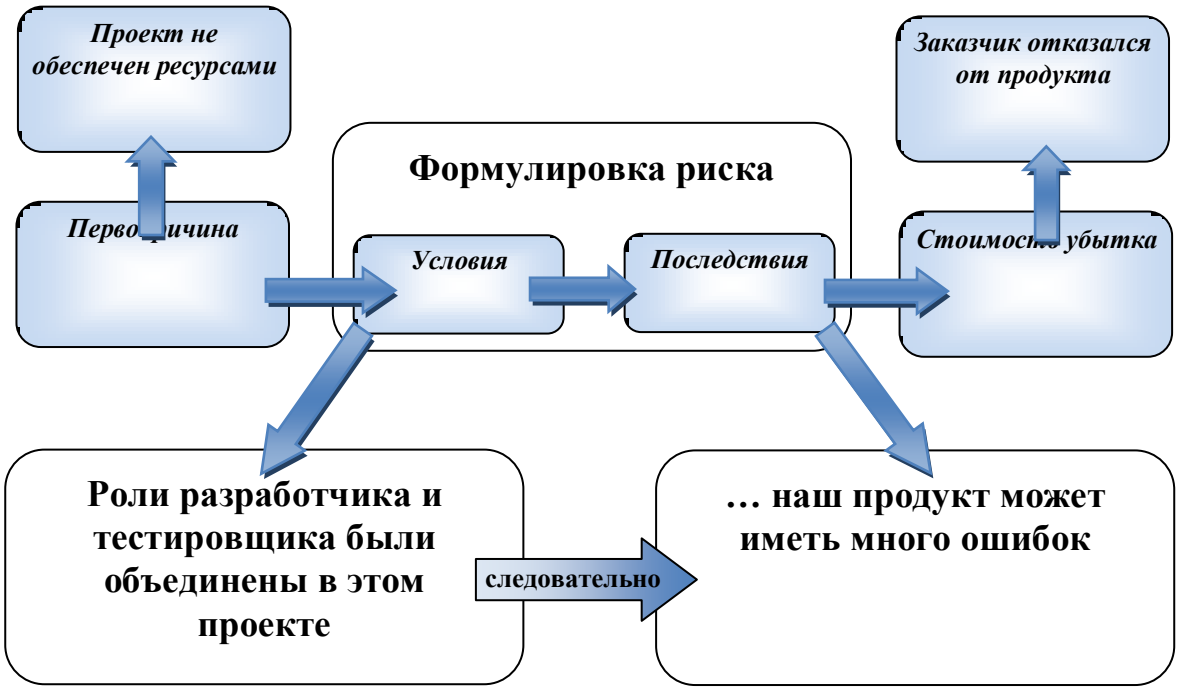


Рисунок А.6 - Пример характеристик риска

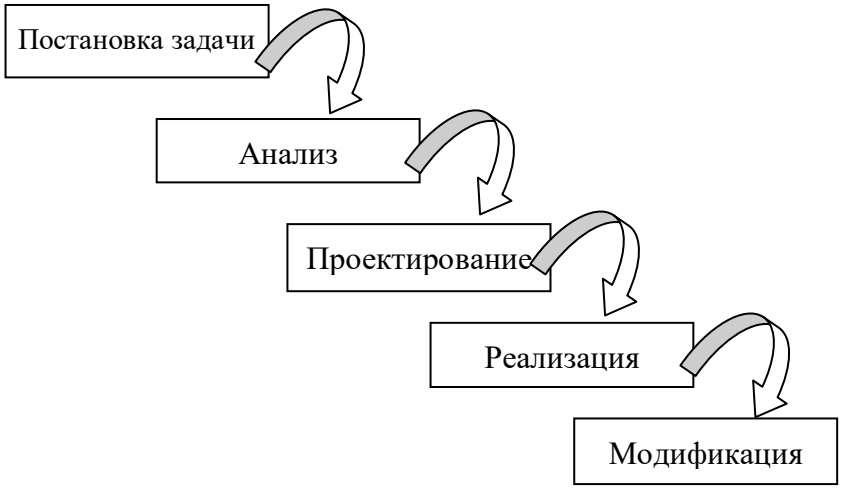


Рисунок А.7 – Каскадная схема разработки программного обеспечения

Рисунок А.8 – Макетирование

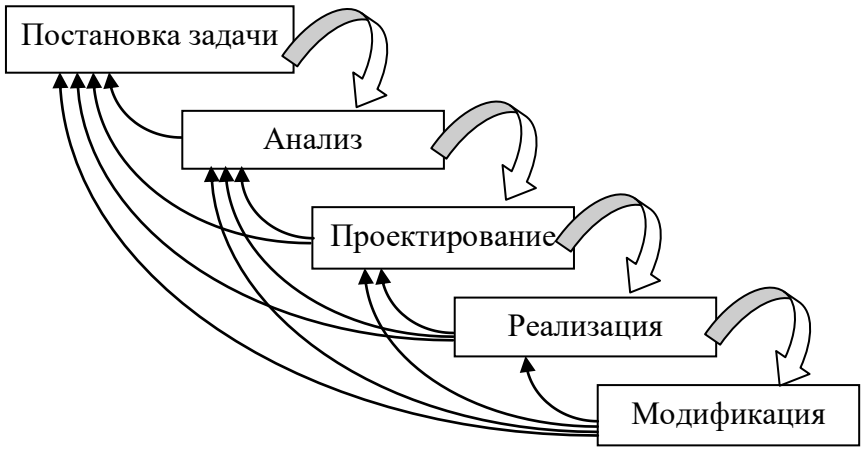


Рисунок А.9 – Схема разработки программного обеспечения с промежуточным контролем

Рисунок А.10 – Схема разработки программного обеспечения с использованием V-образной модели

