КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра технологий программирования

О.А. МЕДВЕДЕВА

ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ГАРАНТИИ КАЧЕСТВА

Учебное пособие

Казань – 2018

УДК 004.05
ББК 32.973

Принято на заседании кафедры технологий программирования
Протокол № 6 от 18 апреля 2018 года

Рецензенты:
кандидат физико-математических наук,
заведующий кафедрой технологий программирования КФУ,
доцент **А.И. Еникеев;**
кандидат технических наук,
ст.преп. кафедры технологий программирования КФУ **А.М. Гусенков**

Медведева О.А.
Программное обеспечение гарантии качества / О.А. Медведева. –
Казань: Казан. ун-т, 2018. – 112 с.

В учебном пособии рассматривается задача программного обеспечения гарантии качества проектов в области информационных технологий с точки зрения автоматизированного тестирования. Представлены способы обзора качества, методы тестирования программного обеспечения, концепция управления качеством. В роли инструментального средства для проведения тестирования предложен проект Selenium. Показаны практические примеры автоматизированного тестирования веб-приложений с помощью Selenium WebDriver.

Учебное пособие соответствует требованиям Федерального государственного образовательного стандарта высшего образования последнего поколения. Для студентов, аспирантов и преподавателей, интересующихся вопросами оценки и гарантии качества программного обеспечения.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Инженерия разработки программного обеспечения – это применение систематического подхода, стандартов и количественного измерения характеристик программного обеспечения к разработке, использованию и сопровождению программного обеспечения. В связи с развитием технологий важность программной инженерии постоянно растет, все более востребованными являются методы определения качества программного обеспечения. Сложность процесса разработки и сопровождения программного обеспечения во многом обуславливается особыми требованиями, предъявляемыми к его качеству. Этот фактор обосновывает важность разработки формализованных методов управления качеством программного обеспечения. В настоящий момент используются несколько определений понятия качества программного обеспечения, которые в целом совместимы друг с другом. Обобщая определения на основе стандартов, можно заключить, что качество программного обеспечения − это способность программного продукта соответствовать установленным или предполагаемым потребностям при использовании в заданных условиях. Качество программного обеспечения играет важную роль для всей системы в целом. Так, качество рассматривается как очень важный аспект для разработчиков, пользователей и руководителей проектов. Качество программного обеспечения − величина, отражающая в каком объеме в программный продукт включен набор желаемых функций для повышения эффективности программного продукта в течение жизненного цикла. Для любой системы, использующей программное обеспечение, должны быть разработаны три вида спецификаций, такие как функциональные требования, требования к качеству, требования к ресурсам. Качество программного обеспечения можно разделить на две составляющие, такие как качество процедур разработки и качество программного продукта. Разработка программного обеспечения, связывающая такие элементы как технологии, средства, сотрудники, организация и оборудование, рассматриваются в контексте качества процедур разработки программного продукта. Тем не менее,

качество программного продукта состоит из определенных аспектов, таких как ясность документации и целостность, прослеживаемость проекта, надежность и полнота тестирования основных характеристик. Модель качества обычно определяется набором характеристик и отношений между ними, которые фактически обеспечивают основу как для определения требования качества, так и оценки качества программного обеспечения

Использование различных подходов к тестированию определяется их эффективностью применительно к условиям, определяемым промышленным проектом. В реальных случаях работа группы тестирования планируется так, чтобы разработка тестов начиналась с момента согласования требований к программному продукту и продолжалась параллельно с разработкой дизайна и кода продукта. В результате, к началу системного тестирования создаются тестовые наборы, содержащие тысячи тестов. Большой набор тестов обеспечивает всестороннюю проверку функциональности продукта и гарантирует качество продукта, но пропуск такого количества тестов на этапе системного тестирования представляет проблему. Ее решение находится в области *автоматизации тестирования*, т.е. в автоматизации разработки. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что достижимо при использовании процесса управления качеством программного продукта.

Одним из популярных инструментов тестирования интерфейса веб-приложений является Selenium WebDriver. Selenium − это проект, включающий в себя несколько программных продуктов, в основном написанных на Java, которые позволяют автоматизировать тестирование интерфейса веб-приложений. Selenium открывает тестируемое веб-приложение в браузере и, отправляя различные команды, регистрирует реакцию веб-страницы на данные команды. Selenium WebDriver имеет удобный API, который позволяет получить доступ к его функциям из приложений, написанных на различных языках программирования. WebDriver интегрируется с большим количеством

браузеров, имеет простой набор команд и легкость создания автотестов, в проекте реализована возможность использования шаблонов проектирования PageObject и PageFactory. Selenium WebDriver является мощным инструментом для автоматизации тестирования web-приложений, который позволяет ускорить процесс тестирования, а, следовательно, и сам процесс разработки программного обеспечения. При внедрении автоматизации сокращаются расходы на производство программного обеспечения за счет экономии времени, что приносит дополнительную выгоду для производителей веб-приложений. Предлагается использовать данное программное обеспечение в сфере информационной безопасности, а именно для тестирования приложений на предмет наличия уязвимостей.

ГЛАВА 1. СПОСОБЫ ОБЗОРА КАЧЕСТВА. ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1. Определение качества программного обеспечения

Целью любого проекта в области информационных технологий является соответствие требованиям всех участников проекта. Обеспечение данной цели достигается путем обеспечения качества проекта. В контексте международных стандартов *качество программного обеспечения (Software Quality)* определяется следующим образом:

1. *Качество программного обеспечения* (ПО) − это степень, в которой ПО обладает требуемой комбинацией свойств [1061-1998 IEEE Standard for Software Quality Metrics Methodology].

2. *Качество программного обеспечения* − это совокупность характеристик ПО, относящихся к его способности удовлетворять установленные и предполагаемые потребности [ISO 8402:1994 Quality management and quality assurance].

Наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126. На верхнем уровне выделено шесть основных характеристик качества ПО, каждую из которых определяют набором атрибутов, имеющих соответствующие метрики для последующей оценки (см. рис. 1.1).

Функциональность (Functionality) − определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. Т.е. эта характеристика отвечает за то, что ПО работает исправно и точно, функционально совместимо, соответствует стандартам отрасли и защищено от несанкционированного доступа.

Надежность (Reliability) – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Атрибуты данной характеристики – это

завершенность и целостность всей системы, способность самостоятельно и корректно восстанавливаться после сбоев в работе, отказоустойчивость.

Рис.1.1. Характеристики качества

Удобство использования (Usability) – возможность легкого понимания, изучения, использования и привлекательности ПО для пользователя.

Эффективность (Efficiency) – способность ПО обеспечивать требуемый уровень производительности в соответствие с выделенными ресурсами, временем и другими обозначенными условиями.

Удобство сопровождения (Maintainability) – легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к имеющемуся окружению.

Портативность (Portability) – характеризует ПО с точки зрения легкости его переноса из одного окружения (software/hardware) в другое.

Рассмотрим следующие основные понятия и определения.

Обеспечение качества (Quality Assurance, QA) − это совокупность мероприятий, охватывающих все технологические этапы разработки, выпуска и эксплуатации программного обеспечения (ПО) информационных систем,

предпринимаемых на разных стадиях жизненного цикла ПО, для обеспечения требуемого уровня качества выпускаемого продукта.

Контроль качества (Quality Control, QC) − это совокупность действий, проводимых над продуктом в процессе разработки, для получения информации о его актуальном состоянии в разрезах: "готовность продукта к выпуску", "соответствие зафиксированным требованиям", "соответствие заявленному уровню качества продукта".

Тестирование программного обеспечения (Software Testing) − это одна из техник контроля качества, включающая в себя активности по планированию работ (*Test Management*), проектированию тестов (*Test Design*), выполнению тестирования (*Test Execution*) и анализу полученных результатов (*Test Analysis*).

Верификация (verification) − это процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа. Т.е. выполняются ли наши цели, сроки, задачи по разработке проекта, определенные в начале текущей фазы.

Валидация (validation) − это определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе.

2. Способы обзора качества

Верификация и *валидация*, как методы, обеспечивают соответственно проверку и *анализ* правильности выполнения заданных функций и соответствия ПО требованиям заказчика, а также заданным спецификациям. Они представлены в международных стандартах как самостоятельные процессы жизненного цикла (ЖЦ) и используются, начиная от этапа анализа требований и заканчивая проверкой правильности функционирования программного кода на заключительном этапе, а именно, тестировании.

Для этих процессов определены цели, задачи и действия по проверке правильности создаваемого продукта (рабочие, промежуточные продукты) на этапах ЖЦ. Рассмотрим их трактовку в стандартном представлении.

Цель процесса верификации − убедиться, что каждый *программный продукт* проекта отражает согласованные требования к их реализации. Этот процесс основывается:

− на стратегии и критериях верификации применительно ко всем рабочим программным продуктам;

− на выполнении действий стандарта по верификации;

− на *устранении недостатков*, обнаруженных в программных (рабочих и промежуточных) продуктах;

− на согласовании результатов верификации с заказчиком.

Процесс верификации может проводиться исполнителем программы или другим сотрудником той же организации, или сотрудником другой организации, например, заказчиком. Этот процесс включает в себя действия по его внедрению и выполнению.

Внедрение процесса заключается в определении критических элементов (процессов и программных продуктов), которые должны подвергаться верификации, в выборе исполнителя верификации, инструментальных средств поддержки процесса верификации, в составлении плана верификации и его утверждении. В процессе верификации выполняются задачи проверки условий: контракта, процесса, требований, интеграции, проекта, кода и документации. При верификации согласно плану и требований заказчика проверяется правильность выполнения функций системы, интерфейсов и взаимосвязей компонентов, а также доступа к данным и к средствам защиты.

Цель процесса валидации − убедиться, что специфические требования для программного продукта выполнены, и осуществляется это с помощью:

− разработанной стратегии и критериев валидации для всех *рабочих продуктов*;

− оговоренных действий по проведению валидации;

− демонстрации соответствия разработанных программных продуктов требованиям заказчика и правилам их использования;

− согласования с заказчиком полученных результатов валидации.

Процесс валидации может проводиться самим исполнителем или другим лицом, например, заказчиком, осуществляющим действия по внедрению и проведению этого процесса по плану, в котором отражены элементы и задачи проверки. При этом используются методы, инструментальные средства и процедуры выполнения задач процесса для установления соответствия тестовых требований и особенностей использования программных продуктов проекта.

На других процессах ЖЦ выполняются дополнительные действия:

− проверка и контроль проектных решений с помощью методик и процедур просмотра хода разработки;

− обращение к *CASE-системам*, которые содержат процедуры проверки требований к продукту;

− просмотры и инспекции промежуточных результатов на соответствие их требованиям для подтверждения того, что ПО имеет корректную реализацию требований и удовлетворяет условиям выполнения системы.

Таким образом, основные задачи процессов верификации и валидации состоят в том, чтобы *проверить и подтвердить*, что итоговый программный продукт отвечает назначению и удовлетворяет требованиям заказчика.

Верификация и валидация основаны на планировании их как процессов, так и проверки для наиболее критичных элементов проекта: компонент, интерфейсов (программных, технических и информационных), взаимодействий объектов (протоколов и сообщений), передач данных между компонентами и их защиты, а также оставленных тестов и тестовых процедур.

После проверки отдельных компонентов системы проводятся их интеграция и повторная верификация и валидация интегрированной системы, создается комплект документации, отображающий правильность проверки формирования требований, результатов инспекций и тестирования.

3. Тестирование программного обеспечения

Обеспечение качества отвечает за весь процесс разработки, поэтому должно быть интегрировано во все этапы разработки: от описания проекта до

тестирования, релиза и даже пост-релизного обслуживания. Специалисты QA создают и реализуют различные тактики для повышения качества на всех стадиях производства: подготовка и установление стандартов, анализ качества, выбор инструментов, предотвращение появления ошибок и постоянное усовершенствование процесса.

Задача контроля качества – гарантировать соответствие требованиям (поиск ошибок и их устранение). Контроль качества ориентирован на проверку продукта, включает в себя многие процессы, такие как анализ кода, технические обзоры, анализ дизайна, тестирование и пр.

Тестирование – это проверка результатов работы на соответствие требованиям.

Взаимосвязь этапов обеспечения качества, контроля качества и тестирования показана на рисунке 1.2.

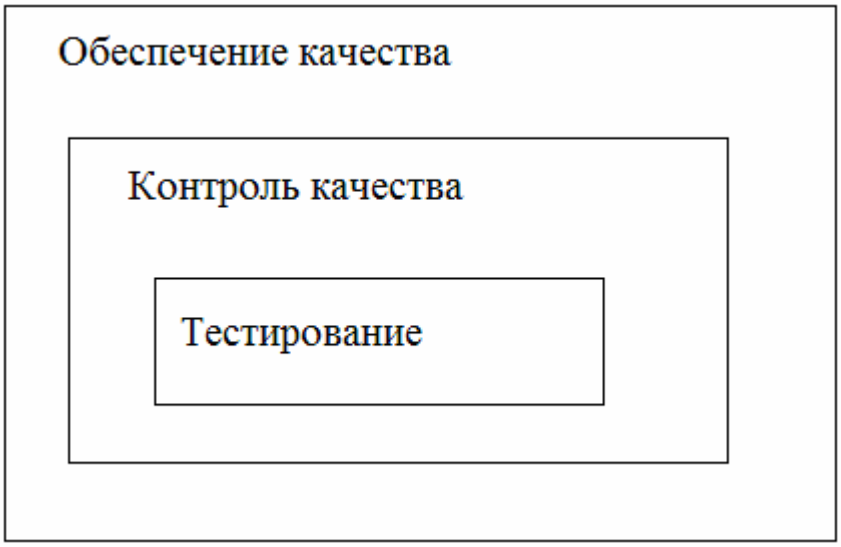


Рис. 1.2.

Тестирование можно рассматривать, как процесс семантической отладки (проверки) программы, заключающийся в исполнении последовательности различных наборов контрольных тестов, для которых заранее известен результат. Т.е. тестирование предполагает выполнение программы и получение конкретных результатов выполнения тестов.

Тесты подбираются так, чтобы они охватывали как можно больше типов ситуаций алгоритма программы. Менее жесткое требование − выполнение хотя бы один раз каждой ветви программы.

Исторически первым видом тестирования была отладка.

Отладка − это проверка описания программного объекта на языке программирования с целью обнаружения в нем ошибок и последующее их устранение. Ошибки обнаруживаются компиляторами при их синтаксическом контроле. После этого проводится верификация по проверке правильности кода и валидация по проверке соответствия продукта заданным требованиям.

Целью тестирования является проверка работы реализованных функций в соответствии с их спецификацией. На основе внешних спецификаций функций и проектной информации на процессах ЖЦ создаются функциональные тесты, с помощью которых проводится тестирование с учетом требований, сформулированных на этапе анализа предметной области.

4. Измерение соответствия теста

Покрытие, основанное на спецификации или на требованиях (*Specification-Based Coverage or Requirements-based Test Coverage*). Этот критерий оценивает степень покрытия, принимая во внимание требования Заказчика или системные спецификации. Основой может быть, например, таблица требований, use-case модель и диаграмма состояний-переходов. Набор тестов должен покрывать все или конкретно определенные функциональные требования. На практике это чаще всего реализуется следующим образом: Заказчик (или системный аналитик) составляет набор требований, которые могут быть переведены в тестовые сценарии. После чего эти сценарии могут быть проверены на правильность и полноту.

Таким образом, данный критерий показывает в процентном отношении количество покрытых тестами требований. Чаще всего данный критерий используется при тестировании методом «черного ящика».

Покрытие, основанное на коде (*Code-Based Coverage*) имеет отношение к потоку управления и потоку данных программы. Чаще всего данный критерий

используется при тестировании методом «белого ящика». Основные критерии покрытия тестирования кода следующие:

- *Покрытие строк* (*Line Coverage*) – мера измерения покрытия кода, указывающая процентное отношение строк программы, затронутых тестами, к общему числу строк. Это очень неточная метрика, потому что даже стопроцентное покрытие по ней пропускает много ошибок.
- *Покрытие ветвей* (*Branch Coverage*). Это мера измерения покрытия кода указывает в процентном отношении, сколько ветвей потока управления было протестировано во время теста. Она надежнее предыдущей метрики, но снова стопроцентное покрытие не гарантирует отсутствие ошибок.
- *Покрытие путей* (*Path Coverage*). Эта единица измерения характеризует процент всевозможных путей (и/или комбинаций ветвей), которые покрываются тестами. Однако, даже не смотря на 100-процентное покрытие (достичь которого практически нереально в коммерческих системах) все еще могут присутствовать скрытые ошибки.

Метрики и критерии тестирования определяются в стратегии тестирования наряду с остальными составляющими процесса.

Контрольные вопросы

1. Дайте определение качества программного обеспечения.
2. Назовите основные характеристики качества программного обеспечения.
3. Приведите примеры способов обзора качества программного обеспечения.
4. В чем отличие методов верификации и валидации?
5. Какова цель тестирования программного обеспечения?
6. В чем заключается измерение соответствия теста?

ГЛАВА 2. МЕТОДЫ ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1. Методы тестирования на основе стратегии черного ящика

Целью *функционального тестирования* является обнаружение несоответствий между реальным поведением реализованных функций и ожидаемым поведением в соответствии со спецификацией и исходными требованиями. Функциональные тесты должны охватывать все реализованные функции с учетом наиболее вероятных типов ошибок. Тестовые сценарии, объединяющие отдельные тесты, ориентированы на проверку качества решения функциональных задач.

Функциональные тесты создаются по внешним спецификациям функций, проектной информации и по тексту на языке программирования, относятся к функциональным его характеристикам и применяются на этапе комплексного тестирования и испытаний для определения полноты реализации функциональных задач и их соответствия исходным требованиям.

В задачи *функционального тестирования* входят:

- идентификация множества функциональных требований;
- идентификация внешних функций и построение последовательностей функций в соответствии с их использованием в ПО;
- идентификация множества входных данных каждой функции и определение областей их изменения;
- построение тестовых наборов и сценариев тестирования функций;
- выявление и представление всех функциональных требований с помощью тестовых наборов и проведение тестирования ошибок в программе и при взаимодействии со средой.

Предпосылки *функционального тестирования*:

- корректное оформление требований и ограничений к качеству ПО;
- корректное описание модели функционирования ПО в среде эксплуатации у заказчика;
- адекватность модели ПО заданному классу.

Методы *функционального тестирования* подразделяются на статические и динамические.

Статические методы используются при проведении инспекций и рассмотрении спецификаций компонентов без их выполнения. Техника статического анализа заключается в обзоре и анализе структуры программ, а также в доказательстве их правильности. Статический анализ направлен на анализ документов, разработанных на всех этапах жизненного цикла программного обеспечения, и заключается в инспекции исходного кода и сквозного контроля программы.

Инспекция ПО − это статическая проверка соответствия программы заданным спецификациями, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программ) на всех этапах жизненного цикла. Просмотры и инспекции результатов проектирования и соответствия их требованиям заказчика обеспечивают более высокое качество создаваемых программ.

На начальном этапе проектирования инспекция предполагает проверку полноты, целостности, однозначности, непротиворечивости и совместимости документов с исходными требованиями к программному обеспечению. На этапе реализации под *инспекцией* понимается анализ текстов программ на соблюдение требований стандартов и принятых руководящих документов технологии программирования.

Эти приемы позволяют на более ранних этапах проектирования обнаружить ошибки или дефекты путем многократного просмотра исходных кодов. Символьное тестирование применяется для проверки отдельных участков программы на входных символьных значениях.

Кроме того, разрабатывается множество новых способов автоматизации символьного выполнения программ. Например, автоматизированное средство статического контроля для языков ориентированной разработки, инструменты

автоматизации *доказательства корректности* и автоматизированный аппарат сетей Петри.

Динамические методы тестирования используются в процессе выполнения программ. Они базируются на графе, связывающем причины ошибок с ожидаемыми реакциями на эти ошибки. В процессе тестирования накапливается информация об ошибках, которая используется при оценке надежности и качества ПО.

Динамическое тестирование ориентировано на проверку корректности ПО на множестве тестов, в целях проверки и сбора данных на этапах ЖЦ и проведения измерения отдельных показателей (число отказов, сбоев) тестирования для оценки характеристик качества, указанных в требованиях, посредством выполнения системы на ЭВМ. Тестирование основывается на систематических, статистических, (вероятностных) и имитационных методах.

Дадим краткую их характеристику. Систематические методы тестирования делятся на методы, в которых программы рассматриваются как "черный ящик" (используется информация о решаемой задаче), и методы, в которых программа рассматривается как "белый ящик" (используется структура программы). Этот вид называют тестированием с управлением по данным или управлением по входу-выходу. Цель − выяснение обстоятельств, при которых поведение программы не соответствует ее спецификации. При этом количество обнаруженных ошибок в программе является критерием качества входного тестирования.

Цель динамического тестирования программ по принципу "черного ящика" − выявление одним тестом максимального числа ошибок с использованием небольшого подмножества возможных входных данных.

Методы "черного ящика" обеспечивают:

- эквивалентное разбиение;
- анализ граничных значений;

• применение функциональных диаграмм, которые в соединении с реверсивным анализом дают достаточно полную информацию о функционировании тестируемой программы.

Методы тестирования по принципу "черного ящика" используются для тестирования функций, реализованных в программе, путем проверки несоответствия между реальным поведением функций и ожидаемым поведением с учетом спецификаций требований. Во время подготовки к этому тестированию строятся таблицы условий, причинно-следственные графы и области разбивки. Кроме того, подготавливаются тестовые наборы, учитывающие параметры и условия среды, которые влияют на поведение функций. Для каждого условия определяется множество значений и ограничений предикатов, которые тестируются.

2. Комбинаторное тестирование

Комбинаторные методы построения тестов основаны на разделении каждого тестового воздействия на ряд элементов и построении тестов как всевозможных комбинаций полученных элементов, объединяемых по определенным правилам. Комбинаторные методы дают более высокую полноту покрытия, чем вероятностные, и при этом требуют ненамного больше ресурсов. Кроме того, они хорошо автоматизируются. Однако с помощью комбинаторных методов трудно найти ошибки в очень специфических ситуациях, требующих учета многих факторов, а трудозатраты на их применение при учете возрастающего числа факторов растут гораздо быстрее. Одним из примеров комбинаторных методов является тестирование по разбиениям на категории (*category partition testing*). В рамках этого подхода для построения тестов выполняются следующие действия:

1) выделяется набор операций тестируемой системы, обращения к которым должны производиться в тестах;

2) для каждой тестируемой операции анализируются требования к ней и на основе этого анализа выделяются, дополнительно к ее параметрам,

некоторые факторы (внешние условия или свойства внутреннего состояния системы), от которых может зависеть ее поведение;

3) возможные значения каждого параметра операции или фактора, влияющего на ее поведения, классифицируются, т.е. разбиваются на конечное множество категорий. Категории выделяются таким образом, чтобы изменение значения параметра или фактора в рамках одной категории слабо изменяло требования к работе операции;

4) определяются зависимости между полученными категориями, взаимосвязи между значениями различных параметров и факторов, а также недопустимые комбинации их значений;

5) тестовые ситуации строятся как возможные комбинации категорий значений параметров и факторов. Для каждой такой комбинации определяются соответствующие конкретные значения параметров и способ достижения соответствующих значений факторов (тестовые последовательности и изменения внешних условий).

3. Метод дерева классификации

Достаточно сильно похож на описанную выше технику и метод построения тестов на основе дерева классификации (*classification tree method*). Он используется при наличии большого количества факторов, влияющих на поведение тестируемой системы. Сначала выделяется набор наиболее заметных факторов, влияние которых на поведение тестируемой системы достаточно сильно. Такие факторы называются в рамках этого метода аспектами. Для каждого из этих факторов пытаются определить разбиение его возможных значений на группы, в рамках которых требования к поведению системы меняются слабо. При этом могут быть выявлены другие факторы, чье влияние на систему становится существенным, если один из базовых аспектов зафиксирован. Базовые аспекты образуют вершины дерева классификации, непосредственно связанные с его корнем. Их классы и вторичные аспекты привязываются к базовым аспектам, и т.д., пока все существенные факторы не будут найдены и классифицированы. После построения дерева классификации

необходимо определить зависимости между выделенными классами значений аспектов. Тесты строятся как возможные комбинации классов значений аспектов, соответствующих листовым вершинам дерева.

4. Методы тестирования на основе стратегии белого ящика

Метод "белого ящика" позволяет исследовать внутреннюю структуру программы, причем обнаружение всех ошибок в программе является критерием исчерпывающего тестирования маршрутов потоков (графа) передач управления, среди которых рассматриваются:

1) критерий покрытия операторов − набор тестов в совокупности должен обеспечить прохождение каждого оператора не менее одного раза;

2) критерий тестирования ветвей (известный как покрытие решений или покрытие переходов) − набор тестов в совокупности должен обеспечить прохождение каждой ветви и выхода, по крайней мере, один раз.

Критерий (2) соответствует простому структурному тесту и наиболее распространен на практике. Для удовлетворения этого критерия необходимо построить систему путей, содержащую все ветви программы. Нахождение такого оптимального покрытия в некоторых случаях осуществляется просто, а в других является более сложной задачей.

Тестирование по принципу "белого ящика" ориентировано на проверку прохождения всех путей программ посредством применения путевого и имитационного тестирования.

Путевое тестирование применяется на уровне модулей и графовой *модели программы* путем выбора тестовых ситуаций, подготовки данных и включает тестирование следующих элементов:

- операторов, которые должны быть выполнены хотя бы один раз, без учета ошибок, которые могут остаться в программе из-за большого количества логических путей и необходимости прохождения подмножеств этих путей;
- путей по заданному графу потоков управления для выявления разных маршрутов передачи управления с помощью путевых предикатов, для вычисления которого создается набор тестовых данных, гарантирующих

прохождение всех путей. Однако все пути протестировать бывает невозможно, поэтому остаются не выявленные ошибки, которые могут проявиться в процессе эксплуатации;

- блоков, разделяющих программы на отдельные части блоки, которые выполняются один раз или многократно при нахождении путей в программе, включающих совокупность блоков реализации одной функции либо нахождения входного множества данных, которое будет использоваться для выполнения указанного пути.

"Белый ящик" базируется на структуре программы, в случае "черного ящика", о структуре программы ничего неизвестно. Для выполнения тестирования с помощью этих "ящиков" известными считаются выполняемые функции, входы (входные данные) и выходы (выходные данные), а также логика обработки, представленные в документации.

5. Выборочное и поисковое тестирование

Выборочное тестирование − разработка тестов методом черного ящика, в котором тестовые сценарии выбираются для соответствия функциональному разрезу, обычно с помощью алгоритма псевдо-случайного выбора. Этот метод может использоваться для тестирования таких нефункциональных атрибутов, как надежность и производительность.

Поисковое тестирование (*exploratory testing*) − неформализованное тестирование (*ad-hoc testing*), когда специалист по тестированию, действуя с учётом руководства по тестированию (*test charter*), активно вносит изменения в тестовые примеры (*test case*) и использует информацию, накопленную при тестировании, для создания новых, улучшенных тестов.

Свободное тестирование (*ad-hoc testing*) – это вид тестирования, который выполняется без подготовки к тестированию продукта, без определения ожидаемых результатов, проектирования тестовых сценариев. Это неформальное, импровизационное тестирование.

