Министерство образования и науки Российской Федерации
Южно-Российский государственный политехнический университет
(НПИ) имени М. И. Платова

В.Я. Потапенко

ТЕСТИРОВАНИЕ И ОТЛАДКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания
для практических занятий

Новочеркасск
ЮРГПУ (НПИ)
2016

УДК 004.432(076.5)

Рецензенты:

1. Панфилов Александр Николаевич, к.т.н., доцент кафедры ИИСТ
2. Мохов Василий Александрович, к.т.н., доцент кафедры ПОВТ

Потапенко В.Я.

Тестирование и отладка программного обеспечения / : Методические указания для практических занятий / Южно-Российский государственный политехнический университет (НПИ) имени М.И.Платова. - Новочеркасск: ЮРГПУ (НПИ), 2016. – 40 с.

Содержатся основные понятия тестирования, верификации и отладки, практическая реализация различных процессов тестирования в среде VS.

Предназначены для студентов факультета информационных технологий и управления по направлению 09.03.04 «Программная инженерия».

УДК 004.432(076.5)

Практическое занятие № 1
КАЧЕСТВО И ТЕСТИРОВАНИЕ ПРОГРАММНОГО ПРОДУКТА
Теоретическая часть
Принципы тестирования программного обеспечения

Тестирование (software testing) – деятельность, выполняемая для оценки и улучшения качества программного обеспечения. Эта деятельность, в общем случае, базируется на обнаружении дефектов и проблем в программных системах.

Тестирование программных систем состоит из *динамической* верификации поведения программ на *конечном (ограниченном)* наборе тестов (set of test cases), *выбранных* соответствующим образом из обычно выполняемых действий прикладной области и обеспечивающих проверку соответствия *ожидаемому* поведению системы.

Общий взгляд на тестирование программного обеспечения последние годы активно эволюционировал, становясь все более конструктивным, прагматичным и приближенным к реалиям современных проектов разработки программных систем. Тестирование более не рассматривается как деятельность, начинающаяся только после завершения фазы конструирования. Сегодня тестирование рассматривается как деятельность, которую необходимо проводить на протяжении всего процесса разработки и сопровождения и является важной частью конструирования программных продуктов. Действительно, планирование тестирования должно начинаться на ранних стадиях работы с требованиями, необходимо систематически и постоянно развивать и уточнять планы тестов и соответствующие процедуры тестирования. Даже сами по себе сценарии тестирования оказываются очень полезными для тех, кто занимается проектированием, позволяя выделять те аспекты требований, которые могут неоднозначно интерпретироваться или даже быть противоречивыми.

В области знаний “Качество программного обеспечения” (Software Quality) техники управления качеством четко разделены на *статические (без выполнения кода)* и *динамические (с выполнением кода)*. Обе эти категории важны. Данная область знаний - “Тестирование” – касается динамических техник. Как уже отмечалось ранее, тестирование тесно связано с областью знаний “Конструирование” (Software Construction). Более того, модульное (unit-) и интеграционное тестирование все чаще рассматривают как неотъемлемый элемент деятельности по конструированию.

Цели тестирования

Тестирование — это проверка соответствия ПО требованиям, осуществляемая с помощью наблюдения за его работой в специальных, искусственно построенных ситуациях. Такого рода ситуации называют тестовыми или просто **тестами**.

Тестирование — конечная процедура. Набор ситуаций, в которых будет проверяться тестируемое ПО, всегда конечен. Более того, он должен быть настолько мал, чтобы тестирование можно было провести в рамках проекта разработки ПО, не слишком увеличивая его бюджет и сроки. Это означает, что при тестировании всегда проверяется очень небольшая доля всех возможных ситуаций. По этому поводу Дейкстра заметил, что тестирование позволяет точно определить, что в программе есть ошибка, но не позволяет утверждать, что там нет ошибок.

Тем не менее, тестирование может использоваться для достаточно уверенного вынесения оценок о качестве ПО. Для этого необходимо иметь **критерии полноты** тестирования, описывающие важность различных ситуаций для оценки качества, а также эквивалентности и зависимости между ними. Этот критерий может утверждать, что все равно в какой из ситуаций, A или B, проверять правильность работы ПО, или, если программа правильно работает в ситуации A, то, скорее всего, в ситуации B все тоже будет правильно.