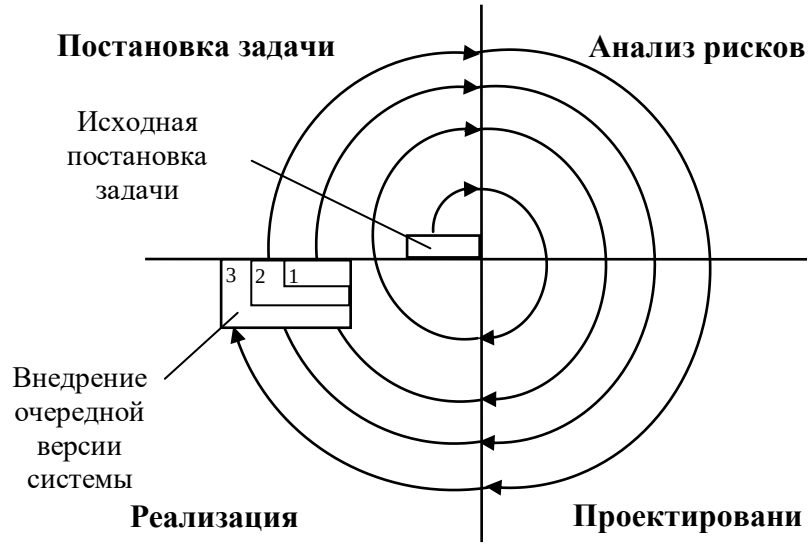


Рисунок А.11 – Спиральная схема разработки программного обеспечения

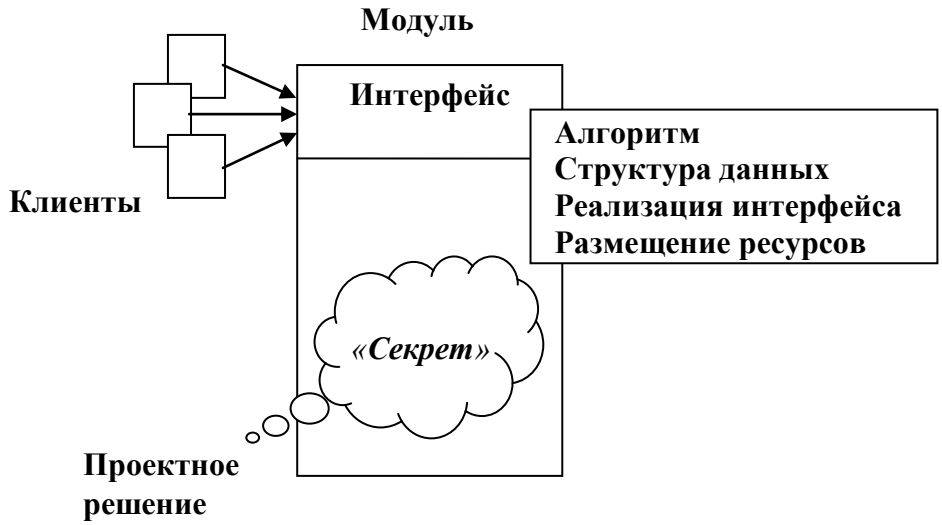


Рисунок А.12 - Информационная закрытость модуля

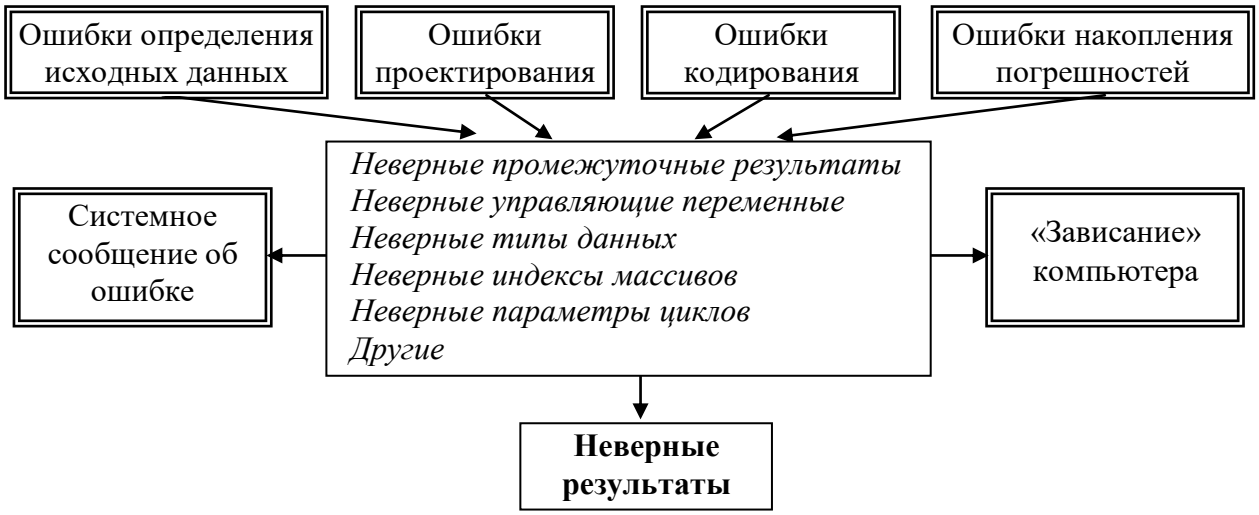


Рисунок А.13 – Способы проявления ошибок

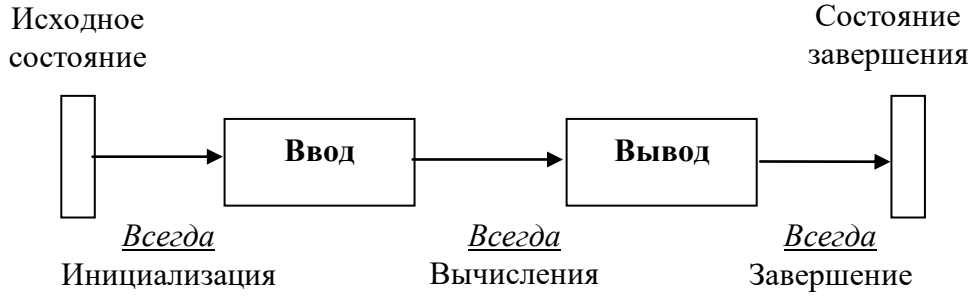


Рисунок А.14 - Пример диаграммы переходов состояний программного обеспечения, не взаимодействующего с окружающей средой