6. Инфраструктура процесса тестирования программного обеспечения

Под инфраструктурой процесса тестирования понимается:

- выделение объектов тестирования;
- проведение классификации ошибок для рассматриваемого класса тестируемых программ;
- подготовка тестов, их выполнение и поиск разного рода ошибок и отказов в компонентах и в системе в целом;
- служба проведения и управление процессом тестирования;
- анализ результатов тестирования.

На современном этапе развития средств поддержки разработки ПО (CASE-технологии, объектно-ориентированные методы и средства проектирования моделей и программ) проводится такое проектирование, при котором ПО защищается от наиболее типичных ошибок и тем самым предотвращается появление программных дефектов.

Фирма IBM разработала подход к классификации ошибок, называемый ортогональной классификацией дефектов (таблица 2.1). Подход предусматривает разбиение ошибок по категориям с соответствующей ответственностью разработчиков за них.

Ортогональность схемы классификации заключается в том, что любой ее термин принадлежит только одной категории.

Многие специалисты сравнивают тестирование системы с созданием новой системы, в которой аналитики отражают потребности и цели заказчика, работая совместно с проектировщиками и добиваясь реализации идей и принципов работы системы.

Проектировщики системы сообщают команде тестировщиков проектные цели, чтобы они знали декомпозицию системы на подсистемы и ее функции, а также принципы их работы. После проектирования тестов и тестовых покрытий, команда тестировщиков проводит анализ возможностей системы.

Таблица 2.1. Ортогональная классификация дефектов IBM	
Контекст ошибки	**Классификация дефектов**
Функция	Ошибки интерфейсов конечных пользователей ПО, вызванные аппаратурой или связаны с глобальными структурами данных
Интерфейс	Ошибки во взаимодействии с другими компонентами, в вызовах, макросах, *управляющих блоках* или в списке параметров
Логика	Ошибки в программной логике, неохваченной валидацией, а также в использовании значений переменных
Присваивание	Ошибки в структуре данных или в инициализации переменных отдельных частей программы
Зацикливание	Ошибки, вызванные ресурсом времени, реальным временем или разделением времени
Среда	Ошибки в репозитории, в управлении изменениями или в контролируемых версиях проекта
Алгоритм	Ошибки, связанные с обеспечением эффективности, корректности алгоритмов или структур данных системы
Документация	Ошибки в записях документов сопровождения или в публикациях

7. План тестирования

Для проведения тестирования создается план (*Test Plan*), в котором описываются стратегии, ресурсы и график тестирования отдельных компонентов и системы в целом. В плане отмечаются работы для разных членов команды, которые выполняют определенные роли в этом процессе. План включает также определение роли тестов в каждом процессе, степень покрытия программы тестами и процент тестов, которые выполняются со специальными данными.

Тестовые инженеры создают множество тестовых сценариев (*Test Cases*), каждый из которых проверяет результат взаимодействия между актором и системой на основе пред- и постусловий использования таких сценариев. Сценарии в основном относятся к тестированию по типу белого "ящика" и ориентированы на проверку структуры и операций интеграции компонентов системы.

Для проведения тестирования тестовые инженеры предлагают процедуры тестирования (*Test Procedures*), включающие валидацию объектов и верификацию тестовых сценариев в соответствии с планом графикам. Области ответственности инженера тестировщика представлена на рисунке 2.1. Оценка тестов (*Test Evaluation*) заключается в оценке результатов тестирования, степени покрытия программ сценариями и статуса полученных ошибок.

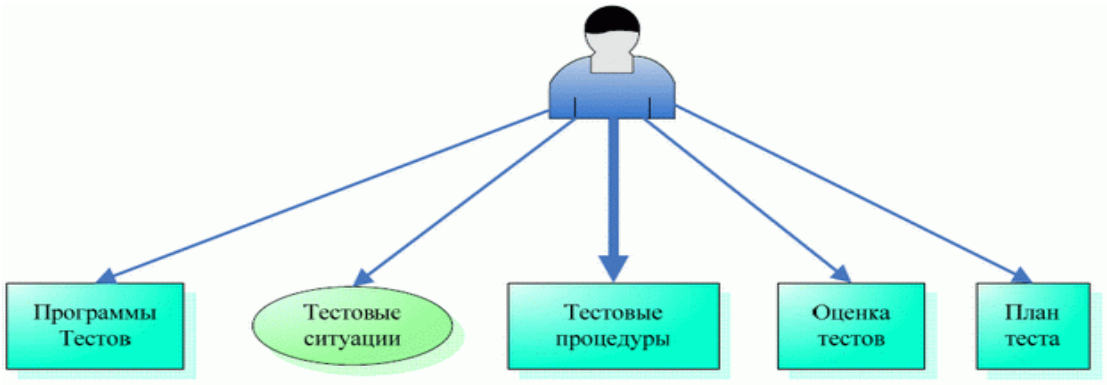


Рис. 2.1. Ответственности инженера тестировщика

Тестировщик интегрированной системы проводит тестирование интерфейсов и дает оценку результатов выполнения соответствующих тестов с помощью создаваемых им системных тестов, выполняет анализ результатов тестирования, проведенного с отдельными элементами системы. При выполнении системных тестов, как правило, находятся дефекты, как результат глубоко скрытых погрешностей в программах, обнаруживаемых при длительной прогонке системы на тестовых данных и сценариях.

Все способы тестирования ПС объединяются базой данных, где помещаются результаты тестирования системы. В ней содержатся все компоненты, тестовые контрольные данные, результаты тестирования и информация о документировании процесса тестирования.

База данных проекта поддерживается специальными инструментальными средствами типа CASE, которые обеспечивают ведение анализа ПО, сборку данных об их объектах, потоках данных и тому подобное. База данных проекта хранит также начальные и эталонные данные, которые используются для сопоставления данных, накопленных в базе, с данными, которые получены в процессе тестирования системы.

Контрольные вопросы

1. Дайте определение функциональному тестированию.
2. Что называется инспекцией программного обеспечения?
3. Дайте характеристику методу тестирования на основе черного ящика.
4. Дайте характеристику методу тестирования на основе белого ящика.
5. В каких случаях применяется комбинаторное тестирование?
6. Когда используется метод дерева классификации?
7. Что такое выборочное тестирование?
8. В чем смысл поискового тестирования?
9. Для чего необходим план тестирования?

ГЛАВА 3. УПРАВЛЕНИЕ КАЧЕСТВОМ ПРОЕКТА

1. Стандарты управления качеством IT − проектов

Управление качеством (в рамках управления IT − проектом) − это система методов, средств и видов деятельности, направленных на выполнение требований участников проекта к качеству самого проекта и его продукции. Как самостоятельная область профессиональной деятельности, *управление качеством* имеет собственные стандарты, к которым относятся:

- ISO9000 (в России ГОСТ Р ИСО 9001-96) − стандарт для обеспечения качества результатов проектов;
- ISO10006 − стандарт регламентирует качество осуществления процессов управления проектами.

Стандарты ISO 9000 имеют самое широкое распространение в мире стандартов по системам качества. С 1 января 2002 года введена новая редакция стандартов ИСО 9000:2000:

- ИСО 9001. Система менеджмента качества. Требования;

- ИСО 9004. Система менеджмента качества. Руководство для улучшения характеристик СМК для повышения эффективности предприятия.

Основные принципы управления качеством по стандартам серии ISO 9000:2000:

- ориентация деятельности Компании на клиента;
- управляемость и наблюдаемость всех процессов Компании;
- вовлечение и *мотивация персонала*;
- процессное представление всех видов деятельности;
- системный подход к управлению;
- непрерывное совершенствование системы менеджмента качества (СМК);
- достоверность информации для управленческих решений;
- взаимовыгодные отношения с поставщиками.

Стандарт ISO10006 имеет название “Менеджмент качества. Руководство качеством при управлении проектами”. Основные принципы управления качеством по стандартам серии ISO 10006:1997:

- ориентация деятельности Компании на клиента;
- ответственность руководства за создание благоприятной среды в отношении качества и непрерывное совершенствование СМК;
- представление проекта как набора запланированных и взаимоувязанных процессов;
- сфокусированность на качестве продуктов и услуг как на необходимом условии соответствия целям проекта;
- процессное представление всех видов деятельности;
- системный подход к управлению проектом в целом.

2. Процессы управления качеством

Основными процессами управления проектами по стандарту ISO 10006:1997 являются процессы определения стратегии, процессы управления взаимосвязями в проекте, процессы управления реализацией проекта, включающие управление предметной областью, управление сроками,

управление затратами, *управление ресурсами*, *управление персоналом*, управление информацией, *управление рисками*, управление материально-техническим снабжением.

Управление качеством проекта осуществляется на протяжении всего жизненного *цикла* проекта. На рисунке 3.1 представлены стадии управления качеством проекта.

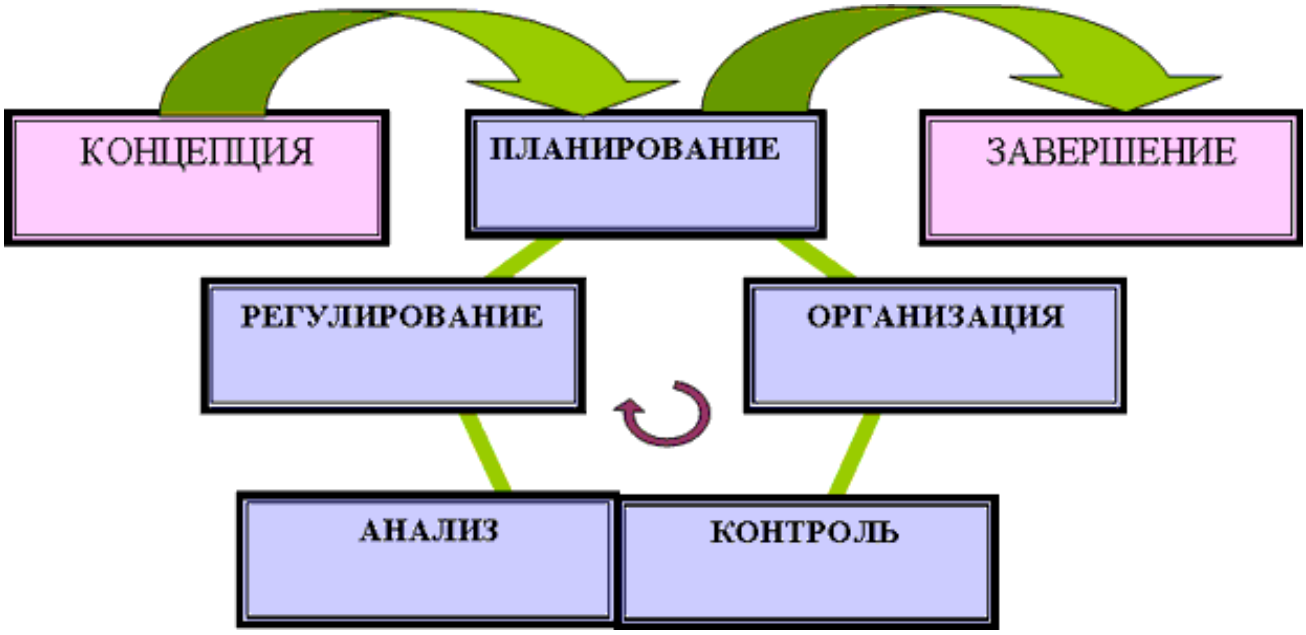


Рис. 3.1. Стадии процесса управления качеством проекта

Стадия Концепция. На этой стадии определяется политика и стратегия для обеспечения качества разрабатываемого продукта, удовлетворяющего ожидаемым запросам потребителя. "Концепция" имеет следующие разделы:

- политика и стратегия качества;
- общие требования и принципы обеспечения качества;
- стандарты, нормы и правила;
- интеграция функций обеспечения качества;
- требования к системе управления качеством.

Стадия планирования. На стадии планирования качества определяются стандарты, которые следует использовать, чтобы содержание проекта оправдывало ожидания участников проекта. Планирование качества включает как идентификацию этих стандартов, так и поиск путей их реализации. Ниже перечислены основные задачи стадии планирования:

- определение показателей оценки качества;
- определение технических спецификаций;
- описание процедур управления качеством;
- составление списка объектов контроля;
- выбор методов и средств оценки качества;
- описание связей с другими процессами;
- разработка *плана управления качеством*.

Стадия организации. Стадии организации контроля качества предполагает создание необходимых и достаточных организационных, технических, финансовых и др. условий для обеспечения выполнения требований к качеству проекта и продукции проекта и возможностей их удовлетворения.

Стадия контроля. *Контроль* качества заключается в определении соответствия результатов проекта стандартам качества и причин нарушения такого соответствия.

Стадии регулирования и анализа. Стадия осуществления контроля качества предполагает регулярную проверку хода реализации проекта в целях установления фактического соответствия определенным ранее требованиям.

- Сравнение фактических результатов проекта с требованиями.
- Анализ прогресса качества в проекте на протяжении его жизненного цикла.
- Формирование списка отклонений.
- Корректирующие действия.
- Документирование изменений.

Стадия завершения. На стадии завершения выполняются сводная оценка качества результатов проекта, завершающая приемка, составление списка претензий по качеству, *разрешение конфликтов* и споров, оформление документации, *анализ* опыта и полученных уроков по управлению качеством.

3. Обеспечение качества проекта

Основными *процессами обеспечения* качества проекта являются планирование качества, его обеспечение и *контроль*. *Связь* этих процессов, их входы и выходы представлены на рисунке 3.2.

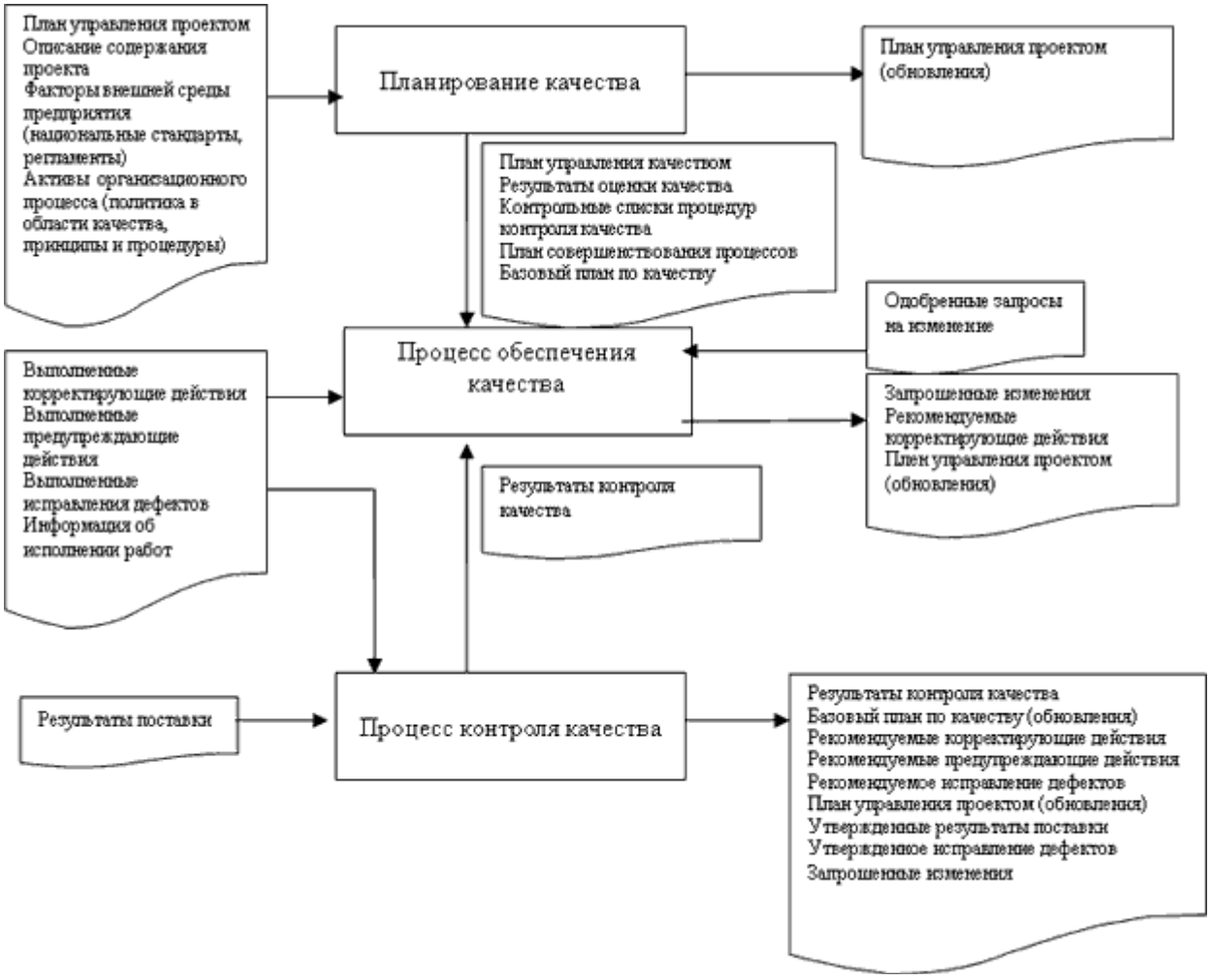


Рис. 3.2. Связь процессов управления качеством проекта

Планирование качества − процесс определения того, какие из стандартов качества относятся к данному проекту и как их удовлетворить.

Планирование качества осуществляется как часть планирования проекта и выполняется совместно руководителем проекта, архитектором проекта и ответственным за качество проекта. В *план управления качеством* включаются работы, выполнение которых обеспечивает качество результатов проекта. Одной из главных составляющих *плана управления качеством* IT-проектов, является план проведения тестирования.

План по качеству должен определять, как в проекте будет обеспечено качество выполнения *работ* с позиции организационной структуры, ресурсов, методического обеспечения. На стадии планирования качества рекомендуется разработать документы, регламентирующие действия по контролю качества управления проектом (форму отчетности по выполнению проекта, анкеты мониторинга проекта) и процедуры управления качеством, например *контроль* качества результатов проекта, *контроль* качества документов проекта, утверждение документов проекта, подготовка и проведение контроля проекта. Для контроля качества документов проекта в плане по качеству следует определить *список* лиц, согласующих и утверждающих каждый документ проекта, сроки и форму их согласования.

На IT-проектах вводится множество специфических терминов, поэтому в план контроля качества проекта необходимо включать разработку и согласование *глоссария проекта*.

Планирование качества начинается с определения целей качества проекта, политик и стандартов, относящихся к содержанию проекта. Потом определяются действия и обязанности членов команды, выполнение которых необходимо для достижения целей и соблюдения стандартов. Результат планирования качества представляется в форме планов обеспечения качества и процессов управления, обеспечивающих выполнение этих планов, и достигается путем синхронизации с основными (планирование содержания, расписания, стоимости) и вспомогательными (планирование рисков, команды) процессами планирования.

4. Методы контроля качества проекта

Задача инструментов планирования качества − сделать процессы управления проектом предсказуемыми. Для планирования качества проекта рекомендуется использовать нижеследующие методы.

Программа обеспечения качеством − план действий, обеспечивающий соответствие фактического качества проекта запланированному качеству.

Анализ выгод и затрат. Цель метода − выдержать необходимое соотношение между доходами и затратами в проекте. Обеспечение качества проекта, несомненно, приводит к *дополнительным расходам*, поэтому для каждого предложенного метода обеспечения качества необходимо анализировать коэффициент рентабельности.

Бенчмаркинг включает в себя сопоставление действующего или планируемого проекта с другими проектами с целью выработать идеи для повышения качества исполнения проекта.

Планирование экспериментов − статистический метод, позволяющий определить факторы, которые оказывают влияние на определенные переменные величины продукта или процесса.

Стоимость качества − совокупная стоимость всех действий, направленных на повышение качества продукта или услуги и обеспечение их соответствия определенным требованиям, а также на предупреждение факторов, способных вызвать снижение качества продукта или услуги и их несоответствие требованиям (доработка).

Мероприятия по обеспечению качества должны быть разработаны в самом начале проекта и проводиться на основе независимых экспертных оценок.

Контрольные списки процедур контроля качества − структурированный документ, который используется для подтверждения выполнения всех намеченных операций. Такие списки позволяют убедиться в правильной последовательности действий в часто выполняемых задачах. *Контрольные списки* качества используются в *процессе контроля* качества.

Базовый план по качеству содержит требования к *качеству данного* проекта и служит основой для оценки и составления отчетов по исполнению требований качества.

План управления проектом (обновления). Обновление плана происходит вследствие добавления к нему вспомогательного *плана управления качеством*.

Запрошенные изменения подвергаются экспертной оценке и вносятся в соответствующие планы в процессе общего управления изменениями.

Обеспечения качества − процесс выполнения плановых систематических операций по качеству, которые обеспечивают выполнение всех предусмотренных процессов, необходимых для того, чтобы проект соответствовал установленным требованиям по качеству. Функцию обеспечения качества может выполнять *команда проекта*, руководство исполняющей организации, заказчик или спонсор, другие участники проекта. Для контроля качества проекта проводятся аудиторские проверки, целью которых является выяснение, удовлетворяет ли качество проекта стандартам, установленным в плане обеспечения качества.

Процесс обеспечения качества включает методы непрерывного улучшения качества будущих проектов. Знания и *опыт* по обеспечению качества, накопленные в текущем проекте, должны использоваться при составлении планов обеспечения качества последующих проектов.

Диаграммы зависимостей помогают анализировать причины возникновения проблем. Диаграмма зависимостей представляет собой графическое отображение процесса. Существует множество различных стилей представления этих диаграмм, но все они отображают операции, точки принятия решений и порядок обработки данных. Диаграммы зависимостей дают представление о том, как различные элементы системы взаимодействуют между собой. На рисунке 3.3 приведен пример диаграммы зависимостей для контрольных оценок. Такая диаграмма зависимостей может оказать помощь команде проекта в прогнозировании, где и какие могут возникнуть проблемы с качеством, и, следовательно, в разработке мер по их предотвращению.

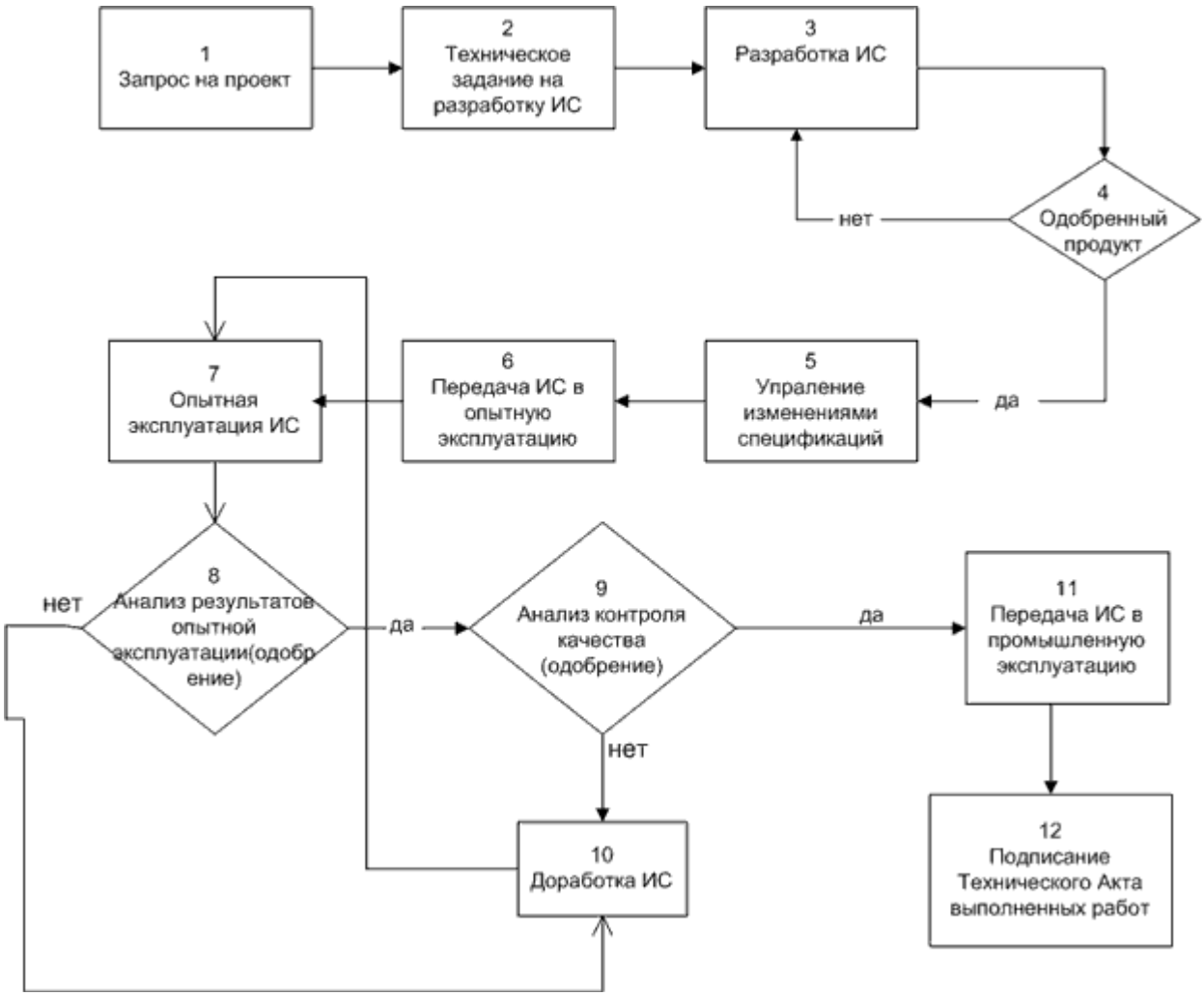


Рис. 3.3. Пример диаграммы зависимостей

Контрольные вопросы

1. Дайте определение качества программного обеспечения на основании стандартов.

2. Назовите стадии управления качеством. Дайте им краткую характеристику.

3. В чем заключается процесс обеспечения качества проекта в области информационных технологий?

4. Какие существуют методы контроля качества?

ГЛАВА 4. АВТОМАТИЗИРОВАННОЕ ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ С ИСПОЛЬЗОВАНИЕМ ПРОЕКТА SELENIUM

1. Автоматизация тестирования

Автоматизированное тестирование ПО (*Automation Testing*) − это процесс верификации программного обеспечения, при котором основные функции и шаги теста, такие как запуск, инициализация, выполнение, анализ и выдача результата, выполняются автоматически при помощи инструментов для автоматизированного тестирования.

Автоматизированное функциональное тестирование ПО (Functional Automation Testing) − это процесс верификации функциональных требований и особенностей тестируемого приложения, посредством инструментов для автоматизированного тестирования.

Преимущества автоматизации тестирования:

• повторяемость – все написанные тесты всегда будут выполняться однообразно, то есть исключен «человеческий фактор». Тестировщик не пропустит тест по неосторожности и ничего не напутает в результатах.

• быстрое выполнение – автоматизированному скрипту не нужно сверяться с инструкциями и документациями, это сильно экономит время выполнения.