Часто критерий полноты тестирования задается при помощи определения эквивалентности ситуаций, дающей конечный набор классов ситуаций. В этом случае считают, что все равно, какую из ситуаций одного класса использовать в качестве теста. Такой критерий называют **критерием тестового покрытия**, а процент классов эквивалентности ситуаций, случившихся во время тестирования, — достигнутым **тестовым покрытием**.

Таким образом, основные задачи тестирования: построить такой набор ситуаций, который был бы достаточно представителен и позволял бы завершить тестирование с достаточной степенью уверенности в правильности ПО вообще, и убедиться, что в конкретной ситуации ПО работает правильно, в соответствии с требованиями (рис. 1.1).

Рис. 1.1. Схема процесса тестирования

Тестирование — наиболее широко применяемый метод контроля качества. Для оценки многих атрибутов качества не существует других эффективных способов, кроме тестирования.

Тестировать можно соблюдение любых требований, соответствие которым выявляется во время работы ПО. Из характеристик качества по ISO 9126 этим свойством не обладают только атрибуты удобства сопровождения. Поэтому выделяют виды тестирования, связанные с проверкой определенных характеристик и атрибутов качества —

тестирование функциональности, надежности, удобства использования, переносимости и производительности, а также тестирование защищенности, функциональной пригодности и пр. Кроме того, особо выделяют **нагрузочное** или **стрессовое** тестирование, проверяющее работоспособность ПО и показатели его производительности в условиях повышенных нагрузок — при большом количестве пользователей, интенсивном обмене данными с другими системами, большом объеме передаваемых или используемых данных и пр.

Виды тестирования

На основе исходных данных, используемых для построения тестов, тестирование делят на следующие виды:

- Тестирование **черного ящика**, нацеленное на проверку требований. Тесты для него и критерий полноты тестирования строятся на основе требований и ограничений, четко зафиксированных в спецификациях, стандартах, внутренних нормативных документах. Часто такое тестирование называется тестированием **на соответствие (conformance testing)**. Частным случаем его является **функциональное** тестирование — тесты для него, а также используемые критерии полноты проведенного тестирования определяют на основе требований к функциональности.
- Еще одним примером тестирования на соответствие является **аттестационное** или **квалификационное** тестирование, по результатам которого программная система получает (или не получает) официальный документ, подтверждающий ее соответствие определенным требованиям и стандартам.
- Тестирование **белого ящика**, оно же **структурное** тестирование — тесты создаются на основе знаний о структуре самой системы и о том, как она работает. Критерии полноты основаны на проценте элементов кода, которые отработали в ходе выполнения тестов. Для оценки степени соответствия требованиям могут привлекаться дополнительные знания о связи требований с определенными ограничениями на значения внутренних данных системы

(например, на значения параметров вызовов, результатов и локальных переменных).

- Тестирование, нацеленное на определенные ошибки. Тесты для такого тестирования строятся так, чтобы гарантированно выявлять определенные виды ошибок. Полнота тестирования определяется на основе количества проверенных ситуаций по отношению к общему числу ситуаций, которые мы пытались проверить. К этому виду относится, например, тестирование **на отказ (smoke testing)**, в ходе которого просто пытаются вывести систему из строя, давая ей на вход как обычные данные, так и некорректные, с нарочно внесенными ошибками.
- Другим примером служит метод оценки полноты тестирования при помощи набора **мутантов** — программ, совпадающих с тестируемой всюду, кроме нескольких мест, где специально внесены некоторые ошибки. Чем больше мутантов не проходит успешно через данный набор тестов, тем полнее и качественнее проводимое с его помощью тестирование.

Еще одна классификация видов тестирования основана на том уровне, на который оно нацелено. Эти же разновидности тестирования можно связать с фазой жизненного цикла, на которой они выполняются.

- **Модульное** тестирование **(unit testing)** предназначено для проверки правильности отдельных модулей, вне зависимости от их окружения. При этом проверяется, что если модуль получает на вход данные, удовлетворяющие определенным критериям корректности, то и результаты его корректны. Модульное тестирование является важной составной частью **отладочного** тестирования, выполняемого разработчиками для отладки написанного ими кода.
- **Интеграционное** тестирование **(integration testing)** предназначено для проверки правильности взаимодействия модулей некоторого набора друг с другом. При этом проверяется, что в ходе совместной работы модули обмениваются данными и вызовами операций, не нарушая

взаимных ограничений на такое взаимодействие, например, предусловий вызываемых операций.