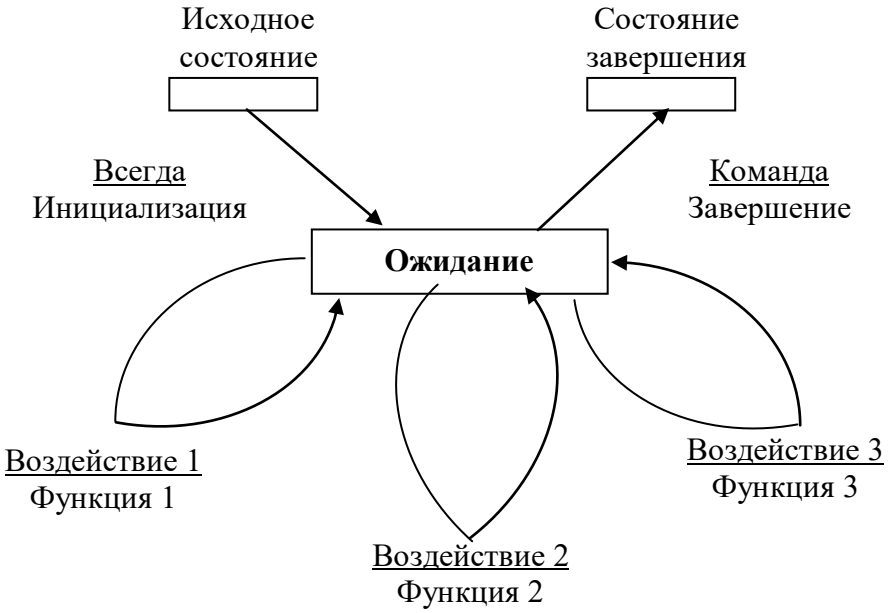


Рисунок А.15 - Пример диаграммы переходов состояний программного обеспечения, активно взаимодействующего с окружающей средой