• меньшие затраты на поддержку – когда автоматические скрипты уже написаны, на их поддержку и анализ результатов требуется, как правило, меньшее время чем на проведение того же объема тестирования вручную.

• отчеты – автоматически рассылаемые и сохраняемые отчеты о результатах тестирования.

Недостатки автоматизации тестирования:

• повторяемость – все написанные тесты всегда будут выполняться однообразно. Это одновременно является и недостатком, так как тестировщик, выполняя тест вручную, может обратить внимание на некоторые детали и, проведя несколько дополнительных операций, найти дефект. Скрипт этого сделать не может.

• затраты на поддержку – несмотря на то, что в случае автоматизированных тестов они меньше, чем затраты на ручное тестирование того же функционала – они все же есть. Чем чаще изменяется приложение, тем они выше.

• большие затраты на разработку – разработка автоматизированных тестов это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Естественно, все это нужно тестировать и отлаживать, а это требует времени.

• стоимость инструмента для автоматизации – в случае если используется лицензионное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты как правило отличаются более скромным функционалом и меньшим удобством работы.

• пропуск мелких ошибок − автоматический скрипт может пропускать мелкие ошибки на проверку которых он не запрограммирован. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки контролов и форм с которыми не осуществляется взаимодействие во время выполнения скрипта.

Автоматизация тестирования применяется в следующих случаях:

1. труднодоступные места в системе (бэкенд процессы, логирование файлов, запись в базу данных);

2. часто используемая функциональность, риски от ошибок в которой достаточно высоки. (автоматизировав проверку критической функциональности, можно гарантировать быстрое нахождение ошибок, а значит и быстрое их решение);

3. рутинные операции, такие как переборы данных (автоматизировать заполнение полей различными данными и их проверку после сохранения);

4. валидационные сообщения (автоматизировать заполнение полей не корректными данными и проверку на появление той или иной валидации);

5. длинные end-to-end сценарии;

6. проверка данных, требующих точных математических расчетов;

7. проверка правильности поиска данных.

А также, многое другое, в зависимости от требований к тестируемой системе и возможностей выбранного инструмента для тестирования.

Для более эффективного использования автоматизации тестирования лучше разработать отдельные тест кейсы проверяющие:

• базовые операции создания/чтения/изменения/удаления сущностей (так называемые CRUD операции - Create / Read / Update / Delete). Например: создание, удаление, просмотр и изменение данных о пользователе.

• типовые сценарии использования приложения, либо отдельные действия. Например: пользователь заходит на почтовый сайт, листает письма, просматривает новые, пишет и отправляет письмо, выходит с сайта. Это так называемый end-to-end сценарий, который проверяет совокупность действий.

Такие сценарии позволяют вернуть систему в состояние, максимально близкое к исходному, а значит – минимально влияющее на другие тесты.

• интерфейсы, работы с файлами и другие моменты, неудобные для тестирования вручную. Например: система создает некоторый xml файл, структуру которого необходимо проверить.

Для обеспечения лучшего качества продукта, необходимо придерживаться стратегии автоматизации тестирования на основе трехуровневой модели:

• уровень модульного тестирования (*Unit Tests Layer*). Под автоматизированными тестами на этом уровне понимаются компонентные или модульные тесты написанные разработчиками. Тестировщикам никто не запрещает писать такие тесты, которые будут проверять код, конечно же, если их квалификация позволяет это. Наличие подобных тестов на ранних стадиях проекта, а также постоянное их пополнение новыми тестами, проверяющими «баг фиксы», убережет проект от многих серьезных проблем.

• уровень функционального тестирования (*Functional Tests Layer Non-UI*). Как правило, не всю бизнес логику приложения можно протестировать через GUI слой. Это может быть особенностью реализации, которая прячет бизнес логику от пользователей. Именно по этой причине по договоренности с разработчиками, для команды тестирования может быть реализован доступ напрямую к функциональному слою, дающий возможность тестировать непосредственно бизнес-логику приложения, минуя пользовательский интерфейс.

• уровень тестирования через пользовательский интерфейс (*GUI Test Layer*). На данном уровне есть возможность тестировать не только интерфейс пользователя, но также и функциональность, выполняя операции вызывающую бизнес логику приложения. С нашей точки зрения, такого рода сквозные тесты дают больший эффект нежели просто тестирование функционального слоя, так как мы тестируем функциональность, эмулируя действия конечного пользователя, через графический интерфейс.

2. Введение в Selenium

SELENIUM – это проект, в рамках которого разрабатывается серия программных продуктов с открытым исходным кодом:

- Selenium WebDriver
- Selenium RC
- Selenium Server
- Selenium Grid
- Selenium IDE

Называть просто словом Selenium любой из этих пяти продуктов, вообще говоря, неправильно, хотя так часто делают, если из контекста понятно, о каком именно из продуктов идёт речь, или если речь идёт о нескольких продуктах одновременно, или обо всех сразу.

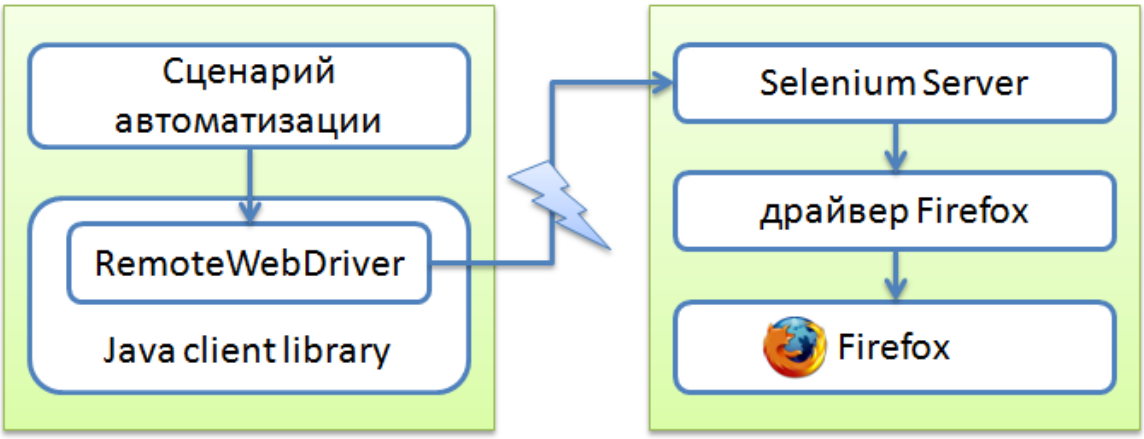
Selenium WebDriver – это программная библиотека для управления браузерами. Часто употребляется также более короткое название WebDriver. Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами. Это основной продукт, разрабатываемый в рамках проекта Selenium. Selenium WebDriver называется также Selenium 2.0. Как уже было сказано, WebDriver представляет собой семейство драйверов для различных браузеров плюс набор клиентских библиотек для этих драйверов на разных языках программирования.

В рамках проекта Selenium разрабатываются драйверы для браузеров Firefox, Internet Explorer и Safari, а также драйверы для мобильных браузеров Android и iOS. Драйвер для браузера Google Chrome разрабатывается в рамках проекта Chromium, а драйвер для браузера Opera (включая мобильные версии) разрабатывается компанией Opera Software. Поэтому они формально не являются частью проекта Selenium, распространяются и поддерживаются независимо. Но логически, конечно, можно считать их частью семейства продуктов Selenium.

Аналогичная ситуация и с клиентскими библиотеками – в рамках проекта Selenium разрабатываются библиотеки для языков Java, .Net (C#), Python, Ruby, JavaScript. Все остальные реализации не имеют отношения к проекту Selenium, хотя, возможно, в будущем, какие-то из них могут влиться в этот проект.

Selenium RC – это предыдущая версия библиотеки для управления браузерами. Аббревиатура RC в названии этого продукта расшифровывается как Remote Control, то есть это средство для «удалённого» управления браузером. Эта версия с функциональной точки зрения значительно уступает WebDriver. Сейчас она находится в законсервированном состоянии, не развивается и даже известные баги не исправляются. А всем, кто сталкивается с ограничениями Selenium RC, предлагается переходить на использование WebDriver. Иногда Selenium RC называется также Selenium 1.0, тогда как WebDriver называется Selenium 2.0. Хотя на самом деле дистрибутив версии 2.0 включает в себя одновременно обе реализации – и Selenium RC, и WebDriver. А вот когда выйдет версия 3.0 – в ней останется только WebDriver. С технической точки зрения WebDriver не является результатом эволюционного развития Selenium RC, они построены на совершенно разных принципах и у них практически нет общего кода. Объединяет их лишь тот факт, что обе реализации были сделаны в рамках проекта Selenium. Ну, или если быть совсем точными, WebDriver сначала был самостоятельным проектом, но в 2008 году произошло слияние и сейчас WebDriver представляет собой основной вектор развития проекта Selenium.

Selenium Server – это сервер, который позволяет управлять браузером с удаленной машины, по сети. Сначала на машине, где должен работать браузер, устанавливается и запускается сервер. Затем на другой машине запускается программа, которая, используя специальный драйвер RemoteWebDriver, соединяется с сервером и отправляет ему команды. Он в свою очередь запускает браузер и выполняет в нём эти команды, используя драйвер, соответствующий этому браузеру:



Selenium Server поддерживает одновременно два набора команд – для новой версии (WebDriver) и для старой версии (Selenium RC).

Selenium Grid – это кластер, состоящий из нескольких Selenium-серверов. Он предназначен для организации распределённой сети, позволяющей параллельно запускать много браузеров на большом количестве машин. Selenium Grid имеет топологию «звезда», то есть в его составе имеется выделенный сервер, который носит название «хаб» или «коммутатор», а остальные сервера называются «ноды» или «узлы». Сеть может быть гетерогенной, то есть коммутатор и узлы могут работать под управлением разных операционных систем, на них могут быть установлены разные браузеры. Одна из задач Selenium Grid заключается в том, чтобы «подбирать» подходящий узел, когда во время старта браузера указываются требования к нему – тип браузера, версия, операционная система, архитектура процессора и ряд других атрибутов.

Ранее Selenium Grid был самостоятельным продуктом. Сейчас физически продукт один – Selenium Server, но у него есть несколько режимов запуска: он

может работать как самостоятельный сервер, как коммутатор кластера, либо как узел кластера, это определяется параметрами запуска.

Selenium IDE – плагин к браузеру Firefox, который может записывать действия пользователя, воспроизводить их, а также генерировать код для WebDriver или Selenium RC, в котором выполняются те же самые действия. В общем, это «Selenium-рекордер».

Тестировщики могут использовать Selenium IDE как самостоятельный продукт, без преобразования записанных сценариев в программный код. Это, конечно, не позволяет разрабатывать достаточно сложные тестовые наборы, но иногда хватает и простых линейных сценариев.

Возможности проекта Selenium представлены на рисунке 4.1.

Рис. 4.1. Возможности проекта Selenium

Приведем характеристику каждого вида тестирования:

1. *Core functional testing* (*функциональное тестирование*) – это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие задачи решает.

Функциональные требования включают в себя:

- Функциональную пригодность (*suitability*).
- Точность (*accuracy*).
- Способность к взаимодействию (*interoperability*).
- Соответствие стандартам и правилам (*compliance*).
- Защищённость (*security*).

2. *GUI testing* − тестирование приложений через графический пользовательский интерфейс. Популярность такого вида тестирования объясняется двумя факторами: во-первых, приложение тестируется тем же способом, которым его будет использовать пользователь, во-вторых, можно тестировать приложение, не имея при этом доступа к исходному коду.

3. *Database testing* − тестирование целостности базы данных. Проверяется согласованность данных. Включает в себя проверку: тестируются данные и базы данных независимо от пользовательского интерфейса – ввод данных и работа с ними непосредственно в базе данных; ссылочной целостности; ограничений на значения параметров; ограничений на не инициализацию значений; ограничений на уникальность значений. Тестируются данные и базы данных независимо от пользовательского интерфейса – ввод данных и работа с ними непосредственно в базе данных.

4. *Usability testing* (*юзабилити тестирование, проверка эргономичности*) − исследование, выполняемое с целью определения, удобен ли некоторый искусственный объект (такой как веб-страница, пользовательский интерфейс или устройство) для его предполагаемого применения. Таким образом, проверка эргономичности измеряет эргономичность объекта или системы. Проверка эргономичности сосредоточена на определённом объекте или небольшом наборе объектов, в то время как исследования взаимодействия человек-компьютер в целом формулируют универсальные принципы. Проверка эргономичности − метод оценки удобства продукта в использовании, основанный на привлечении пользователей в качестве тестировщиков, испытателей и суммировании полученных от них выводов.

5. *Regression testing* (*регрессионное тестирование*) − повторное тестирование после внесение изменений в программное обеспечение или в его окружение (в новой версии приложения), чтобы убедиться, в том, что функции, которые работали в предыдущей версии системы, по-прежнему работают так, как ожидалось. Выявление потенциальных проблем, которые могли возникнуть в результате изменений.

6. *Sanity testing* (*санитарное тестирование или проверка согласованности*) − это узконаправленное тестирование, достаточное для доказательства того, что конкретная функция работает согласно заявленным в спецификации требованиям. Используется для определения работоспособности определенной части приложения после изменений произведенных в ней или окружающей среде.

7. *Cross-browser testing* (*тестирование кроссбраузерности*) − вид тестирования, направленный на поддержку и правильное полное отображение программного продукта в разных браузерах, мобильных устройствах, планшетах, экранах различного размера. Кроссбраузерное тестирование − важный этап при разработке любой программы. Ведь внешний вид сайта и его корректное отображение на любом современном устройстве играет определяющую роль для заказчика.

8. *End-to-end* тестирование − это процесс тестирования приложения на всех уровнях − начиная с фронтенда и заканчивая бэкэндом, включая интерфейс и конечные точки. Выполнение *end-to-end* тестирования гарантирует, что приложение проверено на основе пользовательских сценариев, которые помогают контролировать и избегать рисков, позволяя тестировщикам:

- проверять и выполнять тестирование всего потока приложения;
- увеличивать тестовое покрытие за счет привлечения различных подсистем;
- обнаруживать проблемы в общей производительности приложения.

9. *Business process testing* (*тестирование на основе бизнес-процессов*) − метод тестирования, в котором тестовые сценарии проектируются на основании описаний и/или знаниях бизнес-процессов.

3. Сравнение с другими инструментами

Как выбрать инструмент для автоматизации тестирования?

Во-первых, необходимо обратить внимание насколько хорошо инструмент для автоматизации распознает элементы управления в вашем приложении. В случае, когда элементы не распознаются, стоит поискать плагин, либо соответствующий модуль. Если такового нет – от инструмента лучше отказаться. Чем больше элементов может распознать инструмент – тем больше времени вы сэкономите на написании и поддержке скриптов.

Во-вторых, нужно обратить внимание на то, сколько времени требуется на поддержку скриптов написанных с помощью выбранного инструмента. Для этого запишите простой скрипт, который выбирает пункт меню, а потом представьте, что изменился пункт меню, который необходимо выбрать. Если для восстановления работоспособности сценария вам придется перезаписать скрипт целиком, то инструмент не оптимален, так как реальные сценарии гораздо сложнее. Лучше всего тот инструмент, который позволяет вам вынести название кнопки в переменную в начале скрипта и быстро заменить ее значение. В идеале – описать меню как класс.

И последний момент, на который нужно обратить внимание – насколько удобен инструмент для написания новых скриптов. Сколько требуется на это времени, насколько можно структурировать код (поддержка ООП), насколько код читаем, насколько удобна среда разработки для рефакторинга (переработки кода) и т.п.

Популярные инструменты и фреймворки для автоматизации тестирования программного обеспечения:

✓ Selenium – самый популярный фреймворк для автоматизации тестирования;

✓ Katalon Studio – инструмент для автоматизации тестирования веб/мобильных приложений;

✓ UFT – коммерческий инструмент для функционального тестирования;

✓ Watir – инструмент для автоматизации тестирования, использующий библиотеки Ruby;

✓ IBM Rational Functional Tester – инструмент тестирования приложений HTML, Java, Windows, .NET, Visual Basic, Eclipse, SAP;

✓ TestComplete – инструмент для тестирования десктопных, мобильных и веб-приложений

4. Базовая функциональность Selenium

Тесты Selenium могут быть записаны на нескольких языках программирования, таких как Java, C#, JavaScript, Python и Ruby. Приведем примеры простых скриптов Selenium тестов:

C#

```
using System;
using OpenQA.Selenium;
using OpenQA.Selenium.Firefox;
using OpenQA.Selenium.Support.UI;

class GoogleSearch
{
  static void Main()
  {
    IWebDriver driver = new FirefoxDriver();
    driver.Navigate().GoToUrl("http://www.google.com");
    IWebElement query = driver.FindElement(By.Name("q"));
    query.SendKeys("Hello Selenium WebDriver!");
    query.Submit();
    Console.WriteLine(driver.Title);
  }
}
```

Java

```java
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;

public class GoogleSearch  {
  public static void main(String[] args) {
    // Create a new instance of the html unit driver
    // Notice that the remainder of the code relies on the interface,
    // not the implementation.
    WebDriver driver = new FirefoxDriver();

    // And now use this to visit Google
    driver.get("http://www.google.com");

    // Find the text input element by its name
    WebElement element = driver.findElement(By.name("q"));

    // Enter something to search for
    element.sendKeys("Hello Selenium WebDriver!");

    // Submit the form based on an element in the form
    element.submit();

    // Check the title of the page
    System.out.println("Page title is: " + driver.getTitle());
  }
}
```

JavaScript

```javascript
var webdriver = require('selenium-webdriver');
var driver = new webdriver.Builder()
    .forBrowser('chrome')
    .build();

driver.get('http://www.google.com/ncr');
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
driver.findElement(webdriver.By.name('btnG')).click();
driver.wait(webdriver.until.titleIs('webdriver - Google Search'), 1000);
console.log(driver.title);
```

Python

```python
from selenium import webdriver

driver = webdriver.Firefox()
driver.get("http://www.google.com")

elem = driver.find_element_by_name("q")
elem.send_keys("Hello WebDriver!")
elem.submit()

print(driver.title)
```

Ruby

```ruby
require "selenium-webdriver"

driver = Selenium::WebDriver.for :firefox
driver.navigate.to "http://www.google.com"

element = driver.find_element(:name, 'q')
element.send_keys "Hello Selenium WebDriver!"
element.submit

puts driver.title
```

Поддерживаемые платформы. Список браузеров, поддерживаемых разработчиками проекта Selenium:

Microsoft Windows:

- Firefox 2, 3, 3.x , 4 − 54;
- Internet Explorer 6, 7, 8 , 9, 10;
- Safari 2, 3, 4;
- Opera 8, 9, 10, 11, 12;
- Google Chrome 12.0.712.0+;

Mac OS X:

- Safari 2, 3, 4;
- Firefox 2, 3, 3.x;
- Camino 1.0a1;
- Mozilla Suite 1.6+, 1.7+;
- Seamonkey 1.0;

GNU/Linux:

- Firefox 2, 3, 3.x;
- Mozilla Suite 1.6+, 1.7+;
- Konqueror;
- Opera 8, 9, 10;
- Google Chrome 12.0.712.0+;

и другие.

Существует возможность дополнить Selenium своими проверками, действиями и локаторами. Это делается с помощью JavaScript, добавлением методов к прототипам объектов Selenium и PageBot. Во время загрузки Selenium автоматически просматривает методы этих прототипов используя шаблоны имен для определения того, что является действием, что проверкой, а что локатором.

Все методы прототипа Selenium, которые начинаются с “do”, добавляются как действия. Для каждого действия “foo” также регистрируется действие “fooAndWait”. Метод действия может принимать до двух параметров, которые будут записаны в тест как значения второй и третьей колонки. Рассмотрим следующий пример: добавление действия “typeRepeated” в Selenium, которое будет дважды печатать текст в текстовое поле.

```
Selenium.prototype.doTypeRepeated = function(locator, text) {
        // Все стратегии поиска автоматически обрабатываются
        // с помощью "findElement"
        var element = this.page().findElement(locator);
        // Создает текст для заполнения поля
        var valueToType = text + text;
        // Заменяет содержание текстового поля новым текстом
        this.page().replaceText(element, valueToType);
};
```

Пользовательские расширения можно использовать с Selenium IDE по следующему алгоритму:

1. Создайте свое пользовательское расширение и сохраните его как “user-extensions.js”. Такое имя не является технической необходимостью, но использование однообразных имен - хорошая практика.

2. Откройте Firefox и Selenium IDE. Выберите Tools, затем Options.

3. В Selenium Core Extensions выберите Browse и найдите файл “user-extensions.js”. Нажмите OK.

4. Закройте и перезапустите Selenium IDE, чтобы загрузить свое пользовательское расширение.

5. Создайте новую команду в своем тесте. Ваше пользовательское расширение должно быть теперь доступно в меню Commands.

Контрольные вопросы

1. В чем заключается преимущество автоматизированного функционального тестирования?

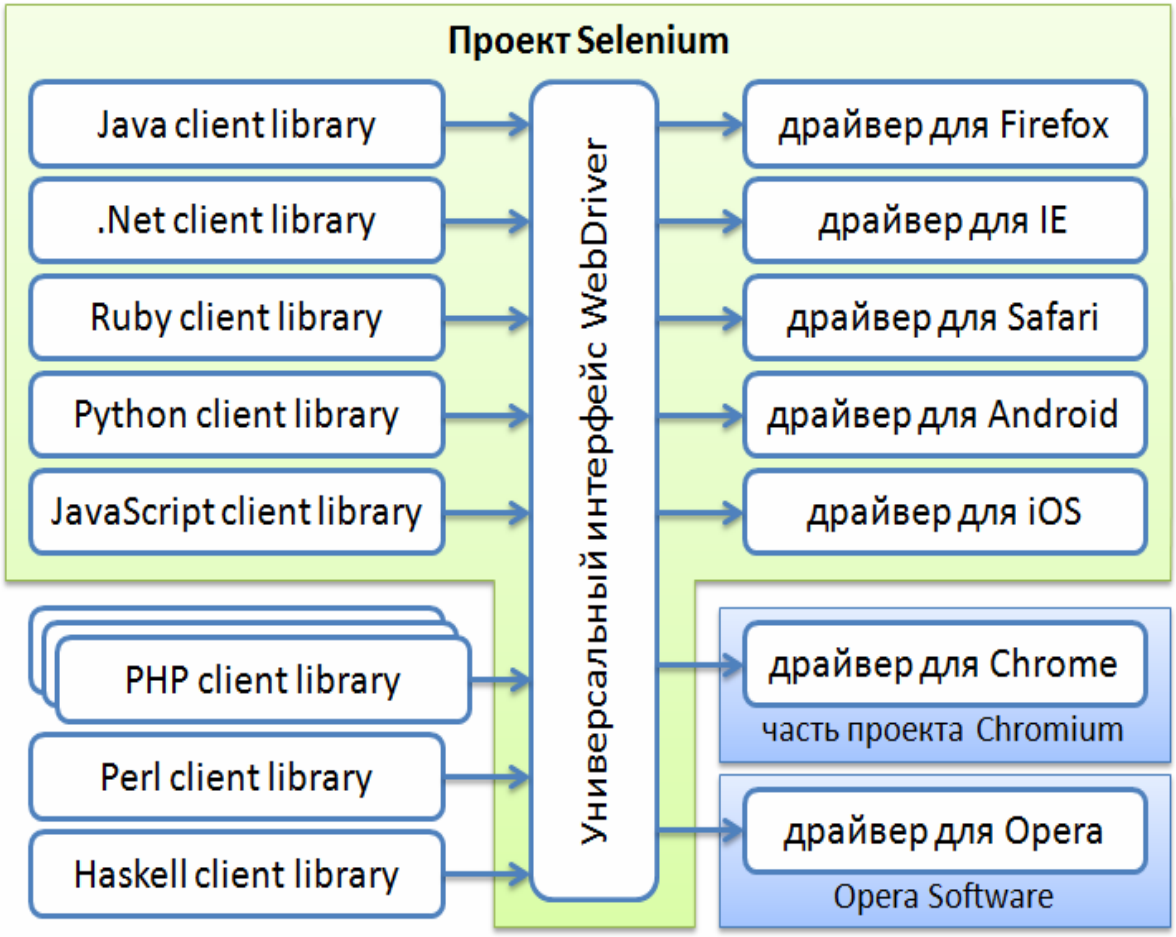
2. Как выбрать инструмент для автоматизации тестирования?

3. Дайте характеристику проекту Selenium: назначение, основные модули, поддерживаемые языки программирования, платформы, браузеры.

4. Приведите краткий обзор других программ для проведения автоматизированного тестирования.

ГЛАВА 5. ПРИНЦИП РАБОТЫ SELENIUM WEBDRIVER

1. Обзор и принцип работы Selenium WebDriver



Selenium Webdriver – инструмент для автоматизации реального браузера, как локально, так и удаленно, наиболее близко имитирующий действия пользователя.

Selenium 2 (или Webdriver) – последнее пополнение в пакете инструментов Selenium и является основным вектором развития проекта. Это абсолютно новый инструмент автоматизации. По сравнению с Selenium RC Webdriver использует совершенно иной способ взаимодействия с браузерами. Он напрямую вызывает команды браузера, используя родной для каждого конкретного браузера API. Как совершаются эти вызовы и какие функции они выполняют зависит от конкретного браузера. В то же время Selenium RC внедрял javascript код в браузер при запуске и использовал его для управления веб-приложением. Таким образом, Webdriver использует способ

взаимодействия с браузером более близкий к действиям реального пользователя.

По сравнению с более старым интерфейсом он обладает рядом преимуществ:

- Интерфейс Webdriver был спроектирован более простым и выразительным;
- Webdriver обладает более компактным и объектно-ориентированным API;
- Webdriver управляет браузером более эффективно, а также справляется с некоторыми ограничениями, характерными для Selenium RC, как загрузка и отправление файлов, попапы и дилоги.

Для работы с Webdriver необходимо 3 основных программных компонента:

1. *Браузер*, работу которого пользователь хочет автоматизировать. Это реальный браузер определенной версии, установленный на определенной ОС и имеющий свои настройки (по умолчанию или кастомные).

2. Для управления браузером необходим драйвер браузера. Драйвер на самом деле является веб-сервером, который запускает браузер и отправляет ему команды, а также закрывает его. У каждого браузера свой драйвер. Связано это с тем, что у каждого браузера свои отличные команды управления и реализованы они по-своему. Найти список доступных драйверов и ссылки для скачивания можно на официальном сайте Selenium проекта.

3. *Скрипт/тест*, который содержит набор команд на определенном языке программирования для драйвера браузера. Такие скрипты используют Selenium Webdriver bindings (готовые библиотеки), которые доступны пользователям на различных языках.

По назначению Selenium WebDriver представляет собой драйвер браузера, то есть программную библиотеку, которая позволяет разрабатывать программы, управляющие поведением браузера.

По своей сущности Selenium WebDriver представляет собой:

✓ спецификацию программного интерфейса для управления браузером,

✓ референсные реализации этого интерфейса для нескольких браузеров,

✓ набор клиентских библиотек для этого интерфейса на нескольких языках программирования.