- **Системное** тестирование **(system testing)** предназначено для проверки правильности работы системы в целом, ее способности правильно решать поставленные пользователями задачи в различных ситуациях. Системное тестирование выполняется через внешние интерфейсы ПО и тесно связано с тестированием **пользовательского интерфейса** (или через пользовательский интерфейс), проводимым при помощи имитации действий пользователей над элементами этого интерфейса.

Если интеграционное и модульное тестирование чаще всего проводят, воздействуя на компоненты системы при помощи операций предоставляемого ими программного интерфейса (Application Programming Interface, API), то на системном уровне без использования пользовательского интерфейса не обойтись, хотя тестирование через API в этом случае также вполне возможно.

Проверка на моделях

Проверка свойств на моделях (model checking) [4] — проверка соответствия ПО требованиям при помощи формализации проверяемых свойств, построения формальных моделей проверяемого ПО (чаще всего в виде автоматов различных видов) и автоматической проверки выполнения этих свойств на построенных моделях. Проверка свойств на моделях позволяет проверять достаточно сложные свойства автоматически, при минимальном участии человека. Однако она оставляет открытым вопрос о том, насколько выявленные свойства модели можно переносить на само ПО (рис. 1.2).

Рис. 1.2. Схема процесса проверки свойств ПО на моделях

Обычно при помощи проверки свойств на моделях анализируют два вида свойств алгоритмов, использованных при построении ПО. **Свойства безопасности (safety properties)** утверждают, что нечто нежелательное никогда не случится в ходе работы ПО. **Свойства живучести (liveness properties)** утверждают, наоборот, что нечто желательное при любом развитии событий произойдет в ходе его работы.

В классическом подходе к проверке на моделях проверяемые свойства формализуются в виде формул так называемых **временных логик**. Их общей чертой является наличие операторов "всегда в будущем" и "когда-то в будущем".

Проверяемая программа в классическом подходе моделируется при помощи конечного автомата. Проверка, выполняемая автоматически, состоит в том, что для всех достижимых при работе системы состояний этого автомата проверяется нужное свойство. Если оно оказывается выполненным, выдается сообщение об успешности проверки, если нет — выдается трасса, последовательность выполнения отдельных шагов программы, моделируемых переходами автомата, приводящая из начального состояния в такое, в котором нужное свойство нарушается.

Задания

1. Сформулировать принципы тестирования ПО.
2. Перечислить основные цели тестирования ПО.
3. Проанализировать схему тестирования ПО.

4. Определение тестирования черного ящика.
5. Дать определение модульного тестирования.
6. Проведение интеграционного тестирования.
7. Системное тестирование ПО.
8. Тестирование на моделях.
9. Общая схема тестирования на моделях.

Практическое занятие № 2
МОДУЛЬНОЕ И ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ
Теоретическая часть
Модульное тестирования ПО в среде Visual Studio

Обозреватель тестов новой Visual Studio (VS) предназначен для поддержки разработчиков и коллективов, применяющих модульное тестирование на практике при разработке программного обеспечения. Модульное тестирование помогает обеспечить корректность программы, проверяя, что код приложения делает именно то, что вы от него ожидаете. В модульном тестировании, вы анализируете функциональность программы для обнаружения дискретных тестовых сценариев поведения, которые можно тестировать как отдельные единицы. Платформа модульного тестирования используется для создания тестов для этих сценариев поведения и отчета по результатам этих тестов.

Модульное тестирование имеет наибольший эффект, когда оно является неотъемлемой частью рабочего процесса разработки программного обеспечения. Как только создаётся функция или другой блок кода приложения, необходимо создать модульные тесты, проверяющие поведение кода в ответ на стандартные, граничные и неверные случаи входных данных, а также проверяющие все явные или неявные предположения, сделанные кодом. В практике разработки программного обеспечения, известной как разработка через тестирование, модульные тесты создаются перед разработкой кода, поэтому модульные тесты используются как проектная документация и функциональная спецификация функциональности.

Обозреватель тестов обеспечивает гибкий и эффективный способ выполнения модульных тестов и просмотра их результатов в VS. VS устанавливает платформы модульного тестирования Майкрософт для управляемого и неуправляемого кода. Обозреватель тестов также может запускать сторонние и открытые платформы модульного тестирования, реализующие интерфейсы дополнительных компонентов для Обозревателя тестов. Можно добавить многие из этих платформ с помощью диспетчера расширений VS и галереи VS. См. раздел Практическое руководство. Установка платформ модульного тестирования сторонних поставщиков.