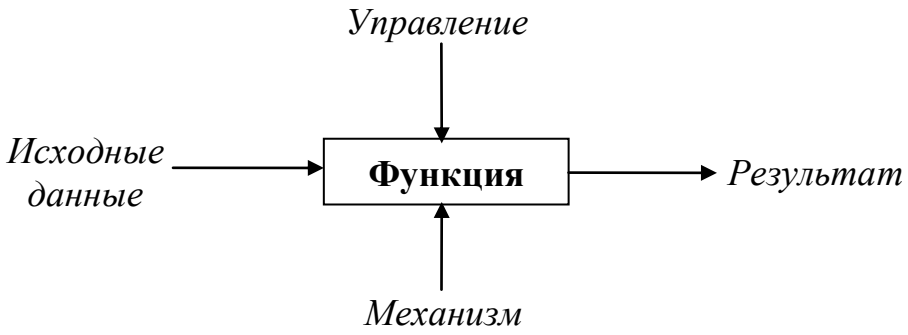


Рисунок А.16 – Функциональный блок и интерфейсные дуги

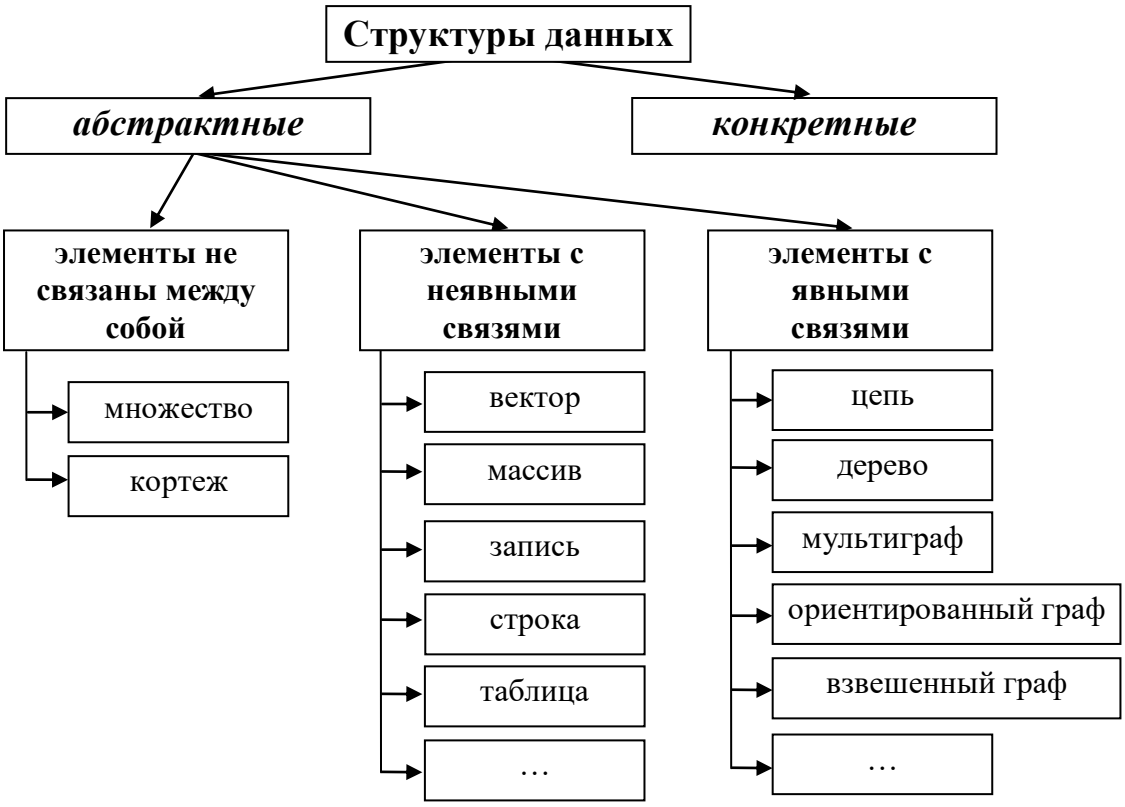


Рисунок А.17 – Классификация структур данных

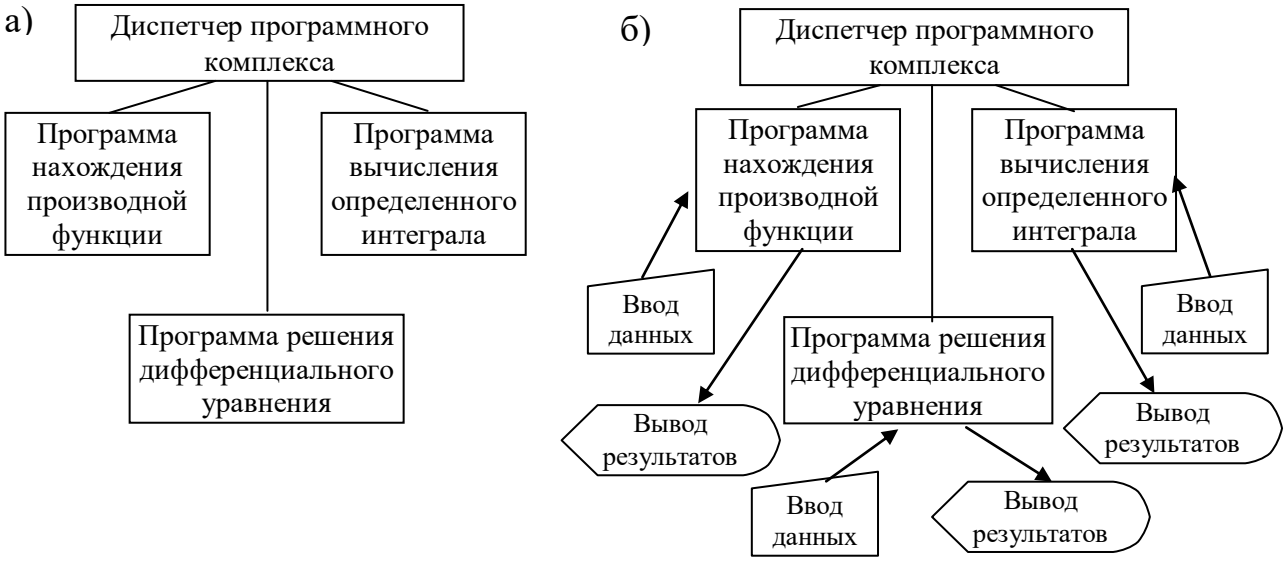


Рисунок А.18 - Пример структурной и функциональной схем программного комплекса

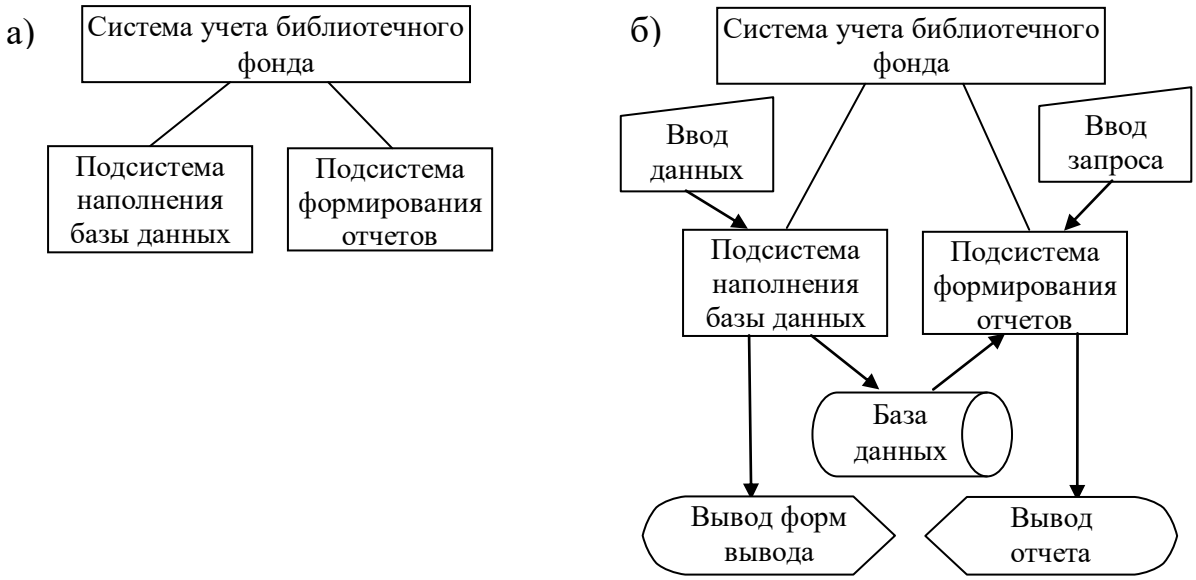


Рисунок А.19 - Пример структурной и функциональной схем программной системы

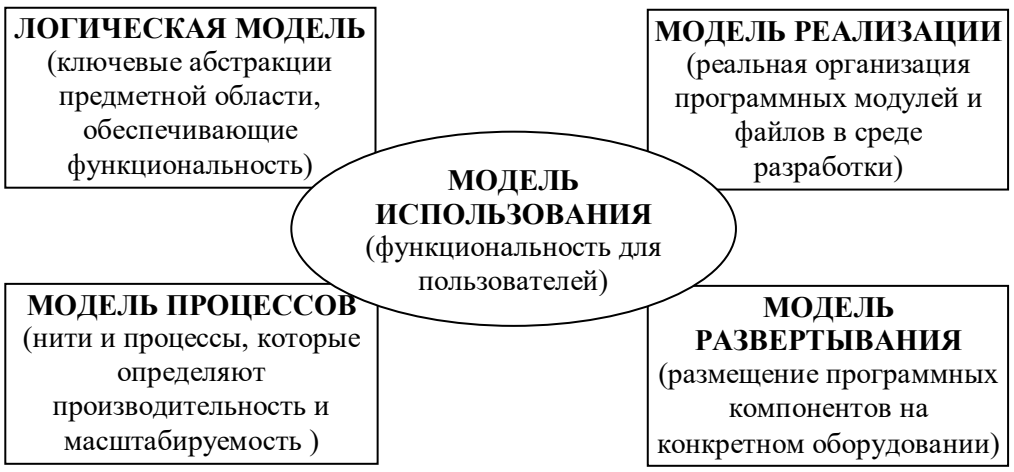


Рисунок А.20 - Полная спецификация разрабатываемого программного обеспечения при объектном подходе (*UML*)