2. Основные понятия и методы Selenium Webdriver API

Основными понятиями в Selenium Webdriver являются:

- Webdriver − самая важная сущность, ответственная за управление браузером. Основной ход скрипта/теста строится именно вокруг экземпляра этой сущности.
- Webelement − вторая важная сущность, представляющая собой абстракцию над веб-элементом (кнопки, ссылки, инпута и др.). Webelement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- By − абстракция над локатором веб-элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

Сам процесс взаимодействия с браузером через Webdriver API довольно прост:

1. Нужно создать Webdriver:
2. `WebDriver driver = new ChromeDriver();`

При выполнении этой команды будет запущен Chrome, при условии, что он установлен в директорию по умолчанию и путь к ChromeDriver сохранен в системной переменной PATH.

3. Необходимо открыть тестируемое приложение (AUT), перейдя по url:

4. `driver.get("http://mycompany.site.com");`

Теоретически в Chrome при этом должен открыться сайт компании.

5. Далее следует серия действий по нахождению элементов на странице и взаимодействию с ними:

6. `By elementLocator = By.id("#element_id");`

7. `WebElement element = driver.findElement(elementLocator));`

Или более кратко:

```
WebElement element = driver.findElement(By.id("#element_id")));
```

После нахождения элемента, кликнем по нему:

```
element.cli List<WebElement> elements =
driver.findElements(By.name("elements_name"))ck();
```

Далее следует совокупность похожих действий, как того требует сценарий.

8. В конце теста (часто также и в середине) должна быть какая-то проверка, которая и определит в конечном счете результат выполнения теста:

9. `assertEquals("Webpage expected title", driver.getTitle());`

Проверки может и не быть, если цель вашего скрипта − не тест, а выполнение некоторого действия.

10.После теста надо закрыть браузер:

11.`driver.quit();`

Следует отметить, что для поиска элементов доступно два метода:

1. Первый - найдет только первый элемент, удовлетворяющий локатору:

```
WebElement element = driver.findElement(By.id("#element_id")));
```

2. Второй - вернет весь список элементов, удовлетворяющих запросу:

```
List<WebElement> elements =
driver.findElements(By.name("elements_name")).
```

3. Типы локаторов. Ожидания

Поскольку Webdriver − это инструмент для автоматизации веб-приложений, то большая часть работы с ним это работа с веб-элементами (*WebElements*). WebElements − ни что иное, как DOM объекты, находящиеся на веб-странице. А для того, чтобы осуществлять какие-то действия над DOM

объектами/веб-елементами необходимо их точным образом определить (найти).

WebElement element = driver.findElement(By.<Selector>);

Таким образом, в Webdriver определяется нужный элемент. By − класс, содержащий статические методы для идентификации элементов:

1. **By.id**

Пример:

```
<div id="element">
     <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.id("element_id"));
```

2. **By.name**

Пример:

```
<div name="element">
     <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.name("element_name"));
```

3. **By.className**

Пример:

```
<img class="logo">
```

Поиск элемента:

```
WebElement element = driver.findElement(By.className("element_class"));
```

4. **By.TagName**

Пример:

```
<div>
     <a class="logo" ref="...">...</a>
     <a class="support" ref="...">...</a>
</div>
```

Поиск элемента:

```
List<WebElement> elements = driver.findElements(By.tagName("a"));
```

5. **By.LinkText**

Пример:

```
<div>
     <a ref="...">text</a>
     <a ref="...">Another text</a>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.linkText("text"));
```

6. **By.PartialLinkText**

Поиск элемента:

```
WebElement element =
driver.findElement(By.partialLinkText("text"));
```

7. **By.cssSelector**

```
<div class='main'>
     <p>text</p>
     <p>Another text</p>
</div>
```

Поиск элемента:

```
WebElement element=driver.FindElement(By.cssSelector("div.main"));
```

8. **By.XPath**

```
<div class='main'>
     <p>text</p>
     <p>Another text</p>
</div>
```

Поиск элемента:

```
WebElement element =
driver.findElement(By.xpath("//div[@class='main']"));
```

Ожидания – непременный атрибут любых UI тестов для динамических приложений. Нужны они для синхронизации работы AUT и тестового скрипта. Скрипт выполняется намного быстрее реакции приложения на команды, поэтому часто в скриптах необходимо дожидаться определенного состояния приложения для дальнейшего с ним взаимодействия.

Самый простой пример − переход по ссылке и нажатие кнопки:

```
driver.get("http://google.com");
driver.findElement(By.id("element_id")).click();
```

В данном случае необходимо дождаться пока не появится кнопка с id = element_id и только потом совершать действия над ней. Для этого и существуют ожидания.

Ожидания бывают:

1. Неявные ожидания (*Implicit Waits)* − конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течении заданного периода времени, если элемент не найден сразу. Только по истечении этого времени WebDriver бросит ElementNotFoundException.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://some_url");
WebElement dynamicElement =
driver.findElement(By.id("dynamicElement_id"));
driver.findElement(By.id("dynamicElement_id"));
```

Неявные ожидания обычно настраиваются сразу после создания экземпляра WebDriver и действуют в течении всей жизни этого экземпляра, хотя переопределить их можно в любой момент. К этой группе ожиданий также можно отнести неявное ожидание загрузки страницы:

```
driver.manage().timeouts().pageLoadTimeout(10, TimeUnit.SECONDS);
```

А также неявное ожидание отработки скриптов:

```
driver.manage().timeouts().setScriptTimeout(10, TimeUnit.SECONDS);
```

2. Явные ожидания (*Explicit Waits)* − это код, который ждет наступления какого-то события (чаще всего на AUT), прежде чем продолжит выполнение команд скрипта. Такое ожидание срабатывает один раз в указанном месте.

Самым худшим вариантом является использование Thread.sleep(1000), в случае с которым скрипт просто будет ждать определенное количество

времени. Это не гарантирует наступление нужного события либо будет слишком избыточным и увеличит время выполнения теста.</p>

Более предпочтительно использовать WebDriverWait и ExpectedCondition:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://some_url");
WebElement dynamicElement = (new WebDriverWait(driver, 10))
.until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicElemen
t_id")));
```

В данном случае скрипт будет ждать элемента с id = dynamicElement_id в течении 10 секунд, но продолжит выполнение, как только элемент будет найден. При этом WebDriverWait класс выступает в роли конфигурации ожидания - задаем, как долго ждать события и как часто проверять его наступление. ExpectedConditions − статический класс, содержащий часто используемые условия для ожидания.

4. Пример использования Webdriver API

```
import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedCondition;
import org.openqa.selenium.support.ui.WebDriverWait;
public class WebDriverExample  {
    public static void main(String[] args) {
        // Создаем экземпляр WebDriver
        // Следует отметить что скрипт работает с интерфейсом,
        // а не с реализацией.
        WebDriver driver = new FirefoxDriver();
        // Открываем гугл, используя драйвер
        driver.get("http://www.google.com");
        // По-другому это можно сделать так:
        // driver.navigate().to("http://www.google.com");
        // Находим элемент по атрибуту name
```

```
        WebElement element = driver.findElement(By.name("q"));
        // Вводим текст
        element.sendKeys("Selenium");
        // Отправляем форму, при этом дравер сам определит как отправить
форму по элементу
        element.submit();
        // Проверяем тайтл страницы
        System.out.println("Page title is: " + driver.getTitle());
        // Страницы гугл динамически отрисовывается с помощью javascript
        // Ждем загрузки страницы с таймаутом в 10 секунд
        (new WebDriverWait(driver, 10)).until(new
ExpectedCondition<Boolean>() {
            public Boolean apply(WebDriver d) {
                return d.getTitle().toLowerCase().startsWith("selenium");
            }
        });
        // Ожидаем увидеть: "Selenium - Google Search"
        System.out.println("Page title is: " + driver.getTitle());
        // Закрываем браузер
        driver.quit();
    }
}
```

Контрольные вопросы

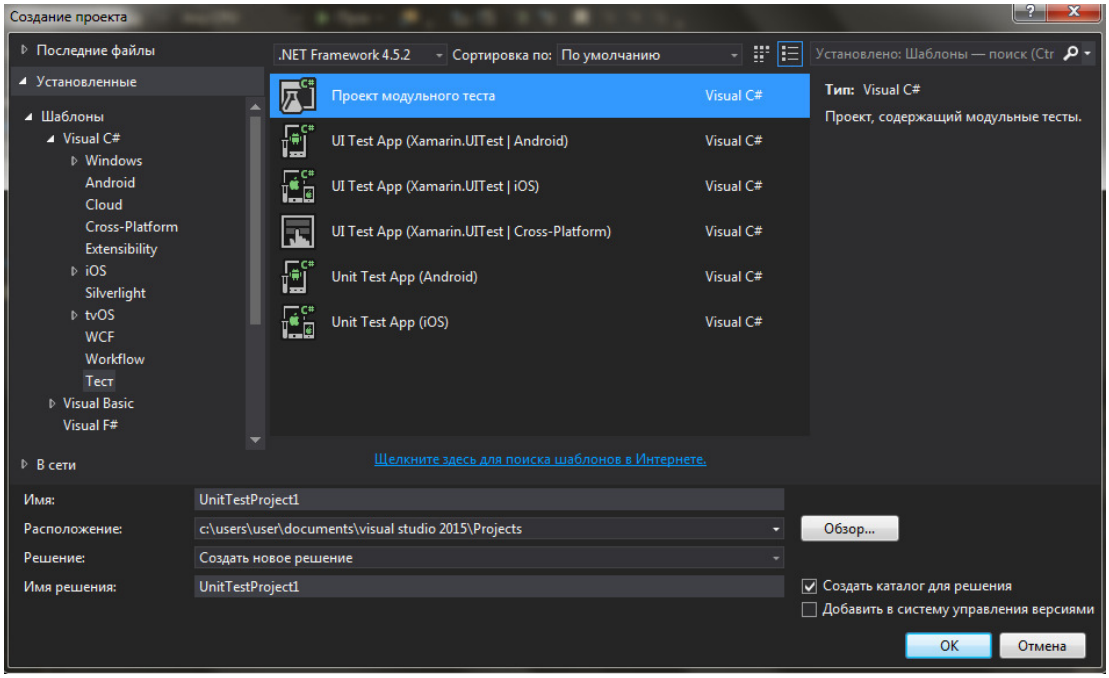
1. Для чего предназначен модуль Selenium WebDriver?
2. Основные понятия и методы Selenium Webdriver API.
3. Какие существуют типы локаторов?
4. Что такое ожидания?
5. Приведите пример использования Webdriver API.

ГЛАВА 6. ЛАБОРАТОРНЫЙ ПРАКТИКУМ

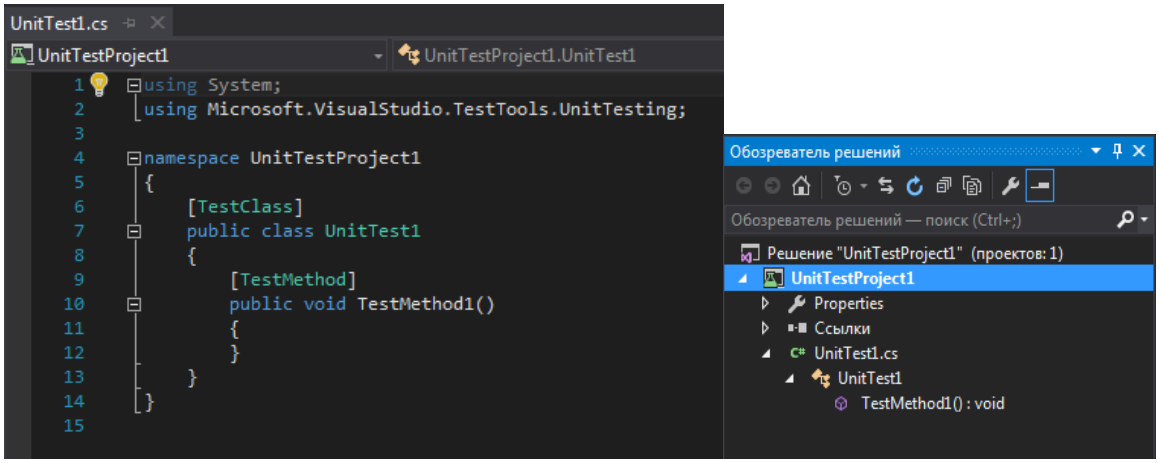
1. Применение Selenium WebDriver и Visual Studio для автоматизации тестирования веб-страниц

Шаг 1: создадим проект.

Создаем новый проект, во вкладке C# выбираем графу тест, далее кликаем на проект модульного теста. Выбираем название и нажимаем ок.

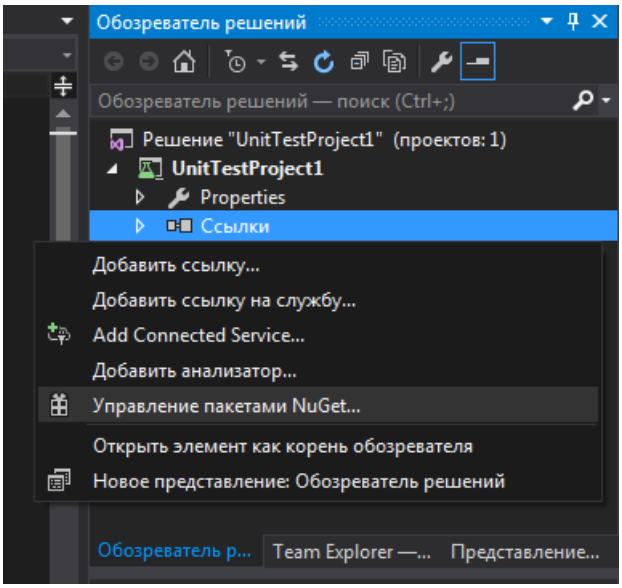


Получим:

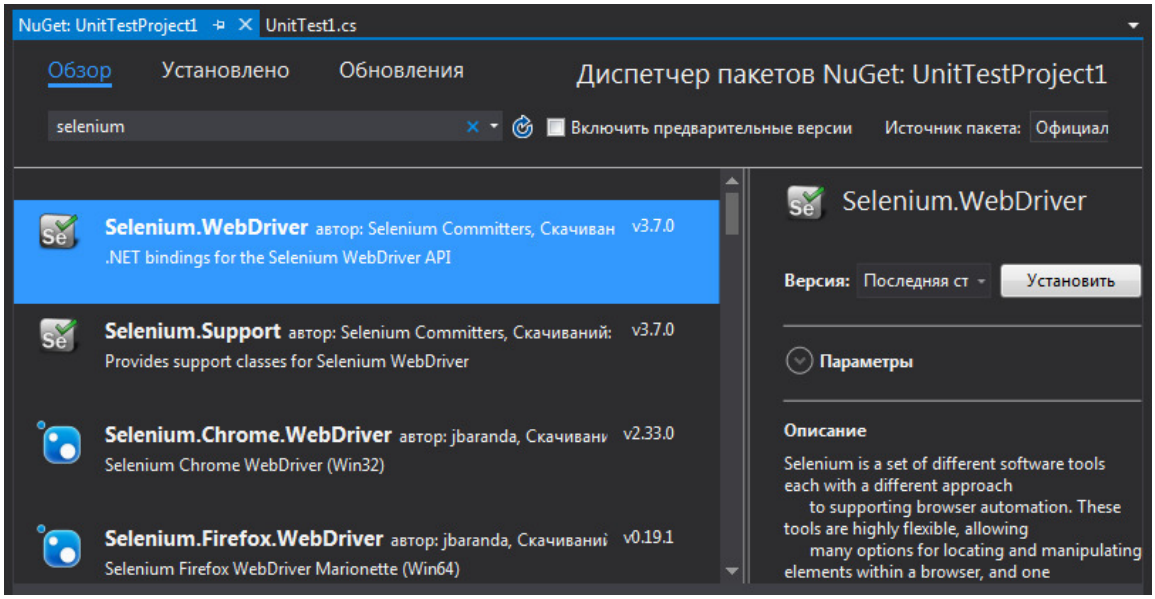


Шаг 2:добавим WebDriver в ссылки.

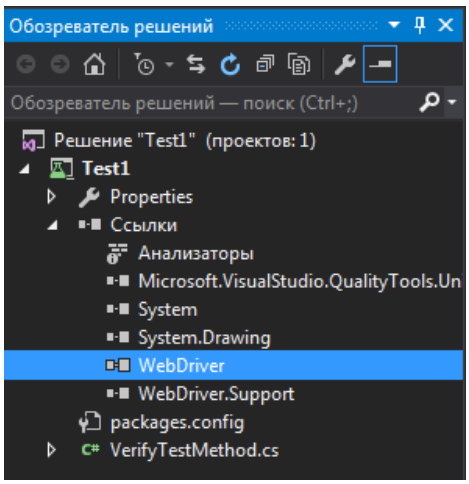
Кликаем правой кнопкой мыши по графе ссылки в обозревателе решений и выбираем управление пакетами NuGet.



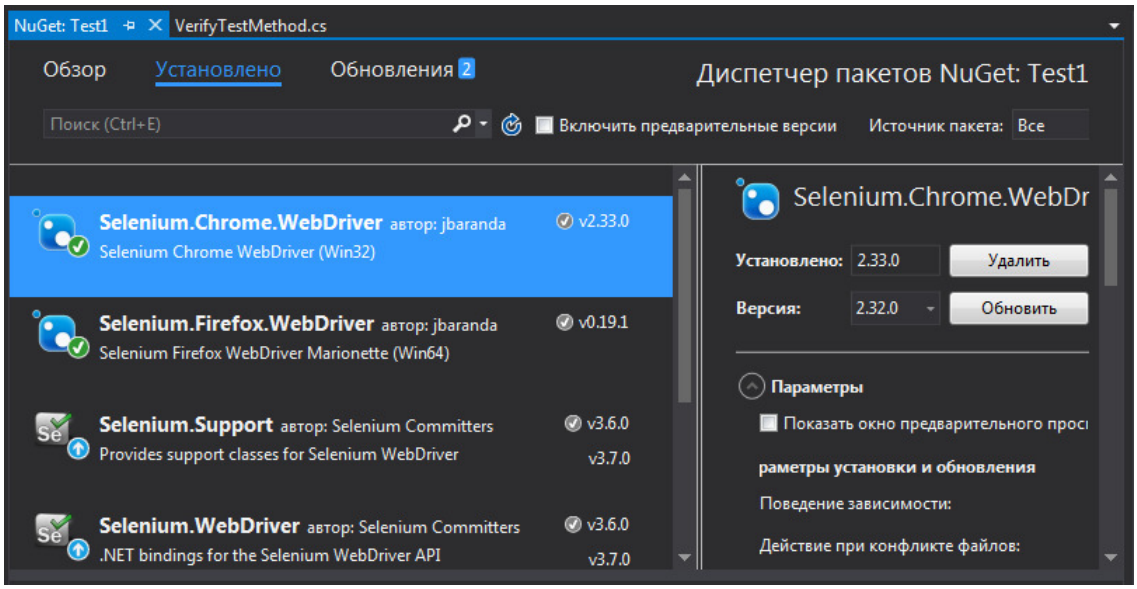
В графе поиска вбиваем selenium и скачиваем и устанавливаем последнюю версию Selenium WebDriver Support Classes.



После установки WebDriver и WebDriver.support будут видны в ссылках.



Так как для проведения тестов будем использовать браузер Chrome, необходимо скачать Chrome Driver.



Для использования других браузеров все аналогично, просто скачиваем нужную компоненту.

Шаг 3: Пробный код.

Перед проектированием необходимо подключить библиотеки:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using OpenQA.Selenium.Chrome;
using OpenQA.Selenium;
using OpenQA.Selenium.Support;
```

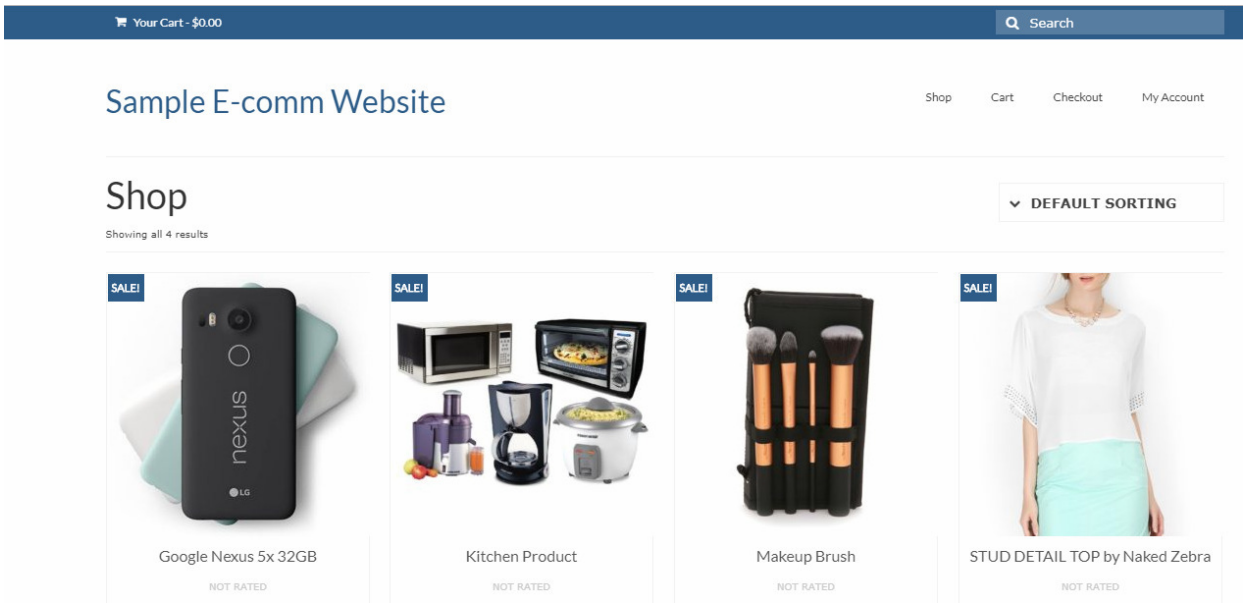
В UnitTest находятся 3 атрибута, которые мы будем использовать:

[TestInitialize] − Идентифицирует метод, который должен выполняться до того, как тест выделит и настроит ресурсы, необходимые всем тестам в тестовом классе.

[TestCleanup] − Идентифицирует метод, содержащий код для использования после запуска всех тестов в тестовом классе и для освобождения ресурсов, полученных тестовым классом.

[TestMethod] − Этот атрибут содержит в себе тестовый код, который будет запускаться после **[TestInitialize].**

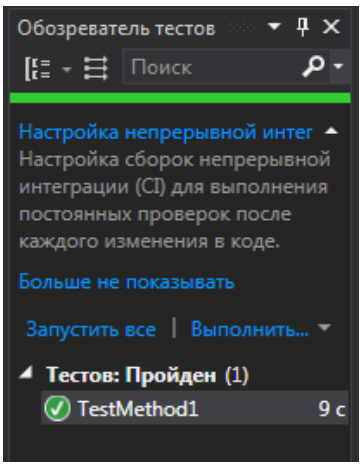
Сайт для тестирования:



Напишем теперь пробный код, который откроет и закроет браузер. Для этого нам нужно создать переменную типа IWebDriver, которая контролирует браузер.

```
namespace Test1
{
    [TestClass]
    public class VerifyTestMethod
    {
        IWebDriver driver;
        string baseUrl = "http://learnseleniumtesting.com/demo/";

        [TestInitialize]
        public void TestSet()
        {
            driver = new ChromeDriver();
            driver.Navigate().GoToUrl(baseUrl);
        }

        [TestCleanup]
        public void TestCleanup()
        {
            driver.Quit();

        }

        [TestMethod]
        public void TestMethod1()
        {

        }


    }
}
```

Запуск тестов происходит через обозреватель тестов, если все прошло успешно, получим следующий результат:



Теперь добавим еще один шаг: перед закрытием браузера, кликнем на элемент веб-страницы, например мобильный телефон.

```
public void TestMethod1()
{
    driver.FindElement(By.PartialLinkText("Nexus")).Click();
    var sku = driver.FindElement(By.CssSelector("#product_wrapper > div:nth-child(1) > div > div > a > h5"));


}
```

Для того чтобы выполнить “клик” по элементу страницы, нужно сначала найти этот элемент. Для этого используем метод FindElement. Искать будем с помощью Css selector.

Чтобы получить Css путь элемента, кликаем правой кнопкой мыши по элементу, далее нажимаем “Inspect”, copy и selector. Копируем и вставляем этот путь в Css selector и запускаем тестирование.

Получим:

Шаг 4: Тестирование.

Имеем 2 сценария:

1) Открываем браузер. Переходим на сайт. Нажимаем на телефон. Добавляем телефон в корзину. Проверяем предыдущий шаг. Проверяем наличие телефона в корзине. Переходим к оформлению заказа.

2) Открываем браузер. Переходим на сайт. Открываем страницу с кистью. Проверяем соответствие описания кисти на подлинность. Добавляем в корзину.

В обоих сценариях имеются общие шаги, поэтому [Testinitialize] будет одинаков.