Представления обозревателя тестов могут отображать все тесты или только тесты, которые прошли удачно, не удались, еще не были запущены, или были пропущены. Можно отфильтровать тесты в любом представлении, введя текст в поле поиска на глобальном уровне, либо выбрав один из предопределенных фильтров. Можно выполнить любую выборку тестов в любое время. При использовании VS Ultimate, тесты можно выполнять автоматически после каждого построения. Результаты тестового запуска немедленно отображаются на панели "пройден/неудача" в верхней части окна обозревателя. Подробные сведения о результатах выполнения метода тестирования отображаются при выборе теста.

Создание проекта модульного теста

Проект модульных тестов обычно отражает структуру отдельного проекта кода. В примере MyBank добавляется два проекта модульных тестов с именами AccountsTests и BankDbTests в решение MyBanks. Имя проекта тестов произвольно, но рекомендуется принять стандартное соглашение об именовании. Добавление нового проекта модульных тестов в решение: в меню Файл выберите Создать а затем выберите Проект (клавиатура: Ctrl + Shift + N); в диалоговом окне нового проекта разверните узел Установлено, выберите язык, который необходимо использовать для конкретного тестового проекта, а

затем выберите Тест; для использования одной из платформ модульного тестирования Microsoft выберите Проект модульного теста из списка шаблонов проектов. В противном случае выберите шаблон проекта платформы модульного тестирования, которую хотите использовать.

Написание тестов

Платформа модульного тестирования, которую вы используете, и VS IntelliSense помогут вам создать модульные тесты для проекта кода. Для запуска в Обозревателе тестов, большинство платформ требуют добавления определенных атрибутов, чтобы определить методы модульного теста. Платформы также предоставляют способ — обычно через заявления Assert или атрибуты метода — для определения того, прошел ли метод теста или нет. Другие атрибуты определяют необязательные методы настройки, которые выполняются при инициализации класса и перед каждым методом теста, а так же методы уничтожения, которые выполняются после каждого метода теста и перед уничтожением класса.

Шаблон AAA (Arrange, Act, Assert) является распространенным способом написания модульных тестов для тестируемого метода: 1. Раздел Arrange метода модульного теста инициализирует объекты и задает значение данных, которые передаются тестируемому методу. 2. Раздел Act вызывает тестируемый метод с подготовленными параметрами. 3. Раздел Assert проверяет, что действие тестируемого метода работает как ожидалось.

Чтобы проверить метод CheckingAccount.Withdraw нашего примера, можно написать 2 теста: один, который проверяет стандартное поведение метода и один, который проверяет, что снятие суммы превышающей баланс завершается ошибкой. Добавьте следующие методы в класс CheckingAccountTests:

```
C#

[public void Withdraw_ValidAmount_ChangesBalance()
```

```
{
   // arrange
   double currentBalance = 10.0;
   double withdrawal = 1.0;
   double expected = 9.0;
   var account = new CheckingAccount("JohnDoe", currentBalance);
   // act
   account.Withdraw(withdrawal);
   double actual = account.Balance;
   // assert
   Assert.AreEqual(expected, actual);
}
[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void Withdraw_AmountMoreThanBalance_Throws()
{
   // arrange
   var account = new CheckingAccount("John Doe", 10.0);
   // act
   account.Withdraw(1.0);
   // assert is handled by the ExcpectedException
}
```

Команда Withdraw_ValidAmount_ChangesBalance использует явное заявление Assert, чтобы указать, прошел ли метод теста или нет, в то время как Withdraw_AmountMoreThanBalance_Throws использует атрибут ExpectedException, чтобы указать успешность метода теста. Платформа модульного тестирования оборачивает методы теста в блоки try/catch. В большинстве случаев, если исключения перехватываются, то тестовый метод считается проваленным и исключения игнорируются.Атрибут ExpectedException приводит к тому, что метод теста считается пройденным, если выбрасывается указанное исключение.

Запуск тестов в Обозревателе тестов

При создании тестового проекта, тесты отображаются в Обозревателе тестов. Если Обозреватель тестов не отображается, выберите пункт Тест в меню VS, выберите Окна, а затем выберите Обозреватель тестов (рис. 2.1).