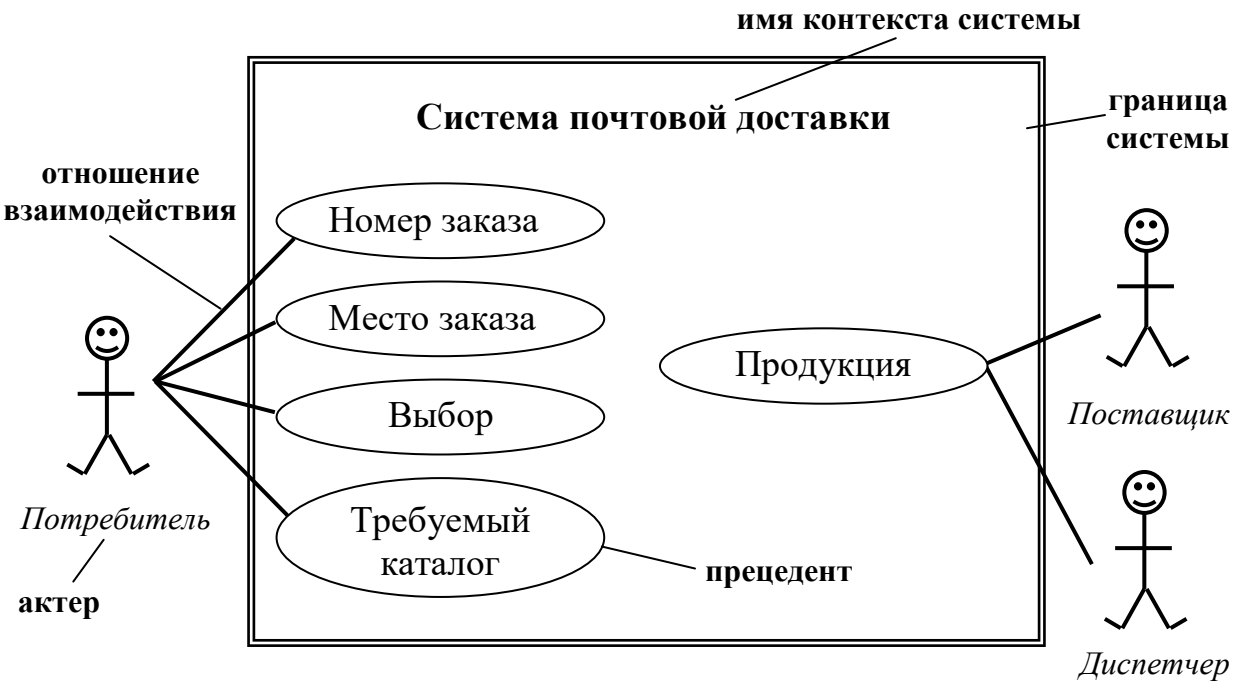


Рисунок А.21 – Пример диаграммы прецедентов

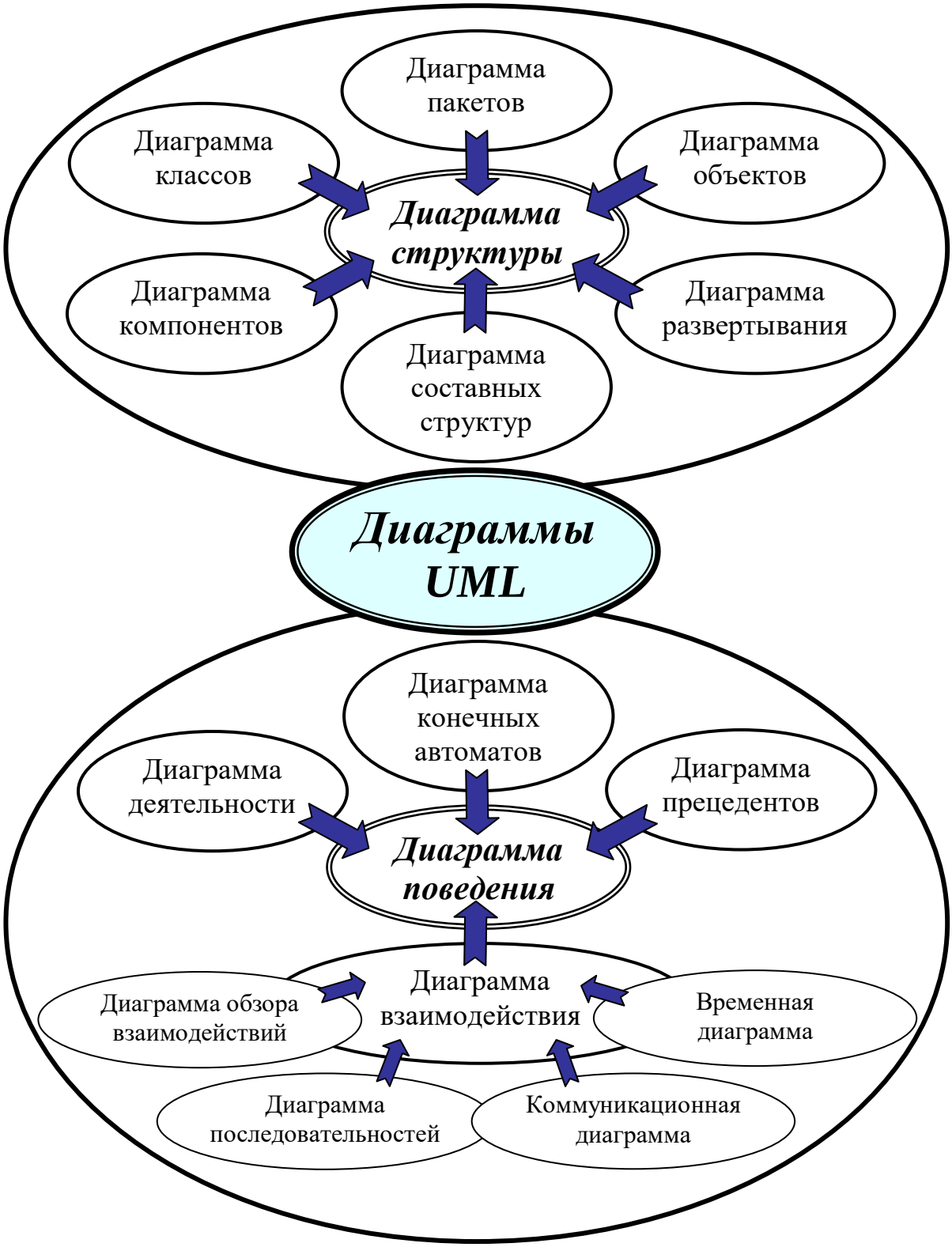


Рисунок А.22– Классификация диаграмм UML

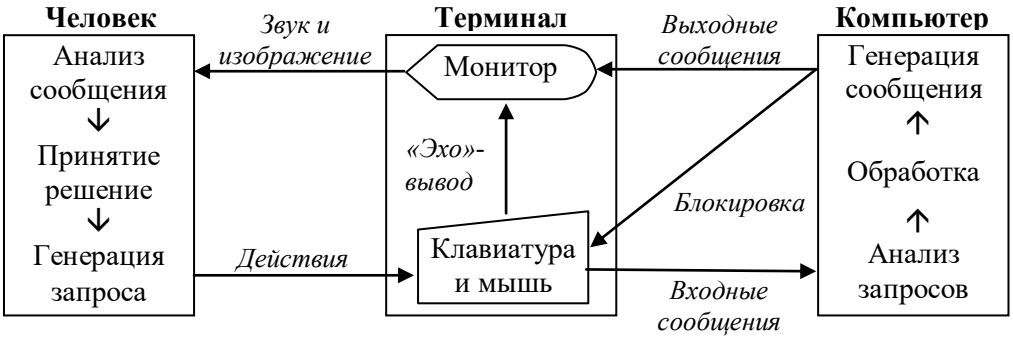


Рисунок А.23 - Организация взаимодействия компьютера и пользователя

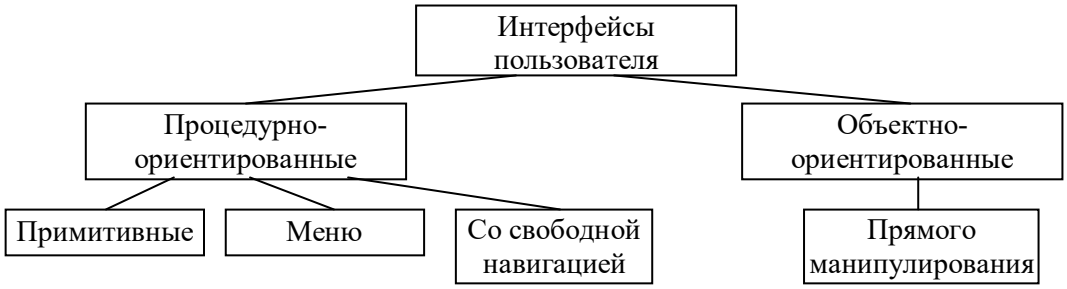


Рисунок А.24 - Типы интерфейсов

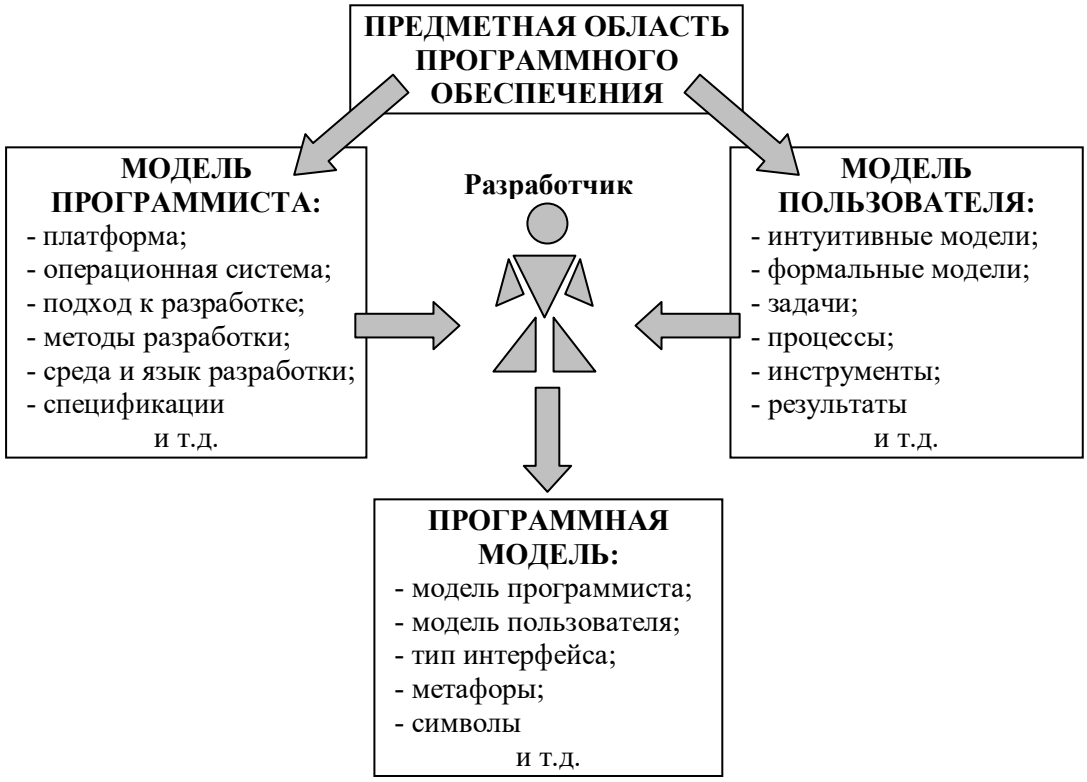


Рисунок А.25 - Процесс разработки пользовательского интерфейса

Продолжение приложения А

Таблица А.1 - Характеристика типов связности модуля

Тип связности	Сопровождаемость	Роль модуля
Функциональная	лучшая	«черный ящик»
Информационная (последовательная)		не совсем «черный ящик»
Коммуникативная		«серый ящик»
Процедурная	худшая	«белый» или «просвечивающий ящик»
Временная		«белый ящик»
Логическая		
Случайная (по совпадению)		

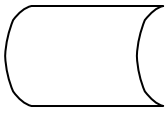
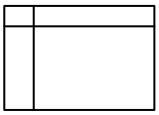
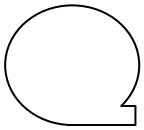
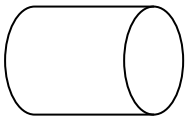
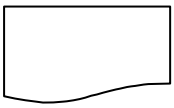
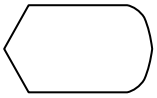
Таблица А.2 – Методы разработки при нисходящем подходе

Метод	Особенности метода	Недостатки
Иерархический	Выполнение разработки строго по уровням. Исключения допускаются при наличии зависимости по данным.	- большое количество достаточно сложных заглушек; - основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.
Операционный	Связывает последовательность разработки модулей с порядком их выполнения при запуске программы.	- порядок выполнения модулей может зависеть от данных; - модули вывода результатов должны разрабатываться одними из первых, чтобы не проектировать сложную заглушку, обеспечивающую вывод результатов при тестировании; - при распределении человеческих ресурсов сложным является начало работ, пока не закончены все модули, находящиеся на так называемом *критическом* пути.