```
namespace Test1
{
    [TestClass]
    public class VerifyTestMethod
    {
        IWebDriver driver;
        string baseUrl = "http://learnseleniumtesting.com/demo/";

        [TestInitialize]
        public void TestSet()
        {
            driver = new ChromeDriver();
            driver.Navigate().GoToUrl(baseUrl);
        }

        [TestCleanup]
        public void TestCleanup()
        {
            driver.Quit();

        }
```

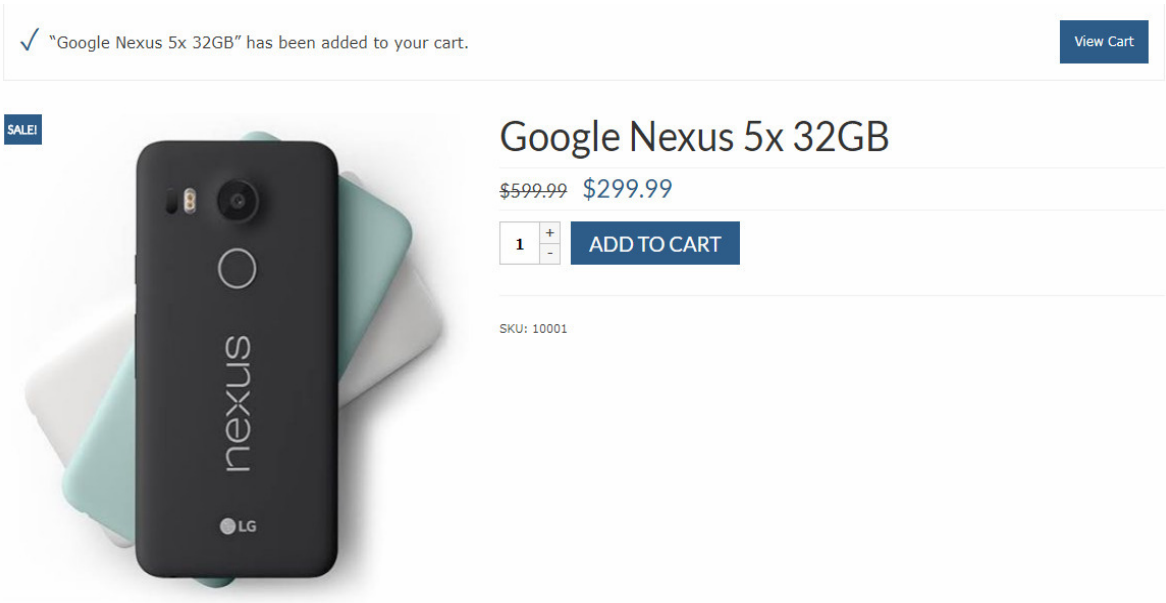
Теперь переходим непосредственно к сценариям, их у нас 2, соответственно нам нужно составить 2 [TestMethod].

Код первого сценария:

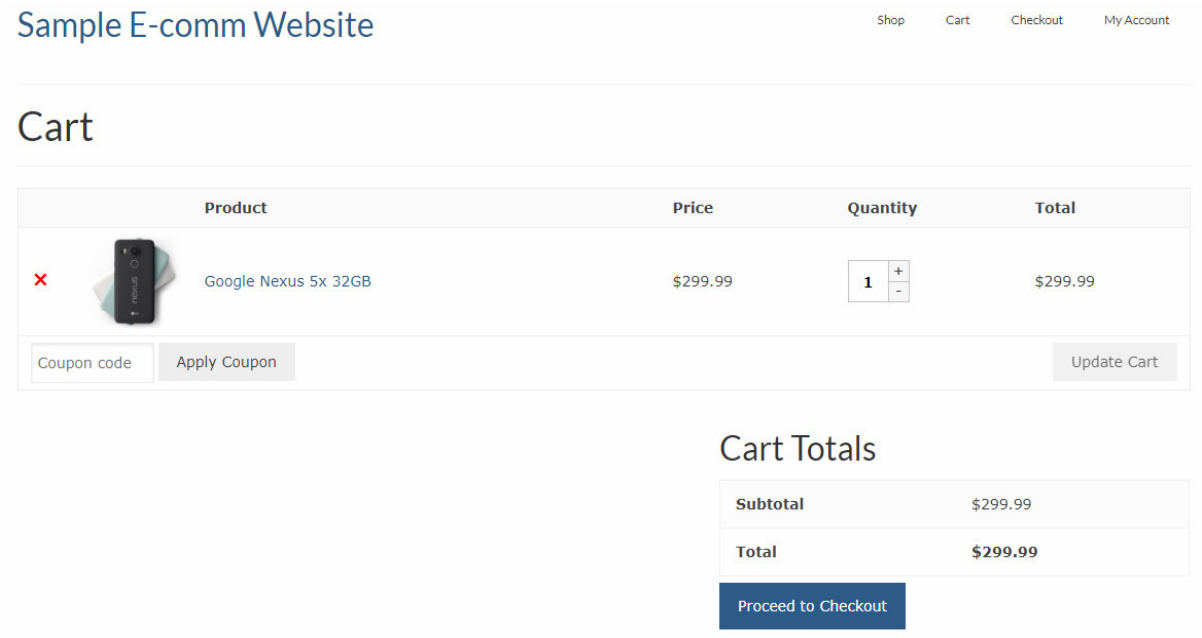
```
[TestMethod]
public void AddPhoneToCartAndVerifyInCart()
{

    driver.FindElement(By.PartialLinkText("Nexus")).Click();
    var sku = driver.FindElement(By.CssSelector("#product_wrapper > div:nth-child(1) > div > div > a > h5")).Text;
    Assert.AreEqual("10001", sku);
    driver.FindElement(By.CssSelector("#product-18 &amp;gt; div.row &amp;gt; div.col-md-7 &amp;gt; div &amp;gt; form &amp;gt; button")).Click();
    //Verify if item added to cart.
    driver.FindElement(By.XPath("//*[@id='content']/div/div/div[2][contains(text(), 'was successfully added to your cart')]"));

    //Clikc Go to cart\
    driver.FindElement(By.CssSelector("#content &amp;gt; div &amp;gt; div &amp;gt; div.woocommerce-message &amp;gt; a")).Click();
    var dd = driver.FindElement(By.XPath("//*[@id='content']/div/div/div/form/table/tbody/tr[1]/td[1]")).Text;

    //Verify If one phone given the cart.
    Assert.AreEqual("x", driver.FindElement(By.XPath("//*[@id='content']/div/div/div/form/table/tbody/tr[1]/td[1]")).Text);
    Assert.AreEqual("1", driver.FindElement(By.XPath("//*[@id='content']/div/div/div/form/table/tbody/tr[1]/td[5]")).Text);
    Assert.AreEqual("Proceed to Checkout", driver.FindElement(By.CssSelector("#content > div > div > div > div > div > div > div > a")).Text);
}
```

Выбираем телефон.

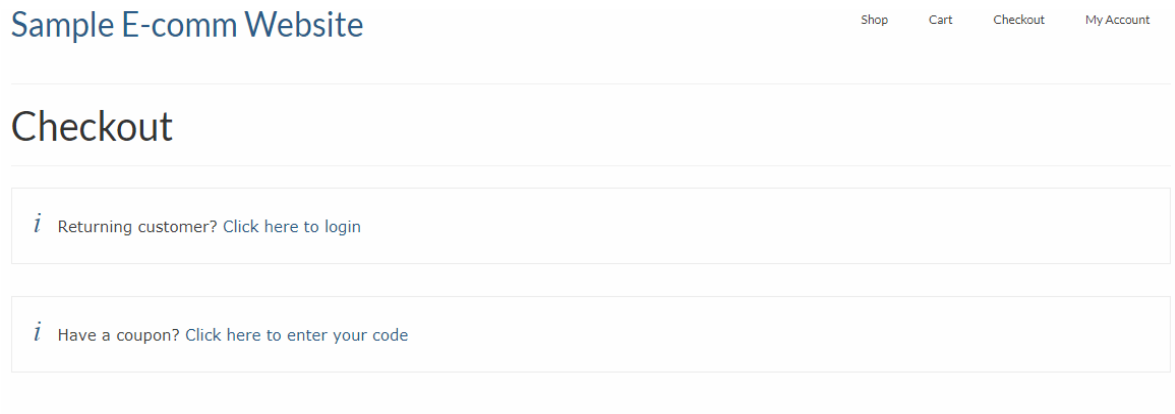
Добавляем телефон в корзину. Тут же проверяем этот шаг.



Переходим в корзину. Проверяем наличие 1 позиции телефона в корзине.



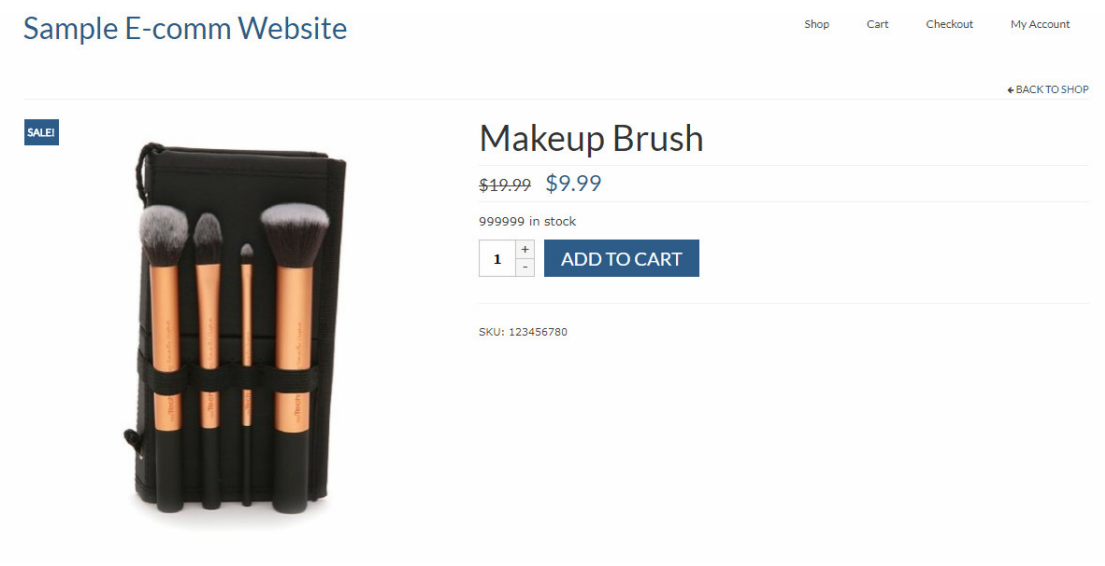
Переходим к оформлению товара. 1 сценарий завершен.



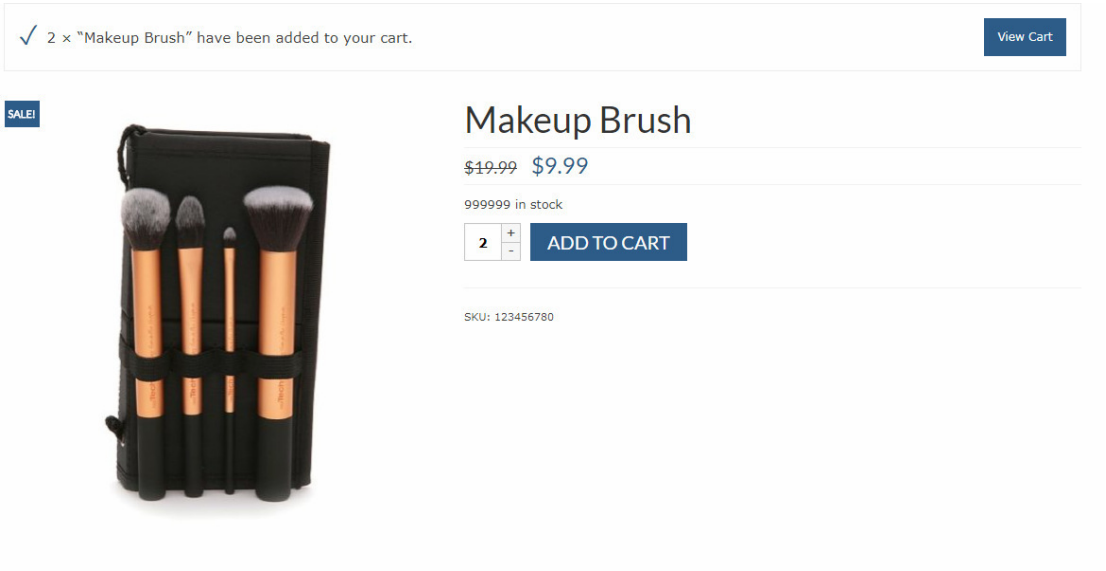
Код второго сценария:

```
[TestMethod]
public void VerifyBrushDescription()
{
    driver.FindElement(By.PartialLinkText("Brush")).Click();
    var description = driver.FindElement(By.CssSelector("#product-13 > div.row > div.col-md-7.product-summary-case > div > h1")).Text;
    Assert.AreEqual("Makeup Brush", description);
    driver.FindElement(By.CssSelector("# product-13 > div.row > div.col-md-7.product-summary-case > div > form > div > input.plus")).Click();
    driver.FindElement(By.CssSelector("#product-13 > div.row > div.col-md-7.product-summary-case > div > form > button")).Click();
}
```

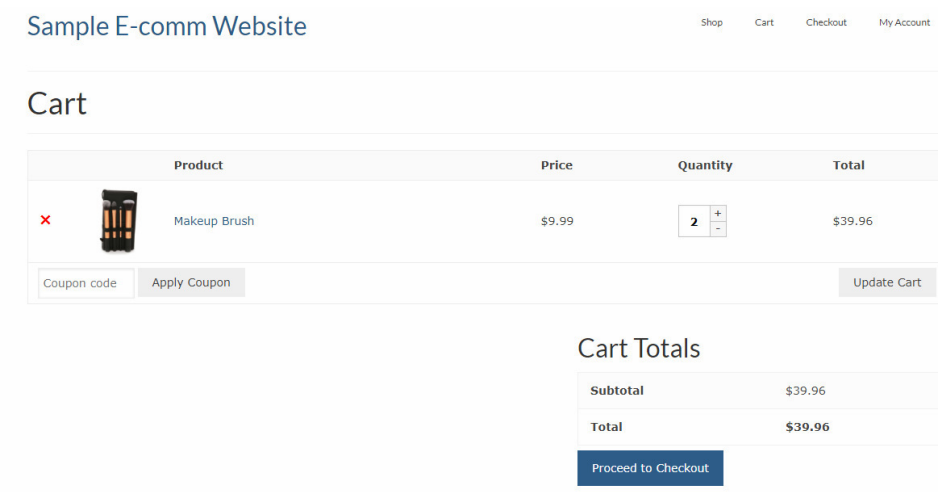
Выбираем кисть, в этот раз, используя PartialLink вместо Csssetletctor. Проверяем описание товара на соответствие.



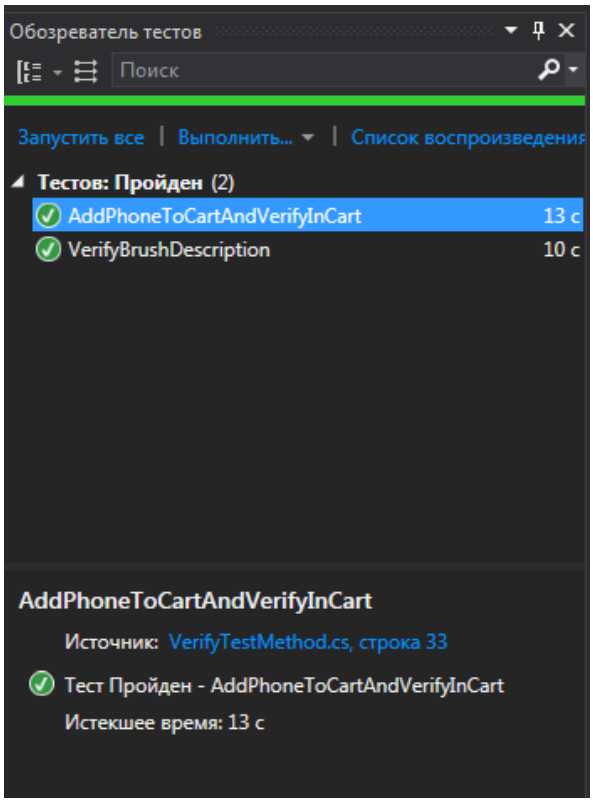
Если описание соответствует действительности, добавляем в корзину.



Просматриваем корзину. Можно проверить на соответствие количеству, аналогично 1 сценарию.

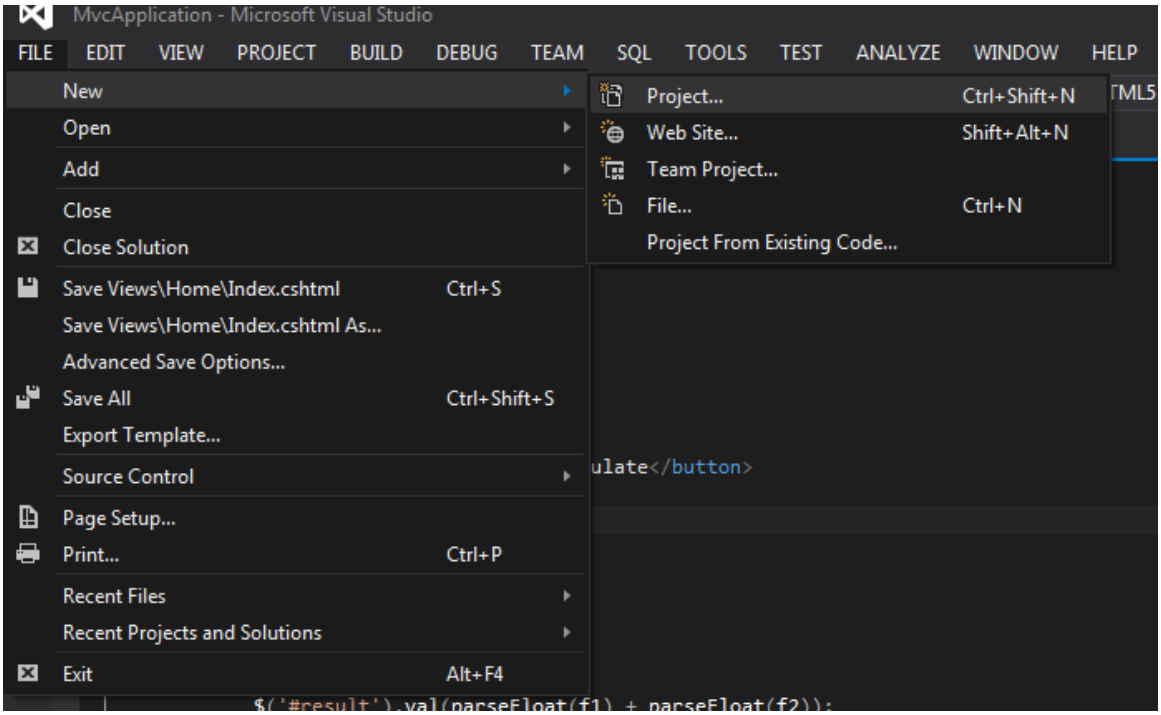


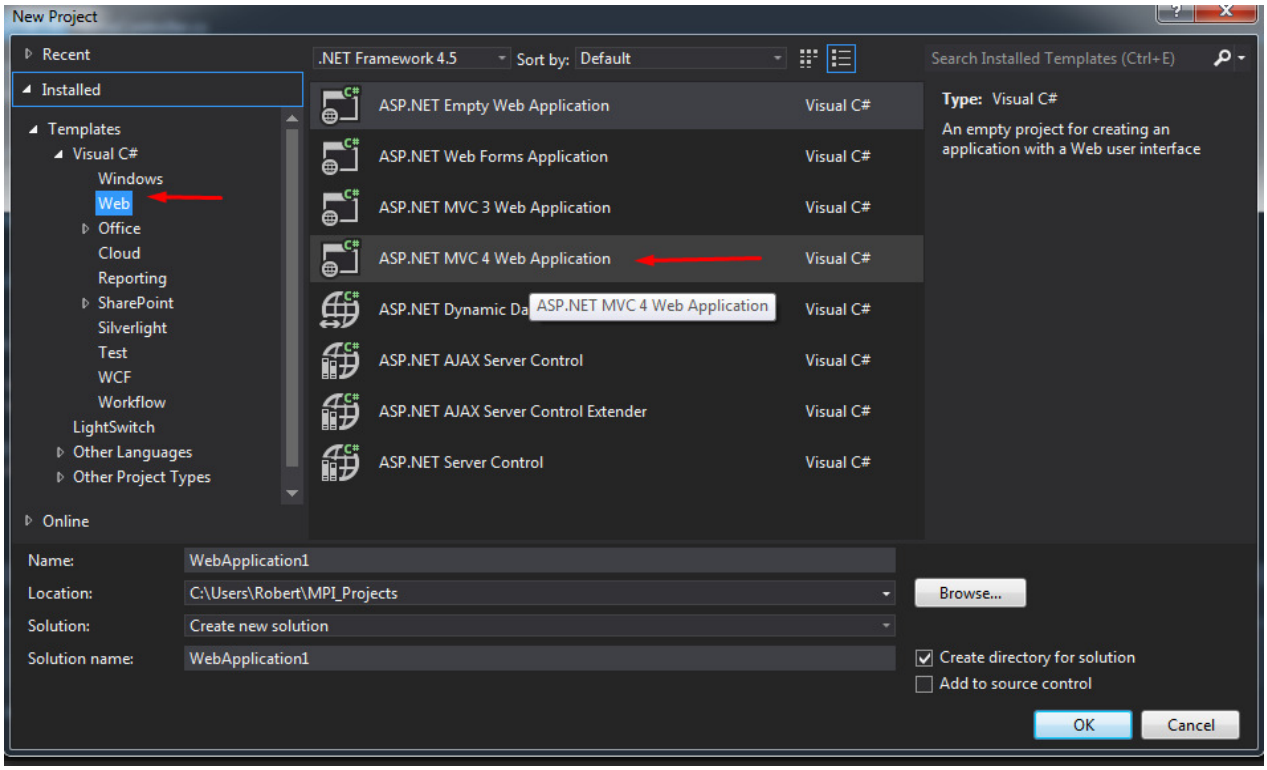
Если все тесты прошли успешно получим вот такой результат:

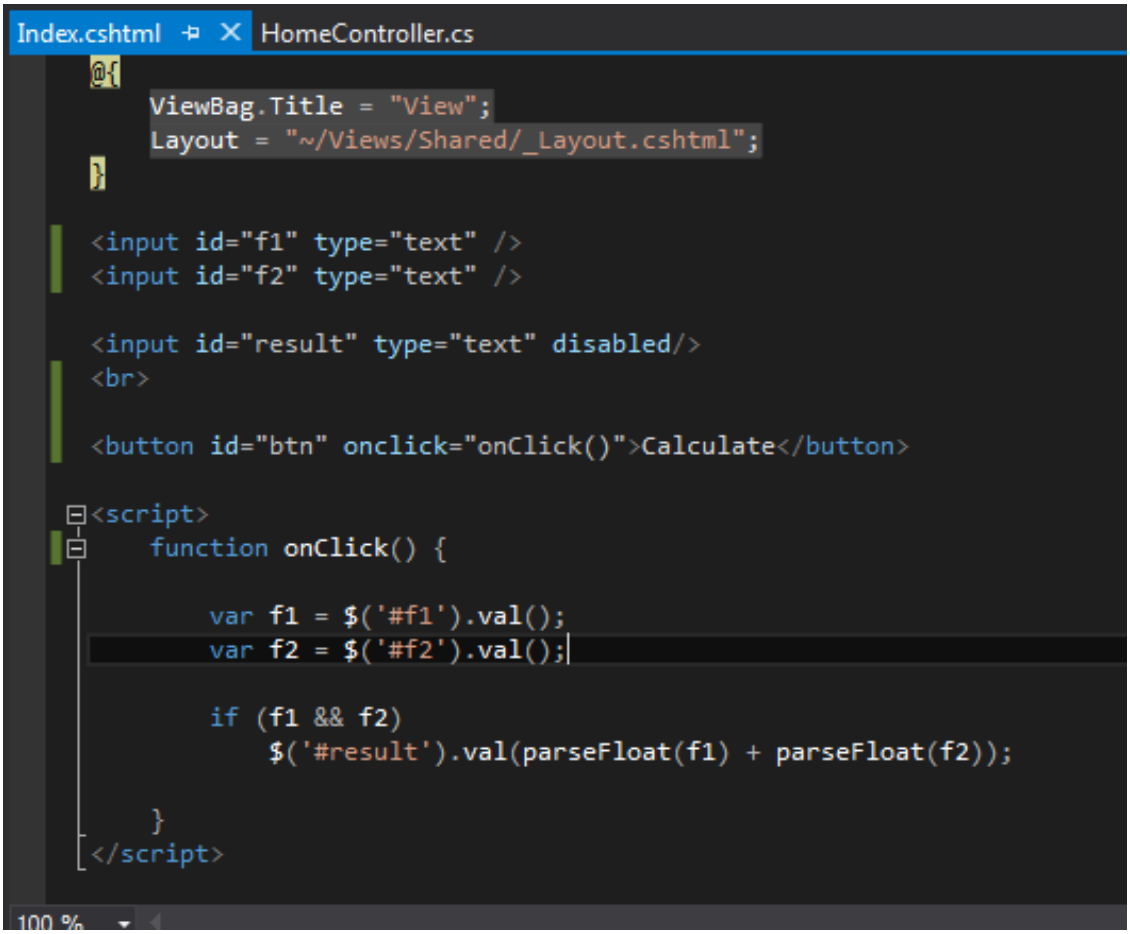


2. Проектирование автоматических тестов с использованием Selenium IDE для тестирования веб-сервисов

Веб сервис. Создадим учебное веб-приложение ASP.NET MVC 4. В нем мы используем единственную страницу, на которой будет выполнено сложение двух чисел.

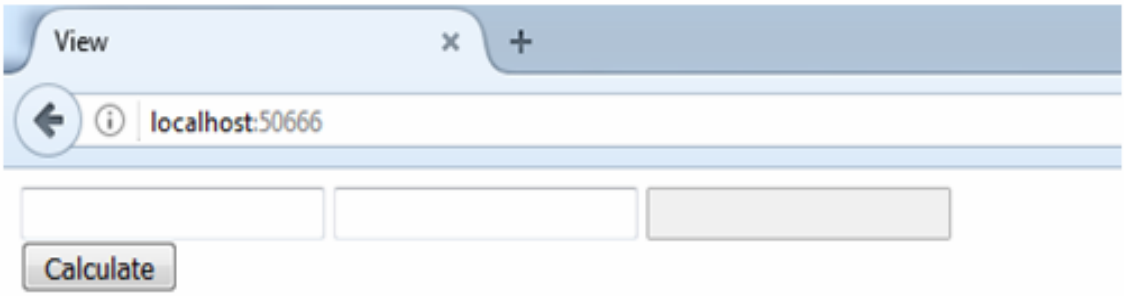



New Project
Recent
Installed
Templates
Visual C#
Windows
Web
Office
Cloud
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
LightSwitch
Other Languages
Other Project Types
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
ASP.NET Empty Web Application
ASP.NET Web Forms Application
ASP.NET MVC 3 Web Application
ASP.NET MVC 4 Web Application
ASP.NET Dynamic Da
ASP.NET MVC 4 Web Application
ASP.NET AJAX Server Control
ASP.NET AJAX Server Control Extender
ASP.NET Server Control
Visual C#
Type: Visual C#
An empty project for creating an application with a Web user interface
Name: WebApplication1
Location: C:\Users\Robert\MPI_Projects
Solution: Create new solution
Solution name: WebApplication1
Browse...
Create directory for solution
Add to source control
OK
Cancel


Index.cshtml
HomeController.cs
@{
ViewBag.Title = "View";
Layout = "~/Views/Shared/_Layout.cshtml";
}
<input id="f1" type="text" />
<input id="f2" type="text" />
<input id="result" type="text" disabled/>

<button id="btn" onclick="onClick()">Calculate</button>
<script>
function onClick() {
var f1 = $('#f1').val();
var f2 = $('#f2').val();
if (f1 && f2)
$('#result').val(parseFloat(f1) + parseFloat(f2));
}
</script>
100 %

При запуске это будет выглядеть так:



Вводим числа, нажимаем на кнопку – получим ответ. Далее нам нужен **Браузер Firefox**.

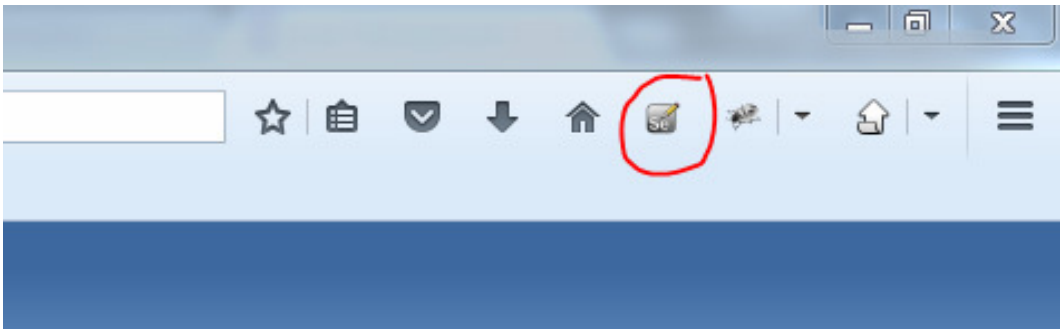
Необходимо загрузить инсталлятор Firefox (последнюю доступную стабильную версию) и установить Firefox, следуя инструкциям инсталлятора (рекомендуется везде оставить предлагаемые по умолчанию настройки)

Selenium IDE нужен для того, чтобы записывать действия, производимые в браузере: нажатия, ввод, переход по ссылкам и т.д. Селениум записывает абсолютно все действия пользователя. После записи, селениум может транслировать эту запись в код на любом имеющемся в нем языке, в нашем случае – C#. Этот код мы используем для автоматизации тестирования: для имитации действия пользователя.

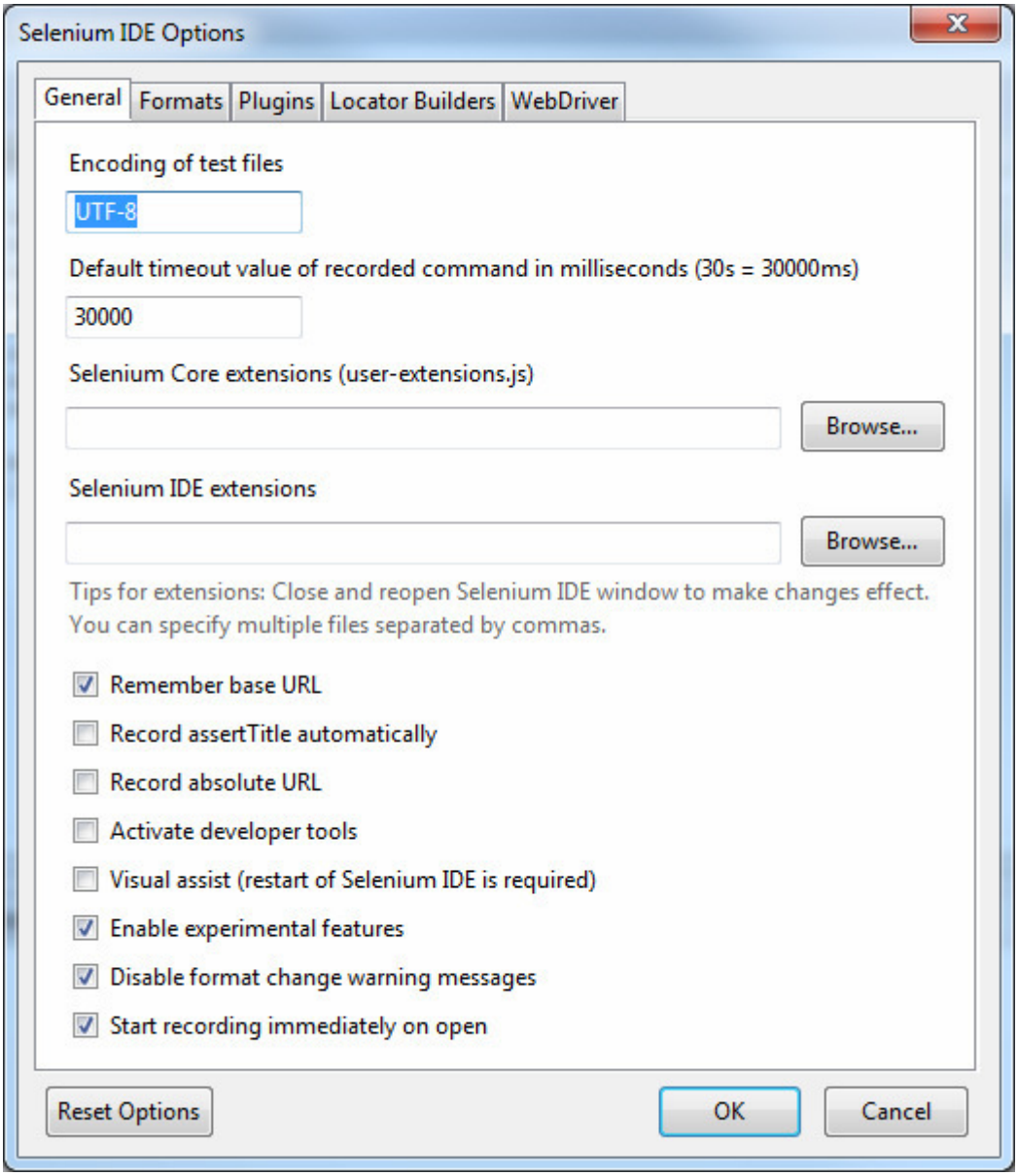
Установить в Firefox плагин Selenium IDE: зайти на страницу https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/

Первые шаги

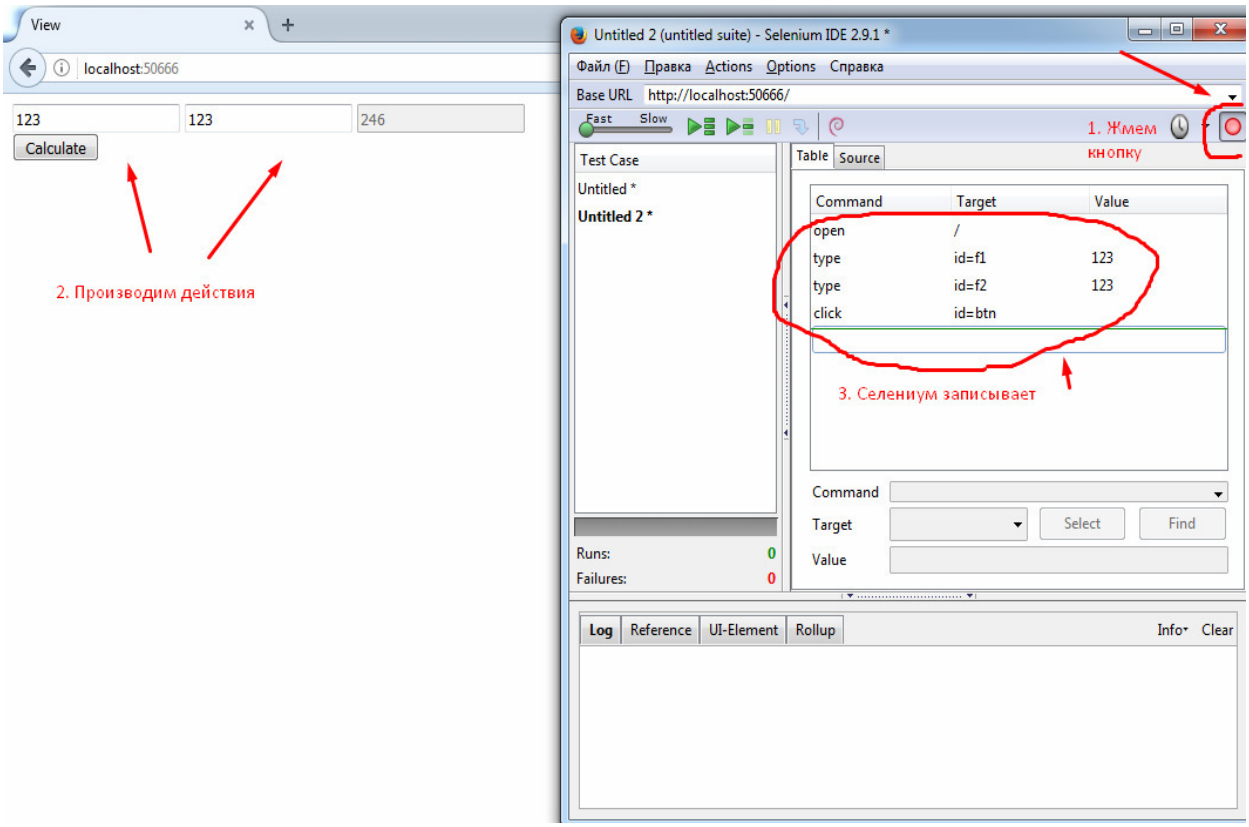
1. Справа сверху доступен Selenium IDE. Запустим его:



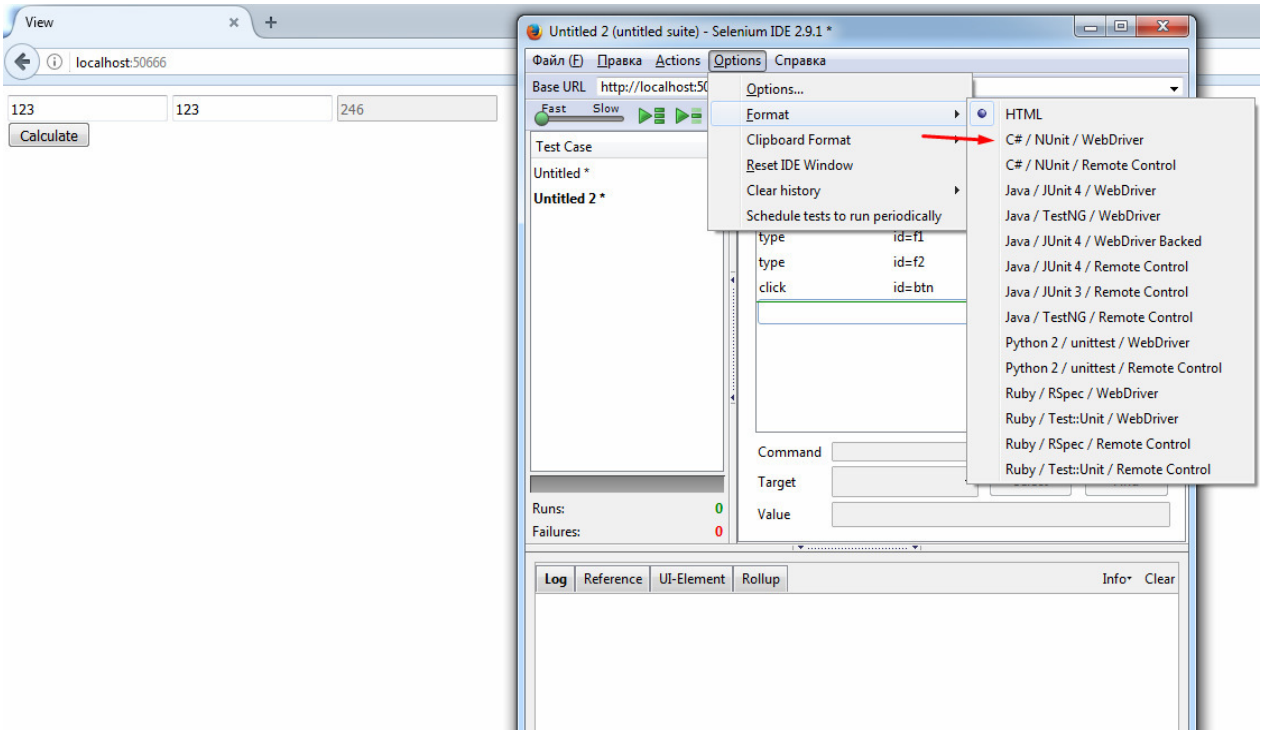
В меню Options -> Options проставьте 2 предпоследних чекбокса:



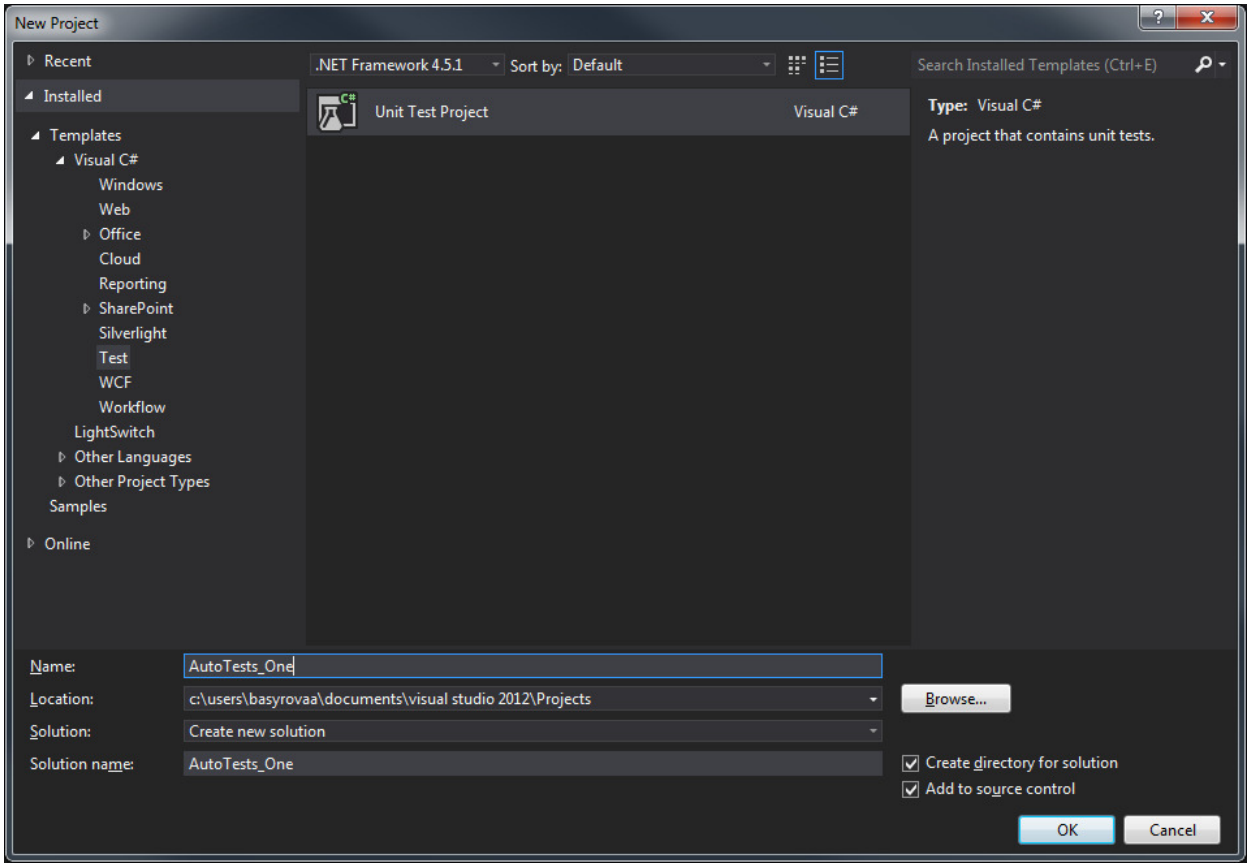
Перейдем к записи действий. Стартуем веб-сервис и жмем на кнопку записи в селениуме. Далее введем два числа и нажмем кнопку Calculate.



Чтобы получить из записанного код на языке программирования, выбираем Options -> Format -> C#/Nunit/Webdriver. Теперь этот код можно использовать.



2. В программе Visual Studio выбираем File -> New -> Project. Выбираете Visual C# -> Test-> Unit Test Project. Вводите имя своего проекта и Ok.



Стираете все, что есть по умолчанию и вставляете то, что есть в Selenim IDE (код на C#).

3. Сейчас у вас почти все выделено красным. Поэтому нужно добавить библиотеки. Выбираете меню Tools -> NuGet Package Manager и внизу в командной строке по очереди запускаете команды:

Install-Package NUnit -Version 2.6.4

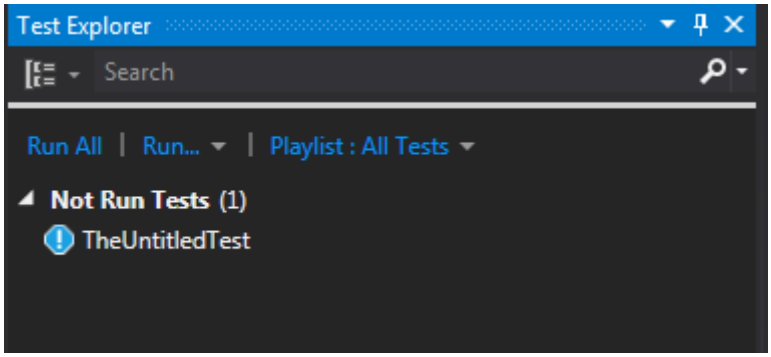
Install-Package Selenium.WebDriver

Install-Package Selenium.Support

Но этого недостаточно. Чтобы у вас тесты определялись, нужно установить адаптер. Отсюда скачиваем NUnit Test Adapter https://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d . Запускаем скачанный файл и ставим в нашу студию. Если библиотеки нужно подключать к каждому новому проекту, то адаптер

достаточно установить один раз для Visual Studio. Возможно, придется перезапустить Visual Studio.

4. Выбираем Build. Если не видно области Text Explorer с вашим тестом, то выбираете меню Test -> Windows -> Test Explorer:

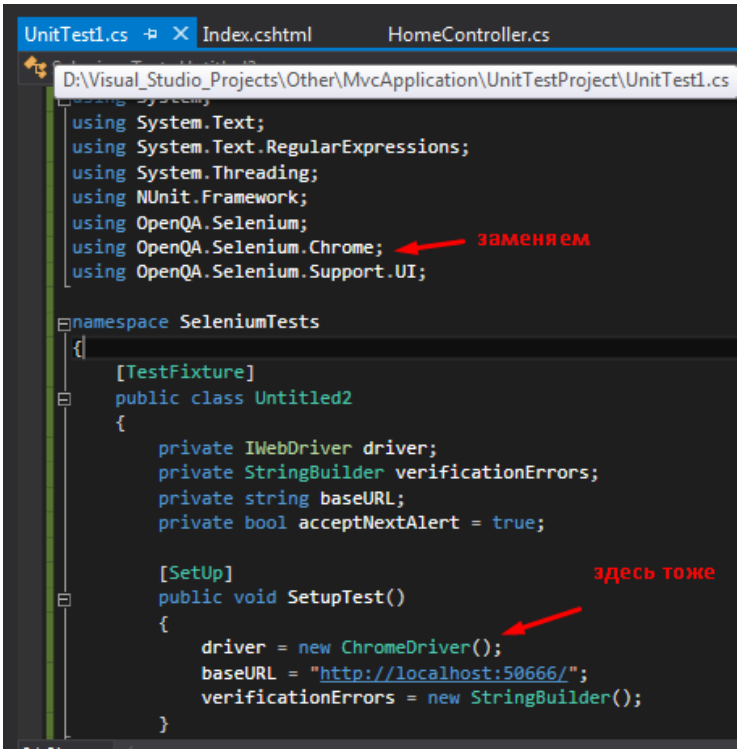


Правый клик по этому тесту и Run Selected Tests. Если ничего не произошло и появилась ошибка, значит, попробуйте запустить в Chrome. Для этого:

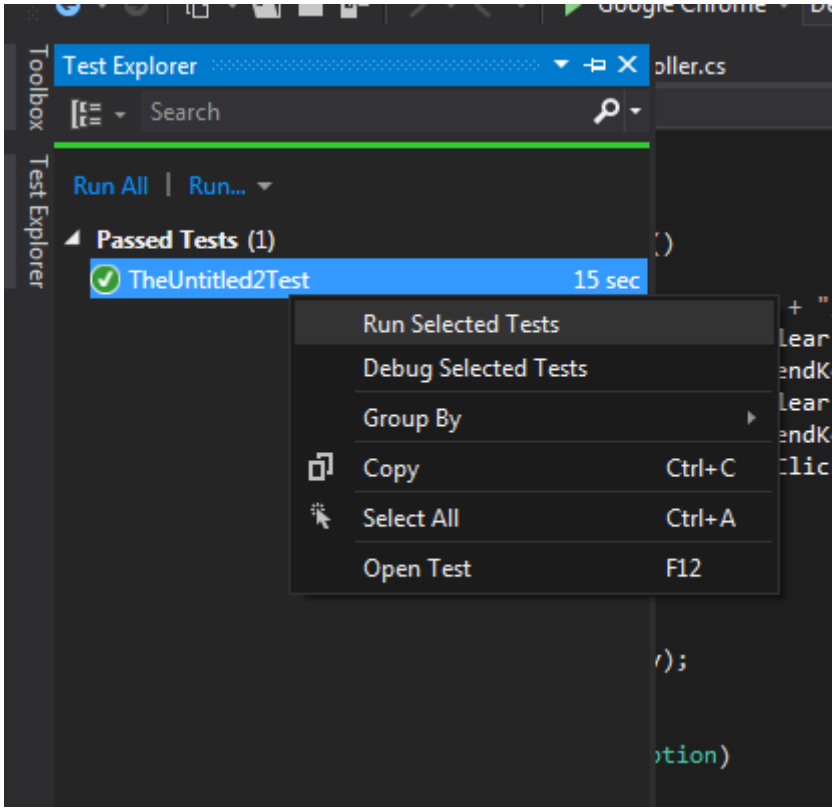
- Необходимо скачать хромдрайвер https://sites.google.com/a/chromium.org/chromedriver/downloads

- Добавить using OpenQA.Selenium.Chrome;

- Заменить driver = new FirefoxDriver() на driver = new ChromeDriver(@"D:\distr"); где в скобках указан путь к драйверу



Запуск теста



Открывается браузер и выполняются записанные действия. Тест просто записывает значения и нажимает на кнопку. Давайте теперь добьемся того, чтобы программа проверяла, верен ли ответ.

Для этого поправим код, который сгенерировал селениум.

```
[Test]
public void TheUntitled2Test()
{
    driver.Navigate().GoToUrl(baseURL + "/");
    driver.FindElement(By.Id("f1")).Clear();
    driver.FindElement(By.Id("f1")).SendKeys("123");
    driver.FindElement(By.Id("f2")).Clear();
    driver.FindElement(By.Id("f2")).SendKeys("123");
    driver.FindElement(By.Id("btn")).Click();
    var result = driver.FindElement(By.Id("result")).GetAttribute("value");
    Assert.AreEqual(int.Parse(result), 246, "sum incorrect!");
}
private bool IsElementPresent(By by)
{
    try
    {
```

Теперь наш тестовый метод проверяет, верно ли посчитано значение.

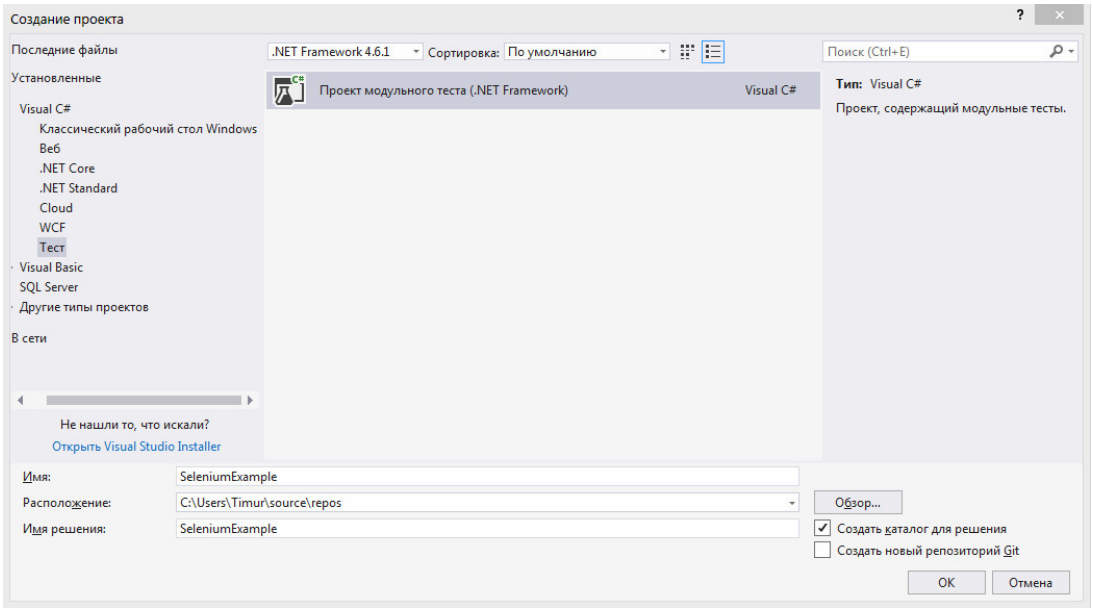
3. Тестирование web-приложения при помощи Selenium

Подготовка окружения

Для упрощения тестирования можно воспользоваться записью сценариев действий в веб-браузере, для этого Selenium предоставляет расширение для браузера Selenium IDE, ссылка для скачивания: https://addons.mozilla.org/en-US/firefox/addon/selenium-ide/. Для работы с данным расширением требуется браузер Firefox версией ниже Quantum (57.0). У Mozilla есть веб-сайт, содержащий старые версии Firefox, https://ftp.mozilla.org/pub/firefox/releases/ для работы была выбрана версия 42.0. По умолчанию, Firefox настроен на автоматические обновления. Чтобы предотвратить автоматическое обновление Firefox необходимо изменить настройки. Для создания проекта тестирования в работе была использована среда разработки Visual Studio 2017 community.

Создание проекта

5. В Visual Studio Файл -> Новый -> Проект. Visual C# -> Тест-> Проект модульного теста.



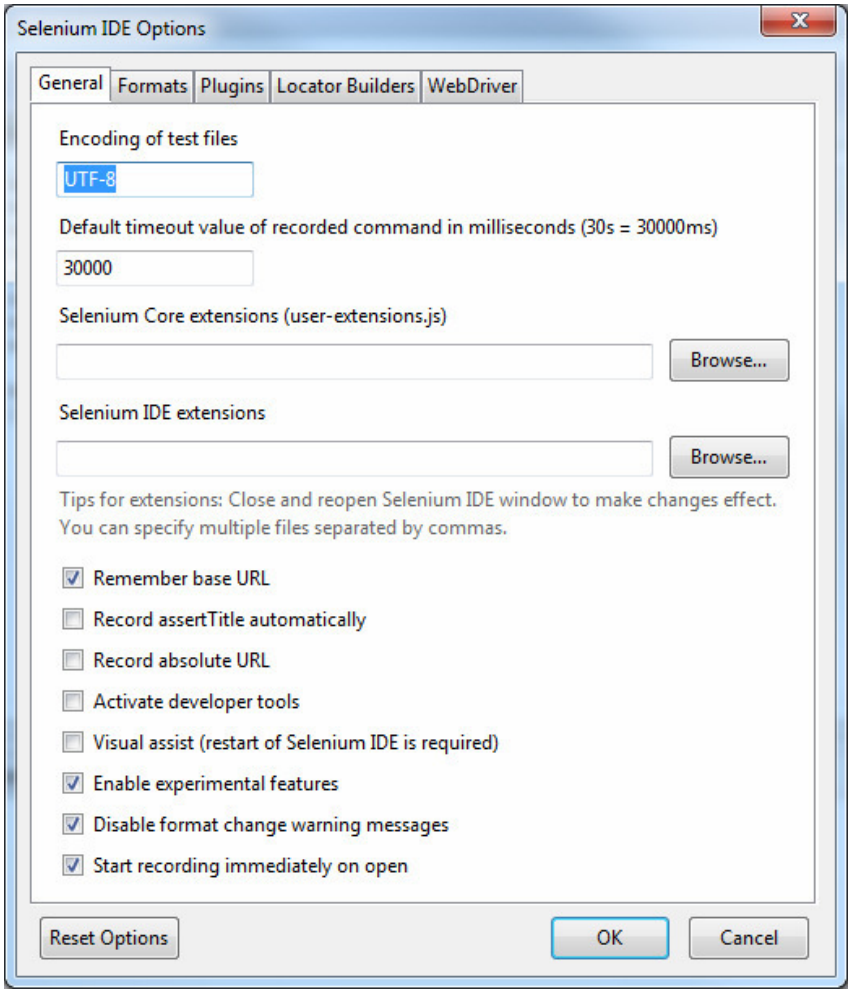
Для того, что бы начать тестирование с Selenium были установлены следующие библиотеки: NUnit-Version 2.6.4, Selenium.WebDriver, Selenium.Support. Их можно установить через NuGet Package Manager. Также потребовалось установить NUnit Test Adapter

https://visualstudiogallery.msdn.microsoft.com/6ab922d0-21c0-4f06-ab5f-4ecd1fe7175d, для установки требуется закрыть Visual Studio. Все готово для начала тестирования.

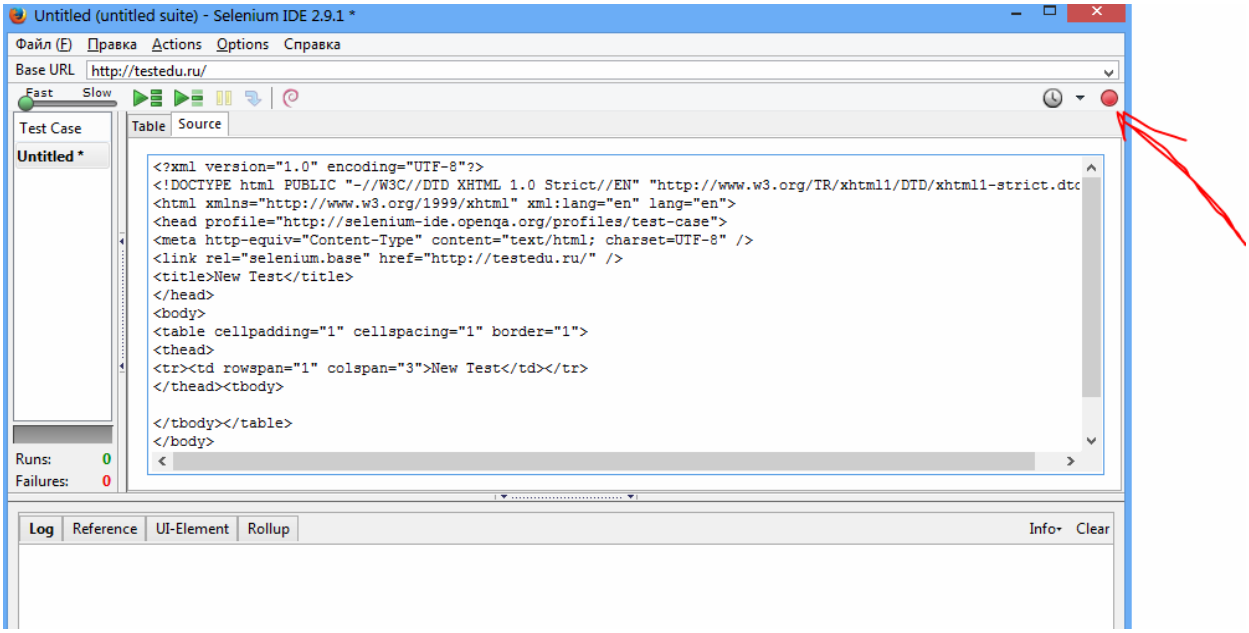
Тестирование

В качестве сайта для тестирования был выбран тест на сайте http://testedu.ru/test/english-language/2-klass/form-2-grammar-vocabulary-test-12.html.

Пройдя на указанный сайт, запустим расширение Selenium IDE. В меню Options -> Options выберем 2 предпоследних чекбокса:



Для начала записи сценария активируем кнопку записи.



Проведем тестирование, после чего отпустим кнопку записи. Чтобы получить из записанного код на языке C#, выберем Options -> Format -> C#/Nunit/Webdriver. Скопируем получившийся код в созданный на предыдущем этапе проект. Для того чтобы выполнить тест нужно открыть обозреватель тестов Тестирование -> окна -> обозреватель тестов. Откуда можно запустить написанный тест. На этапе выполнения возникла ошибка. Была решена установкой браузера chrome, подключением директивы «OpenQA.Selenium.Chrome», установкой ChromeDriver https://sites.google.com/a/chromium.org/chromedriver/downlods, сменой строки driver = new FirefoxDriver(); на driver = new ChromeDriver(@"C:\"); где "C:\" директория в которую был сохранен ChromeDriver.

Листинг программы получившегося теста:

```
using System;
using System.Text;
using NUnit.Framework;
using OpenQA.Selenium;
using OpenQA.Selenium.Chrome;


namespace SeleniumExample
{
    [TestFixture]
    public class Untitled
    {
        private IWebDriver driver;
```

```
        private StringBuilder verificationErrors;
        private string baseURL;
        private bool acceptNextAlert = true;

        [SetUp]
        public void SetupTest()
        {
            driver = new ChromeDriver(@"C:\");
            baseURL = "http://testedu.ru/";
            verificationErrors = new StringBuilder();
        }

        [TearDown]
        public void TeardownTest()
        {
            try
            {
                driver.Quit();
            }
            catch (Exception)
            {
                // Ignore errors if unable to close the browser
            }
            Assert.AreEqual("", verificationErrors.ToString());
        }

        [Test]
        public void TheUntitledTest()
        {
            driver.Navigate().GoToUrl(baseURL    +    "/test/english-
language/2-klass/form-2-grammar-vocabulary-test-12.html");
            driver.FindElement(By.Name("Q1")).Click();
            driver.FindElement(By.Name("Q2")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[7]")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[10]")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[13]")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[16]")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[19]")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[22]")).Click();
            driver.FindElement(By.Name("Q9")).Click();

driver.FindElement(By.XPath("(//span[@name='aspan'])[28]")).Click();
            driver.FindElement(By.Name("simb")).Click();
        }
        private bool IsElementPresent(By by)
```

```
        {
            try
            {
                driver.FindElement(by);
                return true;
            }
            catch (NoSuchElementException)
            {
                return false;
            }
        }

        private bool IsAlertPresent()
        {
            try
            {
                driver.SwitchTo().Alert();
                return true;
            }
            catch (NoAlertPresentException)
            {
                return false;
            }
        }

        private string CloseAlertAndGetItsText()
        {
            try
            {
                IAlert alert = driver.SwitchTo().Alert();
                string alertText = alert.Text;
                if (acceptNextAlert)
                {
                    alert.Accept();
                }
                else
                {
                    alert.Dismiss();
                }
                return alertText;
            }
            finally
            {
                acceptNextAlert = true;
            }
        }
    }
}
```

4. Selenium WebDriver – работа с диалоговыми окнами

Рассмотрим методы Selenium WebDriver для работы с диалоговыми окнами. Окно сообщения, которое выводится, является *модальным окном*. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберётся с окном. В данном случае – пока не нажмёт на «OK».

Приведем пример базовых UI операций: alert, prompt и confirm, которые позволяют работать с данными, полученными от пользователя.

alert

Синтаксис: **alert(сообщение)**

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмёт «OK».

Пример: **alert("Привет");**

prompt

Функция **prompt** принимает два аргумента:

result = prompt(title, default);

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием Esc на клавиатуре.

Вызов **prompt** возвращает то, что ввёл посетитель – строку или специальное значение **null**, если ввод отменён.

Единственный браузер, который не возвращает null при отмене ввода – это Safari. При отсутствии ввода он возвращает пустую строку. Предположительно, это ошибка в браузере.

Если нам важен этот браузер, то пустую строку нужно обрабатывать точно так же, как и null, т.е. считать отменой ввода.

Как и в случае с alert, окно promt модальное.

```
var years = prompt('Сколько вам лет?', 100);
alert('Вам ' + years + ' лет!')
```

Всегда указывайте default

Второй параметр может отсутствовать. Однако при этом IE ставит в диалог значение по умолчанию "undefined".

Запустите этот код в IE:

var test = prompt("Тест");

Поэтому рекомендуется *всегда* указывать второй аргумент:

var test = prompt("Тест", '');

confirm

Синтаксис:

result = confirm(question);

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL.

Результатом будет true при нажатии OK и false – при CANCEL(Esc).

Например:

var isAdmin = confirm("Вы - администратор?");

alert(isAdmin);

Особенности встроенных функций

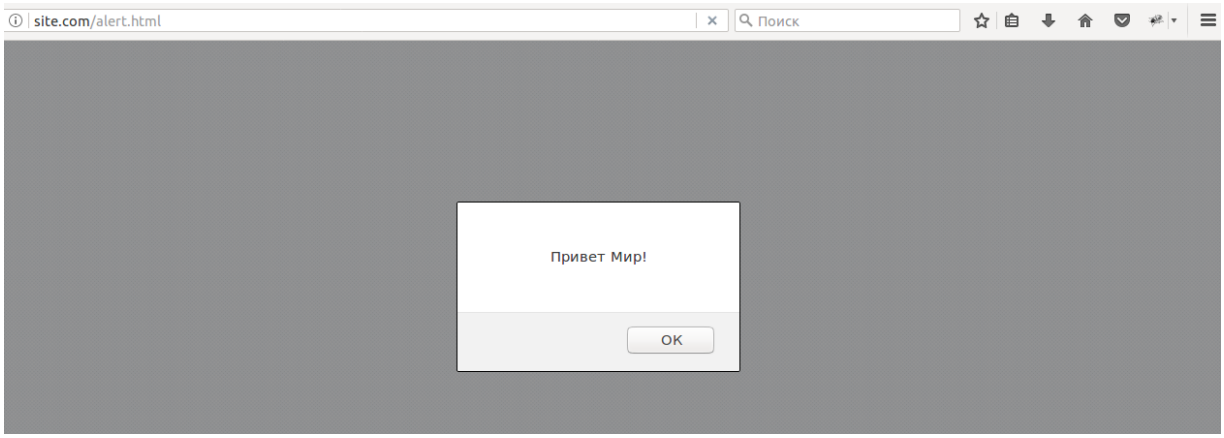
Конкретное место, где выводится модальное окно с вопросом – обычно это центр браузера, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны – это недостаток, так как нельзя вывести окно в своем, особенно красивом, дизайне.

Тестирование:

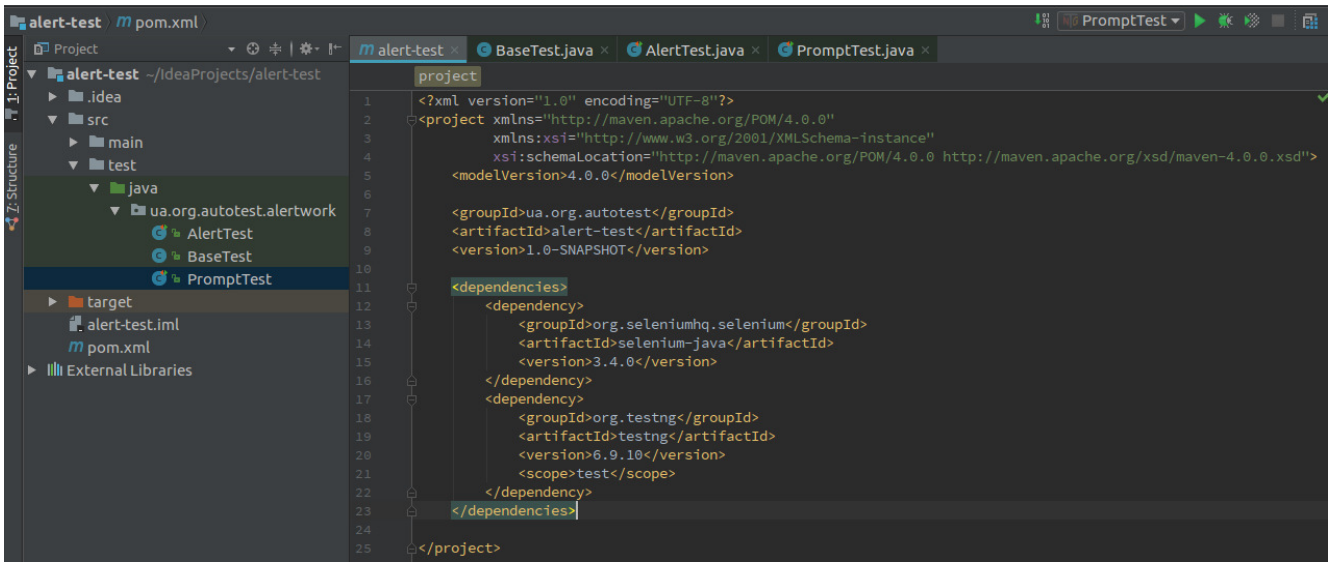
Класс Alert из Selenium содержит методы, которые работают с этими тремя видами диалогов.

Предварительно для тестов создадим простую веб-страницу и выложим ее на локальный сервер Apache2. При открытии страницы сразу же появляется диалоговое окно с приветствием:



Для перехода на страницу и продолжения каких-либо действий с ней необходимо закрыть окно. Посмотрим, как можно сделать это с помощью Selenium.

Структура maven проекта и pom файл:



В классе BaseTest создается и уничтожается экземпляр драйвера:

```
package ua.org.autotest.alertwork;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;

import java.util.concurrent.TimeUnit;

public class BaseTest {

    protected WebDriver driver;

    protected WebDriver getDriver() {
        System.setProperty("webdriver.chrome.driver",
"/home/user/drivers/chromedriver");
        driver = new ChromeDriver();
```

```
        driver.manage().window().maximize();
        driver.manage().timeouts().implicitlyWait(5, TimeUnit.SECONDS);
        return driver;
    }

    protected void tearDown() {
        if(driver != null) {
            driver.quit();
        }
    }
}
```

В классе AlertTest два теста. В первом мы получим текст диалогового окна, во втором закроем окно и проверим, что страница стала доступна:

```
package ua.org.autotest.alertwork;

import org.openqa.selenium.Alert;
import org.openqa.selenium.By;
import org.testng.Assert;

import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class AlertTest extends BaseTest{

    Alert alert;

    @BeforeMethod
    public void goPage() {
        getDriver().get("http://site.com/alert.html");
    }

    @Test(priority = 1)
    public void alertText() {
        alert = driver.switchTo().alert();
        String actualText = alert.getText();
        String expectedText = "Привет Мир!";
        Assert.assertEquals(actualText, expectedText);
    }

    @Test(priority = 2)
    public void alertAccept() {
        alert = driver.switchTo().alert();
        alert.accept();
        Assert.assertTrue(driver.findElement(By.tagName("p")).isDisplayed
());
    }
```

```
    @AfterMethod
    public void quitDriver() {
        tearDown();
    }
}
```

Для переключения и начала работы с alert необходимо создание экземпляра класса Alert:

```
Alert alert  = driver.switchTo().alert();
```

Теперь для получения текста в диалоговом окне нужно вызвать метод getText():

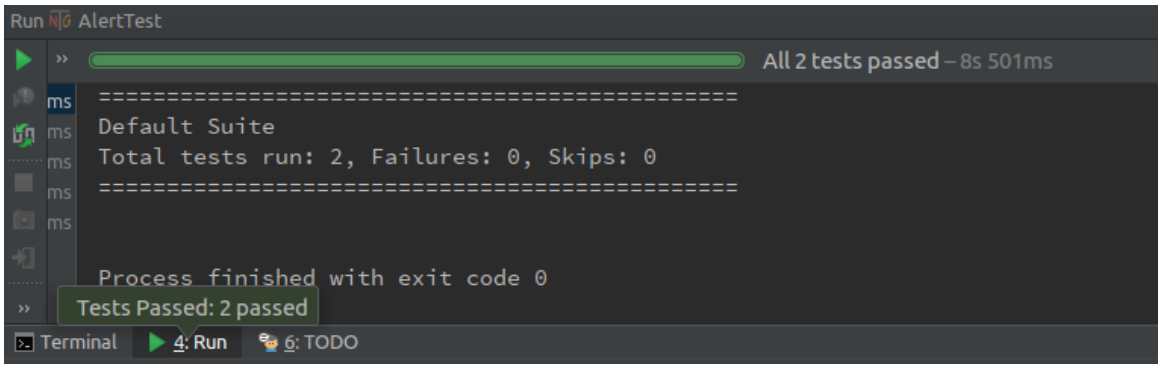
```
alert.getText();
```

Присвоив полученный текст в переменную, с помощью assertEquals() можно убедиться, что фактический и ожидаемый текст эквивалентны.

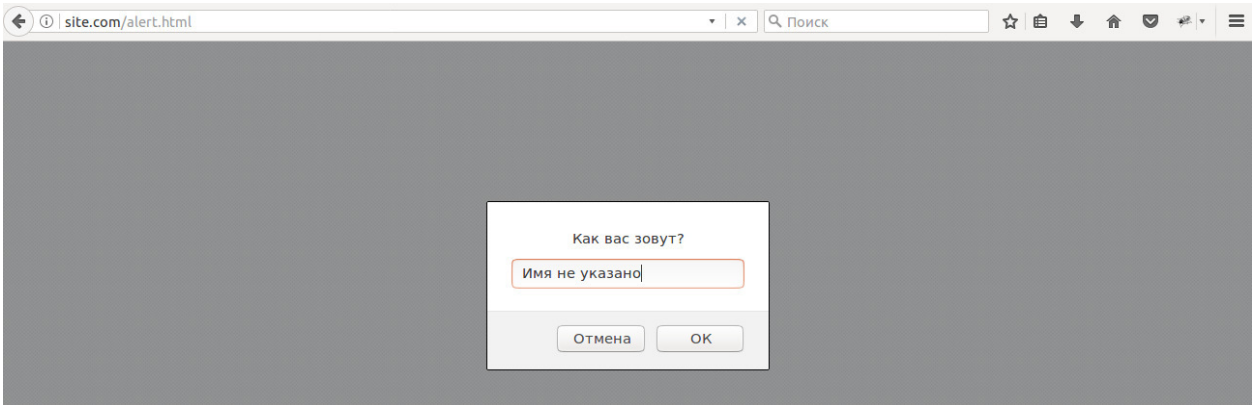
Во втором тесте после переключения в alert вызывается метод accept():

```
alert.accept();
```

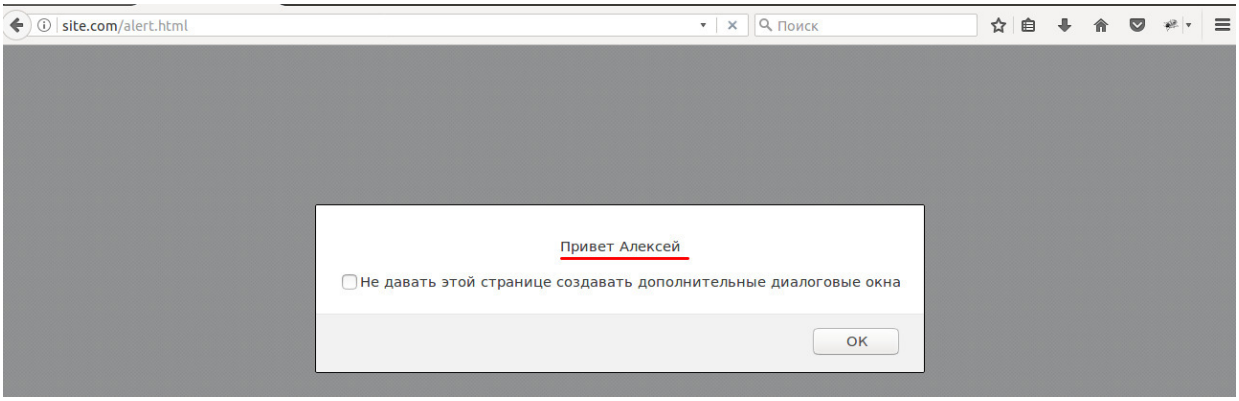
Этот метод соответствует нажатию кнопки «Ок» в диалоговом окне. Затем, чтобы убедиться, что alert действительно закрылся, с помощью assertTrue() делаем проверку доступности элемента страницы. Запускаем тест:



Тесты пройдены. Переходим к классу PromptTest. В нем мы будем тестировать диалоговое окно, которое выводит функция JavaSkript prompt. Это диалоговое окно имеет заголовок, строку для ввода а также две кнопки «Отмена» и «Ок». Внесены некоторые изменения в код тестовой веб-страницы и теперь при запуске она выглядит так:



После ввода имени в строку и нажатия «Ок» должно открыться следующее модальное окно с приветствием пользователя по введенному имени:



Сделаем это с помощью Selenium WebDriver и методов класса Alert. Код тестового класса PromptTest:

```
package ua.org.autotest.alertwork;

import org.openqa.selenium.Alert;
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class PromptTest extends BaseTest{

    Alert alert;

    @BeforeMethod
    public void goPage() {
        getDriver().get("http://site.com/alert.html");
    }

    @Test(priority = 1)
    public void inputTextPrompt() {
        alert = driver.switchTo().alert();
        alert.sendKeys("Алексей");
        alert.accept();
```

```
        String actualText = alert.getText();
        String expectedText = "Привет Алексей";
        Assert.assertEquals(actualText, expectedText);
    }

    @Test(priority = 2)
    public void alertDismiss() {
        alert = driver.switchTo().alert();
        alert.dismiss();
        String actualText = alert.getText();
        String expectedText = "Привет null";
        Assert.assertEquals(actualText, expectedText);
    }

    @AfterMethod
    public void quitDriver() {
        tearDown();
    }
}
```

Для переключения в окно возвращаемое функцией prompt как и в случае с alert нужно вызвать:

driver.switchTo().alert();

Чтобы ввести текст в строку окна нужно вызвать метод sendKeys() со строкой текста в качестве параметра:

alert.sendKeys("Алексей");

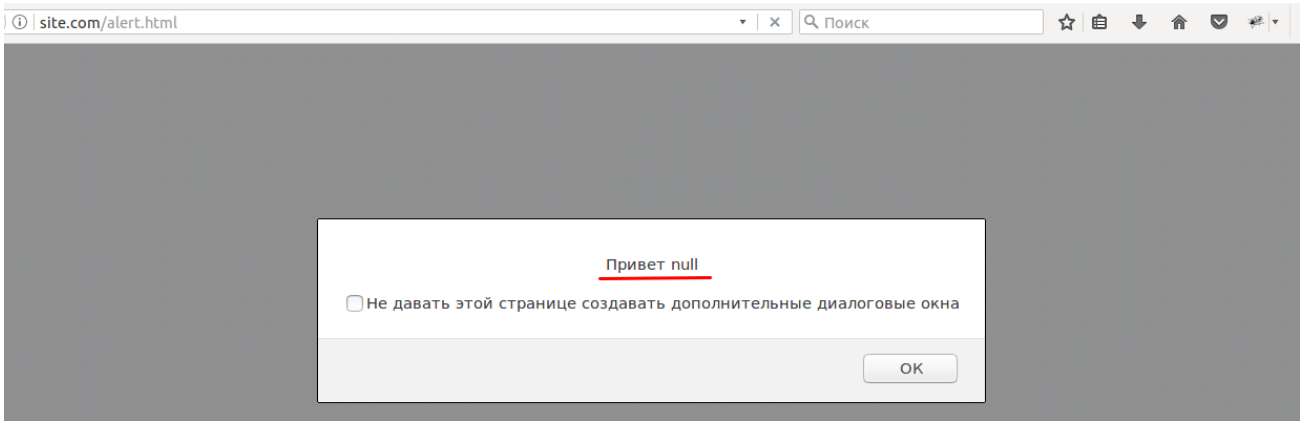
Нажимаю «Ок»:

alert.accept();

После чего открывается следующее модальное окно, из которого извлекаем текст и проверяем, что он соответствует ожидаемому:

```
String actualText = alert.getText();
String expectedText = "Привет Алексей";
Assert.assertEquals(actualText, expectedText);
```

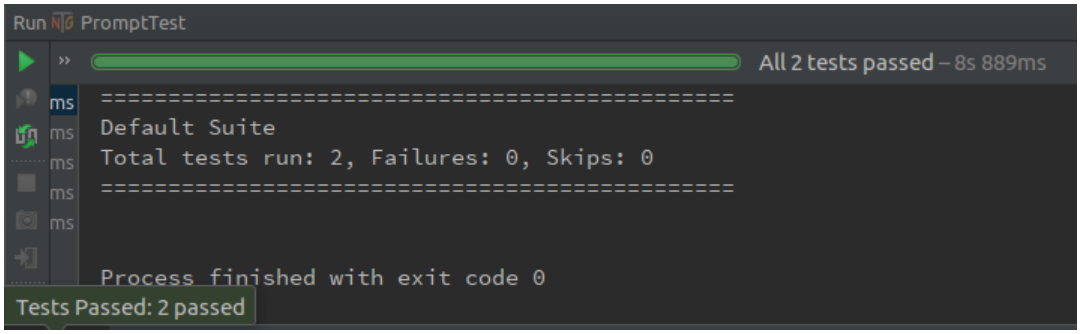
В втором тесте проверим работу еще одного метода из класса Alert-dismiss(). Он соответствует нажатию на кнопку «Отмена». Так как имя не было введено, то функция prompt возвращает null и в следующем окне приветствие выглядит так:



Что и будет проверено с помощью assertEquals():