Комбинированный	Учитывает все факторы, влияющие на последовательность разработки: достижимость модуля, зависимость по данным, обеспечение возможности выдачи результатов, готовность вспомогательных модулей, наличие необходимых ресурсов.	- сложные модули должны разрабатываться раньше простых, чтобы выявить в спецификациях возможные ошибки.

Продолжение приложения А

Т а б л и ц а А.3 – Основные обозначения схем данных

Название блока	Обозначение	Назначение блока
Запоминаемые данные		Обозначение таблиц и других структур данных, которые должны быть сохранены без уточнения типа устройства
Оперативное запоминающее устройство		Для обозначения таблиц и других структур данных, хранящихся в оперативной памяти
Запоминающее устройство с последовательной выборкой		Для обозначения таблиц и других структур данных, хранящихся на устройствах с последовательной выборкой (магнитной ленте и т.п.)
Запоминающее устройство с прямым доступом		Для обозначения таблиц и других структур данных, хранящихся на устройствах с прямым доступом (дисках)
Документ		Для обозначения таблиц и других структур данных, выводимых на печатающее устройство
Ручной ввод		Для обозначения ручного ввода данных с клавиатуры
Карта		Для обозначения данных на магнитных или перфорированных картах
Дисплей		Для обозначения данных, выводимых на дисплей

Т а б л и ц а А.4 – Основные отличия пользовательских моделей интерфейсов

Процедурно-ориентированные пользовательские интерфейсы	*Объектно-ориентированные пользовательские интерфейсы*
Обеспечивают пользователей функциями, необходимыми для выполнения задач	Обеспечивают пользователям возможность взаимодействия с объектами

Акцент делается на задачи	Акцент делается на входные данные и результаты
Пиктограммы представляют приложения, окна или операции	Пиктограммы представляют объекты
Содержание папок и справочников отображается с помощью таблиц и списков	Папки и справочники являются визуальными контейнерами объектов

Приложение Б

Образец выполнения технического задания

Техническое задание на разработку системы учета успеваемости студентов

1 Введение

Настоящее техническое задание распространяется на разработку системы учета успеваемости студентов, предназначенной для сбора и хранения информации о ходе сдачи экзаменационной сессии. Предполагается, что использовать данную систему будут сотрудники деканата, декан и его заместители. Во время сессии необходимо получение оперативной информации о ходе ее сдачи студентами, однако выполнение такого контроля вручную требует значительного времени. Автоматизированная система учета успеваемости позволит улучшить качество контроля сдачи сессии со стороны куратора и деканата и обеспечит получение сведений о динамике работы каждого студента, группы и курса в целом. Кроме того, хранение информации о сдаче сессий в течение всего времени обучения позволит осуществлять автоматическую генерацию справок о прослушанных курсах и приложений к диплому выпускника.

2 Основание для разработки

Система разрабатывается на основании приказа декана теплоэнергетического факультета № 5 от 24 сентября 2014 г. и в соответствии с планом мероприятий по совершенствованию учебного процесса на 2014-2015 учебный год.