```
alert.dismiss();
String actualText = alert.getText();
String expectedText = "Привет null";
Assert.assertEquals(actualText, expectedText);
```

Запускаем тесты класса PromptTest:



Подводя итог, перечислим рассмотренные методы класса Alert для работы Selenium WebDriver с диалоговыми окнами:

- driver.switchTo().alert(); − переключение в диалоговое окно.
- alert.accept(); − то же, что и нажатие кнопки «Ок».
- alert.dismiss(); − соответствует нажатию кнопки «Отмена» или «Cancel».
- alert.sendKeys(); − ввод текста в текстовое поле диалогового окна.
- alert.getText();− получение заголовка текстового окна.

Если вы хотите протестировать работу тестов, но не хотите заниматься установкой Apache2, то можете просто создать файл с расширением html и в метод get(String url) передавать не адрес страницы, а путь к файлу. Например: getDriver().get("file:///home/user/eclipse-workspace/test/test.html");

Приведем код тестовых страниц, которые были использованы.

alert:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8">
 </head>
 <body>
  <script>
   alert('Привет Мир!')
   </script>
   <h1><p align="center">SITE.COM</p></h1>
 </body>
</html>
```

prompt:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8">
 </head>
 <body>
  <script>
   var name = prompt('Как вас зовут?', 'Имя не указано');
   alert('Привет ' + name);
   </script>
   <h1><p align="center">SITE.COM</p></h1>
 </body>
</html>
```

5. Тестирование веб-приложения Stockfish с помощью Selenium

Описание веб-приложения

Веб-приложение представляет собой пользовательский интерфейс к шахматной программе Blackhorse, работающей по протоколу UCI (при желании можно использовать любую другую программу, поддерживающую этот протокол). Пользователь играет белыми фигурами без ограничения времени, программе предоставляется 10 секунд на ход. Представлена запись ходов краткой алгебраической нотацией (SAN), а также описание позиции с помощью нотации Форсайта-Эдвардса (FEN). Приложение реализовано на языках Python (серверная часть) и JavaScript (клиентская часть) с помощью библиотек Flask, python-chess, chess.js и chessboard.js.

Исходные тексты Blackhorse доступны по адресу https://github.com/hyst329/ blackhorse-engine, веб-приложения − по адресу https://github.com/hyst329/ blackhorse-web-flask.

Тестирование

Для тестирования с помощью Selenium WebDriver была выбрана библиотека flask-testing, созданная специально для интеграции Flask-приложений с модулем unittest в составе стандартной библиотеки языка Python. Для тестирования также использовалась одна из сильнейших в мире шахматных программ − Stockfish, которая в ходе тестирования имитировала пользователя, играющего белыми.

Имитация действий пользователя производилась следующим образом − считывалась FEN текущей позиции, после чего Stockfish находила лучший ход, который осуществлялся на доске путем перетаскивания (drag-and-drop) фигуры с исходного поля на конечное.

Результат тестирования в браузере Firefox показан на рисунке 1.

Рис. 1: Тестирование в браузере Firefox

Результат тестирования в браузере Chromium показан на рисунке 2.

Рис. 2: Тестирование в браузере Chromium

Различия между браузерами проявились в размере шахматной доски, используемых шрифтах и отсутствии анимации при перемещении фигур в браузере Chromium.

В ходе тестирования была выявлена ошибка в Blackhorse (рис. 3): После начальных ходов 1. e2–e4 Nb8–c6 2. d2–d4 Nc6:d4 3. Qd1:d4 e7–e5 4. Qd4:e5+ Blackhorse пыталась сделать ход 4...d7–d6, противоречащий правилам шахмат (ход не спасает от шаха черному королю, который возникает в позиции после 4-го хода белых).

Рис. 3: Возникшая ошибка

Класс TestChess представляет собой тест-кейс для Flask-сервера (LiveServerTestCase).

Метод setUp (строки 15 − 23) вызывается для первичной настройки среды (переход на нужную страницу, инициализация программ и т. п.), метод tearDown (строки 25 − 31) − при завершении (выход из браузера, освобождение ресурсов).

Метод test_chess (строки 34 − 52) представляет собой собственно тестирующий метод, осуществляющий взаимодействие Stockfish с веб-приложением.

Исходный код теста

```
import unittest
import urllib.request

from flask_testing import LiveServerTestCase
from selenium import webdriver
from flask_chess import app
import sys, os, time
import chess.uci, chess

class TestChess(LiveServerTestCase):

    def create_app(self):
        return app

    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.get(self.get_server_url())

        # navigate to chess page
        self.driver.get(self.driver.current_url + "play")

        self.board = chess.Board()
        self.engine = chess.uci.popen_engine("stockfish")

    def tearDown(self):
        self.driver.quit()
        self.engine.quit()
        if sys.platform == "win32":
            os.system("taskkill /im Blackhorse.exe")
        elif sys.platform == "linux":
            os.system("killall -e Blackhorse")


    def test_chess(self):
        response = urllib.request.urlopen(self.driver.current_url)
        self.assertEqual(response.code, 200)
        board_elem = self.driver.find_element_by_id("board")
        while 1:
            fen = self.driver.find_element_by_id(
                "fen").get_attribute("innerHTML")
            if fen:
                self.board.set_fen(fen)
            if self.board.is_game_over():
```

```
44                  break
45              self.engine.position(self.board)
46              bestmove = self.engine.go(movetime=10000)[0].uci()
47              from_sq, to_sq = bestmove[0:2], bestmove[2:4]
48              source = board_elem.find_element_by_class_name(f"square-{from_sq}").
49              target = board_elem.find_element_by_class_name(f"square-{to_sq}")
50              webdriver.ActionChains(self.driver).drag_and_drop(
51                  source, target).perform()
52              time.sleep(15)
53
54
55  if __name__ == '__main__':
56      unittest.main()
```

6. Selenium Webdriver: тестирование на мобильных браузерах

Обзор инструментов

В последнее время начало набирать обороты такое направление, как мобильная автоматизация, которая включает в себя и автоматизацию мобильных веб-приложений. Как и любое новое направление, оно породило вместе с собой множество инструментов для решения его задач. Рассмотрим некоторые из них: Ranorex, Monkey Talk и Appium. Два первых мы рассмотрим в ознакомительном плане, а на последнем остановимся подробнее, поскольку он очень активно набирает популярность.

Ranorex Automation Tools − это полноценная среда разработки, а также набор инструментов и библиотек для написания тестов. Она позволяет автоматизировать следующие виды приложений:

- Desktop
- Web
- Mobile (в том числе Mobile web)

Нас в данном случае интересует последний тип. На официальном сайте есть пример иллюстрирующий работу одновременно с Web и Web Mobile. Данная среда предоставляет следующие возможности:

1. поддержка динамически генерируемых графических элементов управления (контролов);
2. настраиваемая система поиска контролов;

3. простая поддержка тестов, основанных на данных (Data Driven Testing);

4. возможность разрабатывать свои модули (фреймворки) и использовать их при разработке тестов на C#;

5. поддержка запуска тестов на сервере Continuous Integration (TeamCity);

6. генерация информативных отчетов по результату прогона тестов;

7. возможность интеграции тестов с тест-кейсами системы тест-менеджмента (TMS);

8. простота изучения и использования тестировщиками.

Monkey Talk − это инструмент для мобильного тестирования, который служит для написания тестов под Android и iOS. В отличие от выше описанного, этот инструмент предназначен только для мобильного тестирования. Monkey Talk довольно прост в освоении благодаря подробным гайдам с пояснениями и скриншотами. Благодаря собственной IDE, с возможностью Record\Play решений, легко осваивается тестировщиками.

Плюсы:

- распространяется бесплатно;
- возможность создание тестов под 2 платформы (iOS & Android);
- использование полноценного языка высокого уровня (Java API).

Минусы:

- необходимость исходников тестируемого приложения;
- нельзя использовать привычные локаторы, такие как CSS и Xpath (использует собственные).

Установка и настройка Appium. Принципы и основы работы с инструментом

Appium – инструмент автоматизации мобильных приложений, использующих Webdriver API. Appium – HTTP сервер, который создает и управляет сессиями Webdriver.

Если мы хотим использовать Appium на Windows, то нам нужно воспользоваться Appium.exe.

Установка:

1. с помощью Appium.exe.
2. Используя NPM:
   - устанавливаем NPM: C:\node> npm install express -g (C:\Node - место установки node.js);
   - запускаем NPM и выполняем команду: $ npm install appium.

Настройка:

1. в первую очередь мы устанавливаем Node.js (выше 0.10 версии);
2. затем устанавливаем Android SDK, с поддержкой API Level 17 или выше. Создаем переменную окружения ANDROID_HOME, куда добавляем пути к папкам tools и platform-tools;
3. устанавливаем Java JDK и прописываем к нему пути в переменной JAVA_HOME;
4. далее нам нужно установить Apache Ant. Так же добавляем путь к его папке в переменную окружения – PATH;
5. после чего устанавливаем Apache Maven. Создаем две переменные M2HOME и M2, куда соответственно вписываем пути к папке Maven и папке bin, внутри неё.

Запуск тестов на десктоп и мобильных браузерах

Для запуска тестов с помощью Appium нужно в начале настроить драйвер, а для этого нужно ему выставить правильные свойства.

Для запуска тестов под Android свойства драйвера будут следующие:

```
public class AndroidTest {

    private WebDriver driver;

    @Before
    public void setUp() throws Exception {
        File classpathRoot = new File(System.getProperty("user.dir"));
        File appDir = new File(classpathRoot,
"../../../apps/ApiDemos/bin");
        File app = new File(appDir, "ApiDemos-debug.apk");
```

```
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability("device","Android");
        capabilities.setCapability(CapabilityType.BROWSER_NAME, "");
        capabilities.setCapability(CapabilityType.VERSION, "4.2");
        capabilities.setCapability(CapabilityType.PLATFORM, "MAC");
        capabilities.setCapability("app", app.getAbsolutePath());
        capabilities.setCapability("app-package",
"com.example.android.apis");
        capabilities.setCapability("app-activity", ".ApiDemos");
        driver          =          new          SwipeableWebDriver(new
URL("http://127.0.0.1:4723/wd/hub"), capabilities);
    }

    //Other Test methods...
}
```

Если же мы хотим тесты на мобильное Safari, то:

```
@Before
    public void setUp() throws Exception {
        DesiredCapabilities capabilities = new DesiredCapabilities();
        capabilities.setCapability("device", "iPhone Simulator");
        capabilities.setCapability("version", "6.1");
        capabilities.setCapability("app", "safari");
        driver           =           new           RemoteWebDriver(new
URL("http://127.0.0.1:4723/wd/hub"),
                capabilities);
        driver.manage().timeouts().implicitlyWait(30,
TimeUnit.SECONDS);
    }
```

СЛОВАРЬ ТЕРМИНОВ

Автоматизация выполнения тестов

Использование программного обеспечения (например, средств захвата/воспроизведения) для контроля выполнения тестов, сравнения полученных результатов с эталонными, установки предусловий тестов и других функций контроля тестирования и организации отчетов.

Автоматизация тестирования

Использование программного обеспечения для осуществления или помощи в проведении определенных тестовых процессов, например, управление тестированием, проектирование тестов, выполнение тестов и проверка результатов.

Автоматизированное тестирование

Выполнение тестов, реализуемое при помощи заранее записанной последовательности тестов.

Актор

Пользователь, или же любое другое действующее лицо или система, взаимодействующая определенным образом с тестируемой системой.

Анализ тестируемости

Детальная проверка базиса тестирования с целью определения, является ли он достаточно качественным, чтобы выступать в роли первоисточника для процесса тестирования.

Базовый набор тестов

Набор тестовых сценариев полученных на основании внутренней структуры компонента или спецификации, предназначенный для убеждения в 100% достижении заданных критериев покрытия.

Валидация

Доказанное объективными результатами исследования подтверждение того, что требования для конкретного определенного использования приложения были выполнены. [ISO 9000]

Верификация

Доказанное объективными результатами исследования подтверждение того, что определенные требования были выполнены. [ISO 9000]

Выборочное тестирование

Разработка тестов методом черного ящика, в котором тестовые сценарии выбираются для соответствия функциональному разрезу, обычно с помощью алгоритма псевдо-случайного выбора. Этот метод может использоваться для тестирования таких нефункциональных атрибутов, как надежность и производительность.

Главный план тестирования

План тестирования, обычно охватывающий несколько уровней тестирования.

График тестирования

Список задач, действий или событий в процессе тестирования, определяющий даты и/или время их начала и завершения, и их взаимозависимости.

Дефект

Изъян в компоненте или системе, который может привести компонент или систему к невозможности выполнить требуемую функцию, например неверный оператор или определение данных. Дефект, обнаруженный во время выполнения, может привести к отказам компонента или системы.

Драйвер

Компонент программного обеспечения или средство тестирования, которое заменяет компонент, обеспечивающий управление и/или вызов компонента или системы.

Жизненный цикл программного обеспечения

Период времени, начинающийся с момента появления концепции программного обеспечения и заканчивающийся тогда, когда использование программного обеспечения более невозможно. Жизненный цикл программного обеспечения обычно включает в себя следующие этапы: концепт, описание

требований, дизайн, реализация, тестирование, инсталляция и наладка, эксплуатация и поддержка и, иногда, этап вывода из эксплуатации. Данные фазы могут накладываться друг на друга или проводиться итерационно.

Завершение тестирования

Во время фазы завершения тестирования собираются данные обо всех завершенных процессах с целью объединения опыта, тестового обеспечения, фактов и чисел. Фаза завершения тестирования состоит из архивирования тестового обеспечения и оценки процесса тестирования, включающей в себя подготовку аналитического отчета о тестировании.

Инкрементное тестирование

Тестирование, при котором компоненты или системы интегрируются и тестируются по одному или вместе до тех пор, пока все компоненты или системы не интегрированы и не протестированы.

Инспекция программного обеспечения

Это статическая проверка соответствия программы заданным спецификациями, проводится путем анализа различных представлений результатов проектирования (документации, требований, спецификаций, схем или исходного кода программ) на всех этапах жизненного цикла.

Инструмент покрытия

Средство, обеспечивающее объективное измерение того, какие структурные элементы (например, операторы или ветви) были проверены наборами тестов.

Инструмент проектирования тестов

Инструмент, упрощающий проектирование теста при помощи генерации входных данных тестов на основе спецификаций, которые могут находиться в хранилище инструмента CASE (например, инструмент управления требованиями); тестовое условие, хранящихся в памяти самого инструмента, или же на основе кода.

Интеграционное тестирование

Тестирование, выполняемое для обнаружения дефектов в интерфейсах и во взаимодействии между интегрированными компонентами или системами.

Итоговый отчет о тестировании

Документ, подводящий итог задачам и результатам тестирования, также содержащий оценку соответствующих объектов тестирования относительно критериев выхода.

Качество программного обеспечения

Совокупность функциональности и свойств программного продукта, влияющих на его способность удовлетворить сформулированные или подразумеваемые потребности.

Компонентное тестирование

Тестирование отдельных компонентов программного обеспечения [Согласно IEEE 610].

Контроль тестирования

Задача управления тестированием, связанная с разработкой и применением комплекса корректирующих мер для возвращения тестирования проекта в график при выявлении отклонений от плана.

Концепция тестирования

Изложение целей тестирования и, возможно, идей относительно процесса тестирования. Используются в исследовательском тестировании.

Метрика

Шкала измерений и метод, используемый для измерений [ISO 14598].

Мониторинг тестирования

Задача управления тестированием, связанная с периодической проверкой статуса тестирования проекта. Составляемые отчеты содержат сравнение реального состояния с запланированным.

Нагрузочный тест

Тип тестирования производительности, проводимый с целью оценки поведения компонента или системы при возрастающей нагрузке, например

количестве параллельных пользователей и/или операций, а также определения какую нагрузку может выдержать компонент или система.

Ожидания

Непременный атрибут любых UI тестов для динамических приложений.

План тестирования

Документ, описывающий цели, подходы, ресурсы и график запланированных тестовых активностей. Он определяет объекты тестирования, свойства для тестирования, задания, ответственных за задания, степень независимости каждого тестировщика, тестовое окружение, метод проектирования тестов, определяет используемые критерии входа и критерии выхода и причины их выбора, а также любые риски, требующие планирования на случай чрезвычайных обстоятельств. [IEEE 829]

Покрытие

Уровень, выражаемый в процентах, на который определенный элемент покрытия был проверен набором тестов.

Покрытие кода

Метод анализа, определяющий, какие части программного обеспечения были проверены (покрыты) набором тестов, а какие нет, например, покрытие операторов, покрытие альтернатив или покрытие условий.

Программное обеспечение

Компьютерные программы, процедуры и, возможно, соответствующая документация и данные, относящиеся к функционированию компьютерной системы [IEEE Std 829-2008]

Разработка тестов методом белого ящика

Процедура разработки или выбора тестовых сценариев на основании анализа внутренней структуры компонента или системы.

Разработка тестов методом черного ящика

Процедура создания и/или выбора тестовых сценариев, основанная на анализе функциональной или нефункциональной спецификации компонента или системы без знания внутренней структуры.

Регрессионное тестирование

Тестирование уже протестированной программы, проводящееся после модификации для уверенности в том, что процесс модификации не внес или не активизировал ошибки в областях, не подвергавшихся изменениям. Проводится после изменений в коде программного продукта или его окружения.

Сертификация

Процесс подтверждения того, что компонент, система или лицо отвечает предъявляемым требованиями, например, посредством сдачи экзамена.

Тестирование программного обеспечения

Процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом [ISO/IEC TR 19759:2005].

Управление качеством ПО

Это система методов, средств и видов деятельности, направленных на выполнение требований участников проекта к качеству самого проекта и его продукции.

Функциональное тестирование

Тестирование, основанное на анализе спецификации функциональности компонента или системы.

Эталонный тест

(1). Стандарт, согласно которому может производиться измерение или сравнение. (2). Тест, который может использоваться для сравнения компонентов или систем друг с другом или на соответствие стандарту, указанному в (1). [Согласно IEEE 610]

SELENIUM

Это проект, в рамках которого разрабатывается серия программных продуктов с открытым исходным кодом.

Selenium Grid

Это кластер, состоящий из нескольких Selenium-серверов.

Selenium IDE

Плагин к браузеру Firefox, который может записывать действия пользователя.

Selenium RC

Это предыдущая версия библиотеки для управления браузерами.

Selenium Server

Это сервер, который позволяет управлять браузером с удалённой машины, по сети.

Selenium Webdriver

Инструмент для автоматизации реального браузера, как локально, так и удаленно, наиболее близко имитирующий действия пользователя.

СПИСОК ЛИТЕРАТУРЫ

1. Бейзер Б. Тестирование черного ящика. СбП: Питер, 2004.
2. Липаев В.В. Тестирование программ. М., Радио и связь, 1986.
3. Липаев В.В. Методы обеспечения качества крупномасштабных программных средств. М., Синтег, 2003.
4. Майерс Г. Искусство тестирования программ / Пер. с англ. под ред. Б. А. Позина. – М.: Финансы и статистика, 1982. – 176 с.
5. Майерс Г. Надежность программного обеспечения. М : Мир, 1980.
6. Канер С., Фолк Дж., Нгуен Е.К. Тестирование программного обеспечения. М: Диасофт, 2000.
7. Коберн А. Современные методы описания требований к системам. М.: Лори, 2002.
8. Коломейченко, А.С. Информационные технологии: учеб. пособие / А.С. Коломейченко, Н.В. Польшакова, О.В. Чеха. – Санкт-Петербург: Лань, 2018. – 228 с. – Режим доступа: https://e.lanbook.com/book/101862.
9. Леффингуэлл Д., Уидриг Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход. М.: Вильямс, 2002.
10. Мессарош Д. Шаблоны тестирования xUnit. М.: Вильямс, 2008.
11. Оценка качества программного обеспечения: Практикум: Учебное пособие / Б.В. Черников, Б.Е. Поклонов; Под ред. Б.В. Черникова – М.: ИД ФОРУМ: НИЦ Инфра-М, 2012. – 400 с.: ил.; 60x90 1/16. – (Высшее образование). – ISBN 978-5-8199-0516-6 http://znanium.com/catalog.php?bookinfo=315269
12. Петрушин, В.Н. Информационная чувствительность компьютерных алгоритмов [Электронный ресурс]: учеб. пособие / В.Н. Петрушин, М.В. Ульянов. – Электрон. дан. – Москва: Физматлит, 2010. – 224 с. – Режим доступа: https://e.lanbook.com/book/2275.
13. Стандартизация, сертификация и управление качеством программного обеспечения: Учебное пособие / Ананьева Т.Н., Новикова Н.Г., Исаев Г.Н. – М.:НИЦ ИНФРА-М, 2016. – 232 с.: 60x90 1/16. – (Высшее образование) – ISBN 978-5-16-011711-9 http://znanium.com/catalog.php?bookinfo=541003

14. Таганов, А.И. Основы идентификации, анализа и мониторинга проектных рисков качества программных изделий в условиях нечеткости [Электронный ресурс]: учеб.-метод. пособие. – Электрон. дан. – Москва: Горячая линия Телеком, 2012. – 224 с. – Режим доступа: https://e.lanbook.com/book/5244.
15. 3. Технология разработки программного обеспечения: Учеб. пос. / Л.Г.Гагарина, Е.В.Кокорева, Б.Д.Виснадул; Под ред. проф. Л.Г.Гагариной – М.: ИД ФОРУМ: НИЦ Инфра-М, 2013. – 400 с.: ил.; 60х90 1/16. – ISBN 978-5-8199-0342-1 http://znanium.com/catalog.php?bookinfo=389963
16. Управление качеством программного обеспечения: Учебник / Б.В. Черников. – М.: ИД ФОРУМ: ИНФРА-М, 2012. – 240 с.: ил.; 60х90 1/16. – ISBN 978-5-8199-0499-2 http://znanium.com/catalog.php?bookinfo=256901
17. Ascoly J. et al. Code Inspection Specification. – TR–21. 630, IBM System Communication Division, Kingston, N. Y., 1976.
18. Bansiya J., Davis C. A Hierarchical Model for Object-Oriented Quality Assessment // IEEE Transactions on Software Engineering. 2002. Vol. 28, No. 1. P. 4-17.
19. Bass L., Clements P., Kazman R. Software Architecture in Practice. 2Ed. Addison Wesley. 2003. 528 p.
20. Beizer B. Software Testing Techniques. 2-nd edition. Int. Thomson Publishing, 1990.
21. Binder R. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley, 1999.
22. Boehm B.W., Brown J.R., Kaspar H., Lipow M., MacLeod G.J., Merritt M.J.. Characteristics of Software Quality, TRW Series of Software Technology, Amsterdam, North Holland, 1978. 166 p.
23. Boehm B.W. Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1980.
24. Broy M., Jonsson B., Katoen J.-P., Leucker M., Pretschner A. Model Based Testing of Reactive Systems. LNCS 3472, Springer, 2005.

25. Brooks F.B., The mythical man-month and other essays on software engineering. Anniversary edition. University of North Carolina at Chapel Hill, 1995. Русский перевод: Брукс Ф. Мифический человеко-месяц или как создаются программные системы: Пер. с англ. – СПб.: Символ-Плюс, 1999. – 304 с.
26. Chang C., Wu C., Lin H. 2008. Integrating Fuzzy Theory and Hierarchy Concepts to Evaluate Software Quality // Software Quality Control. 2008. Vol. 16, No. 2. P. 263-267.
27. Copi I. M. Introduction to Logic. New York, Macmillan, 1968.
28. Dromey G.R. A model for software product quality // Transactions of Software Engineering. 1995. Vol. 21, No. 2. P. 146-162.
29. Fagan M. E. Design and Code Inspections to Reduce Errors in Program Development. – IBM Systems J., 1976, 15(3), p. 182–211.
30. Firesmith D. G. Common concepts underlying safety, security, and survivability engineering, Technical Note CMU/SEI-2003-TN-033, Carnegie Mellon Software Engineering Institute. 2003.
31. Freeman R. D. An Experiment in Software Development. – The Bell System Technical Journal, Special Safeguard Supplement, 1975, p. 199–209.
32. Grady R.B., Caswell D.L. Software Metrics: Establishing a Company-Wide Program. Prentice-Hall, 1987. 275 p.
33. Ghezzi C., Jazayeri M., Mandrioli D. Fundamental of Software Engineering, Prentice–Hall, NJ, USA. 1991.
34. Hetzel W.C. (ed.). Program test methods, Prentice-Hall, Inc., 1973.
35. Hyatt L.E., Rosenberg L.H. A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality // Proceedings of Product Assurance Symposium and Software Product Assurance Workshop. Noordwijk, 1996. P. 209-212.
36. IBM Developer Works [Электронный ресурс] / Diagnosing Java Code: Designing "testable" applications. / Eric E. Allen – Электрон. дан. 2001. – http://www-106.ibm.com/developerworks/java/library/j-diag0911.html, свободный. – Загл. с экрана. – яз. англ.

37. ISO/IEC 9126-1:2001. Software engineering – Software product quality – Part 1: Quality model.
38. ISO/IEC TR 9126-2:2003 Software engineering – Product quality – Part 2: External metrics.
39. ISO/IEC TR 9126-3:2003 Software engineering – Product quality – Part 3: Internal metrics.
40. ISO/IEC TR 9126-4:2004 Software engineering – Product quality – Part 4: Quality in use metrics.
41. Khosravi K., Gueheneuc Y. On Issues with Software Quality Models // Proceedings of 9th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering. 2005. P. 70-83.
42. Marick B. The Craft of Software Testing, Prentice Hall, 1995.

Utting M., Legeard B.. Practical Model-Based Testing: A Tools Approach, M-K, 2006.

43. Mathur A. P. Foundations of Software Testing. Copymat Services, 2006.
44. McCall J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Concept and Definitions of Software Quality. Final Technical Report. Vol. 1. National Technical Information Service, Springfield. 1977.
45. McCall J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Metric Data Collection and Validation. Final Technical Report. Vol. 2. National Technical Information Service, Springfield. 1977.
46. McCall J. A., Richards P. K., Walters, G. F. Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisition Manager. Final Technical Report. Vol. 3. National Technical Information Service, Springfield. 1977.
47. Myers G. J. Composite/Structured Design. New York, Van Nostrand Reinhold, 1978.
48. Myers G. J. A Controlled Experiment in Programm Testing and Code Walkthroughs/Inspections. – Commun. ACM, 1978, 21(9), p. 760–768.

49. Perriens M. P. An Application of Formal Inspections to Top-Down Structured Program Development. – RADC-TR-77-212, IBM Federal System Div., Gaithersburg, Md., 1977 (NTISAD/A-041645).
50. Sharma A., Kumar R., Grover P.S. Estimation of Quality for software components: an empirical approach // ACM SIGSOFT Software Engineering Notes. 2008. Vol. 33, No. 6. P. 1-10.
51. Shooman M. L., Bolsky M. I. Types, Distribution and Test and Correction Times for Programming Errors. – Proceedings of the 1975 International Conference on Reliable Software. New York, IEEE, 1975, p. 347–357.
52. Software Engineering Body of Knowledge, 2005. http://www.swebok.org/ironman/pdf/SWEBOK_Guide_2004.pdf.
53. Software Testing - https://software-testing.org/blog/testing/page4/
54. Selenium Documentation. https://www.seleniumhq.org/docs/index.jsp
55. Selenium WebDriver − Selenium WebDriver Documentation. Режим доступа: https://www.seleniumhq.org/projects/webdriver/ [Дата обращения: 25.04.18]
56. Selenium ChromeDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/chrome/ChromeDriver.html [Дата обращения: 22.04.18]
57. Selenium FirefoxDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/firefox/FirefoxDriver.html [Дата обращения: 20.04.18]
58. Selenium InternetExplorerDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/ie/InternetExplorerDriver.html [Дата обращения: 22.03.18]
59. Selenium EventFiringWebDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/support/events/EventFiringWebDriver.html [Дата обращения: 20.03.18]
60. Selenium HtmlUnitDriver. Режим доступа: http://seleniumhq.github.io/htmlunit-

driver/org/openqa/selenium/htmlunit/HtmlUnitDriver.html [Дата обращения: 21.03.18]

61. Selenium PhantomJSDriver. Режим доступа: https://github.com/detro/ghostdriver/blob/master/binding/java/src/main/java/org/openqa/selenium/phantomjs/PhantomJSDriver.java [Дата обращения: 10.02.18]
62. Selenium RemoteWebDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/remote/RemoteWebDriver.html [Дата обращения: 12.03.18]
63. Selenium SafariDriver. Режим доступа: https://seleniumhq.github.io/selenium/docs/api/java/org/openqa/selenium/safari/SafariDriver.html [Дата обращения: 15.02.18]
64. Weinberg G. M. The Psychology of Computer Programming. New York, Van Nostrand Reinhold, 1971.
65. Yourdon E. Techniques of program structure and design. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1975. Русский перевод: Йодан Э. Структурное проектирование и конструирование программ. – М.: Мир, 1979. – 416 с.