3 Назначение

Система предназначена для хранения и обработки сведений об успеваемости студентов учебных групп факультета в течение всего срока

обучения. Обработанные сведения об успеваемости студентов могут быть использованы для оценки успеваемости каждого студента, группы, курса и факультета в целом.

4 Требования к программе или программному изделию

4.1 Требования к функциональным характеристикам.

4.1.1 Система должна обеспечивать возможность выполнения следующих функций:

- инициализацию системы (ввод списков групп, перечней дисциплин и т. п.);
- ввод и коррекцию текущей информации о ходе сдачи сессии конкретными студентами;
- хранение информации об успеваемости в течение времени обучения студента;
- получение сведений о текущем состоянии сдачи сессии студентами.

4.1.2 Исходные данные:

- списки студентов учебных групп;
- учебные планы кафедр;
- расписания сессий;
- текущие сведения о сдаче сессии каждым студентом.

4.1.3 Результаты:

- итоги сдачи сессии конкретным студентом;
- итоги сдачи сессии студентами конкретной группы;
- процент успеваемости по всем студентам группы при сдаче конкретного предмета в целом на текущий момент;
- проценты успеваемости по всем группам специальности на текущий момент;
- проценты успеваемости по всем группам курса на текущий момент;
- проценты успеваемости по всем курсам и по факультету на текущий момент;
- список задолжников группы на текущий момент;
- список задолжников курса на текущий момент.

4.2 Требования к надежности.

4.2.1. Предусмотреть контроль вводимой информации.

4.2.2. Предусмотреть блокировку некорректных действий пользователя.

4.2.3. Обеспечить целостность хранимой информации.

4.3 Требования к составу и параметрам технических средств.

4.3.1. Система должна работать на IBM совместимых персональных компьютерах.

4.3.2. Минимальная конфигурация: тип процессора - Pentium и выше; объем оперативного запоминающего устройства - 32 Мб и более.

4.4 Требования к информационной и программной совместимости.

Система должна работать под управлением семейства ОС Windows.

5 Требования к программной документации

5.1 Разрабатываемые программные модули должны быть самодокументированы, т.е. тексты программ должны содержать все необходимые комментарии.

5.2 Программная система должна включать справочную информацию о работе и подсказки пользователю.

5.3 В состав сопровождающей документации должны входить:

5.3.1 Пояснительная записка, содержащая описание разработки.

5.3.2 Руководство системного программиста.

5.3.3 Руководство пользователя.

5.3.4 Графическая часть на трех листах формата А3 (блок-схемы алгоритмов): структурная схема; диаграмма компонентов данных; формы интерфейса пользователя.

6 Этапы разработки

№	Название этапа	Срок	Отчетность
1	Проектирование программного продукта	01.10.2011-01.11.2011	Разработка технического задания и уточнение спецификаций. Принятие принципиальных решений. Алгоритмов решения задачи. Проектирование структурной и функциональной схем системы учета успеваемости.
2	Реализация	01.12.2011-29.02.2012	Описание внутренних форматов, интерфейса и форматов данных базы. Реализация системы на уровне интерфейса и программных модулей.
3	Тестирование и составление документации	01.03.2012-30.04.2012	Результаты тестов. Программная документация.

Список литературы

1 Брукс Фредерик, «Мифический человеко-месяц, или Как создаются программные комплексы», Пер. с англ. - СПб.: Символ-Плюс, 1999.

2 «PMBOK. Руководство к Своду знаний по управлению проектами», 3-е изд., PMI, 2004.

3 Жоголев Е.А. Технологические основы модульного программирования. - М.: Программирование, 1980.

4 Орлов С. Технологии разработки программного обеспечения: Учебник - СПб.: Питер, 2012.

5 Иванова Г.С. Технология программирования. - М.: Изд-во МГТУ им. Н.Э.Баумана, 2002.

6 Майерс Г. Надежность программного обеспечения. - М.: Мир, 1980.

7 Непейвода Н.Н. Стили и методы программирования. Курс лекций. - М.: Интернет-университет информационных технологий, 2005.

8 Немцова Т.И. Программирование на языке высокого уровня. Программирование на языке С++. М.: «Форум», 2012.

9 Канер С., Фолк Д., Нгуен Е.К. Тестирование программного обеспечения. – Киев, ДиаСофт, 2010.

10 Соммервиль И. Инженерия программного обеспечения. - М.: Изд-во Вильямс, 2012.

11 Мандел Т. Разработка пользовательского интерфейса. – М.: ДМК Пресс, 2010.

12 Потопахин В. Искусство алгоритмизации. - М.: «ДМК Пресс», 2011.

13 Терехов А.Н. Технология программирования. – М.: БИНОМ, Интернет-университет информационных технологий, 2006.

14 Иванов Д., Новиков Ф. Моделирование на UML. Учебно-методическое пособие. – СПб.: СПбГУ ИТМО, 2010.

15 Арлоу Д., Нейштадт И. UML2 и Унифицированный процесс. Практический объектно-ориентированный анализ и проектирование. – СПб.: Символ-Плюс, 2007.

16 Гома Х. UML-проектирование систем реального времени параллельных и распределенных приложений. – М.: ДМК Пресс, 2011.

17 Фаулер М. UML. Основы. – СПб: Символ-Плюс, 2005.

18 http://citforum.ru/SE/project/arkhipenkov_lectures/3.shtml#ref.1.1

Сводный план 2014г., поз. 322

Наталья Валерьевна Сябина

ОСНОВЫ ИНЖЕНЕРИИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Конспект лекций
для студентов специальности
5В070200 – Автоматизация и управление

Редактор Л.Т.Сластихина
Специалист по стандартизации Н.К.Молдабекова

Подписано в печать __.__.__.
Тираж 75 экз.
Объем _____ уч.-изд. л.

Формат 60х84 1/16
Бумага типографская №1
Заказ _____. Цена _____тг.

Копировально-множительное бюро
некоммерческого акционерного общества
«Алматинский университет энергетики и связи»
050013 Алматы, Байтурсынова, 126