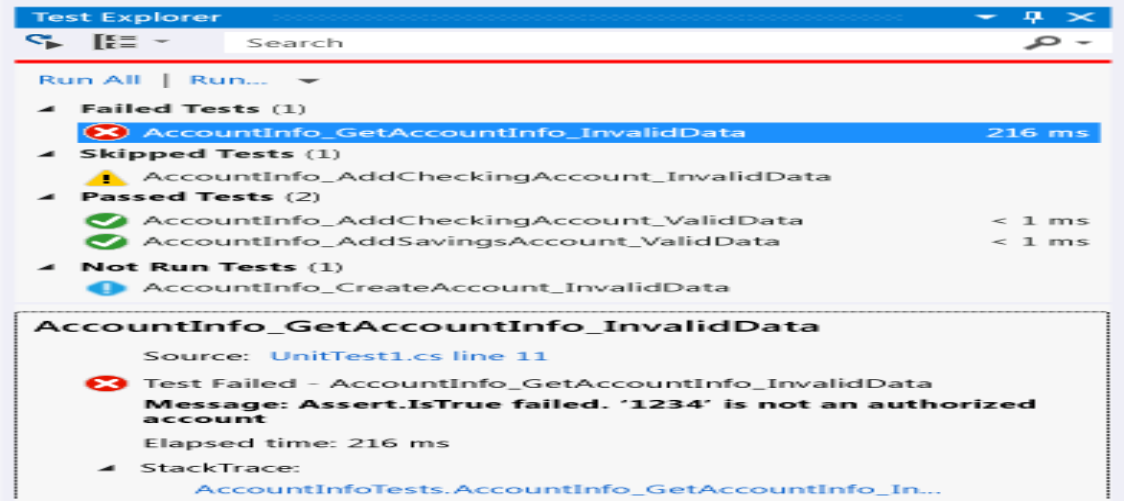

Рис. 2.1. Обозреватель тестов

При запуске, записи и повторном запуске тестов, представление по умолчанию в Обозревателе тестов отображает результаты в группах: Неудачные тесты, Пройденные тесты, Пропущенные тесты и Незапускавшиеся тесты. Можно выбрать заголовок группы, чтобы открыть представление, которое показывает все тесты в этой группе. Панель инструментов Обозревателя тестов помогает обнаружить, организовать и выполнить тесты, которые вас интересуют (рис. 2.2).

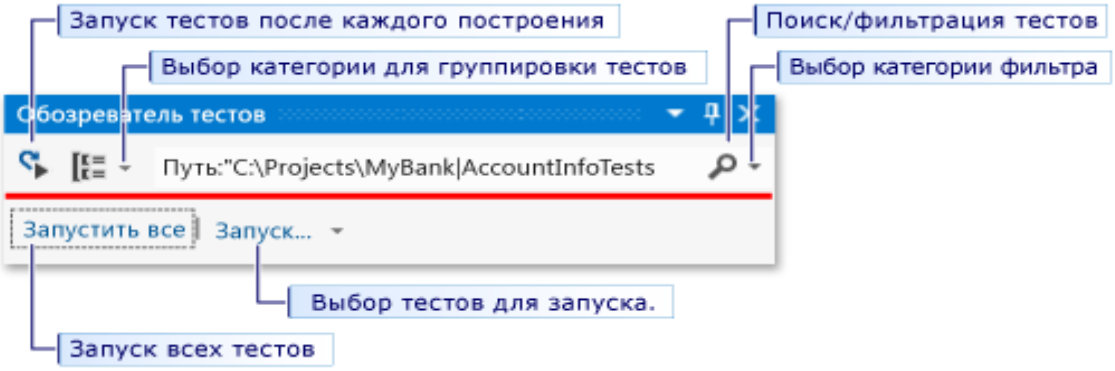

Рис. 2.2. Панель инструментов обозревателя тестов

Можно нажать Выполнить все, чтобы выполнить все тесты или нажать Выполнить для выбора подмножества тестов для выполнения. После выполнения набора тестов, отчет о тестовом запуске отображается в нижней части окна Обозревателя тестов. Выберите тест, чтобы просмотреть подробные сведения по нему в нижней панели. Выберите в контекстном меню

Открыть тест (клавиша F12) для отображения исходного кода для выбранного теста.

Внимание. Выполнение модульных тестов после каждого построения поддерживаются только в VS Ultimate.

Для выполнения модульных тестов после каждого локального построения в стандартном меню выберите Тест, на панели инструментов Обозревателя тестов выберите команду Выполнить тесты после построения.

Фильтрация и группирование списка тестов. При наличии большого количества тестов для фильтрации списка в поле поиска Обозревателя тестов можно ввести строку поиска. Можно ограничить фильтр еще больше, выбрав фильтр из списка (рис. 2.3).

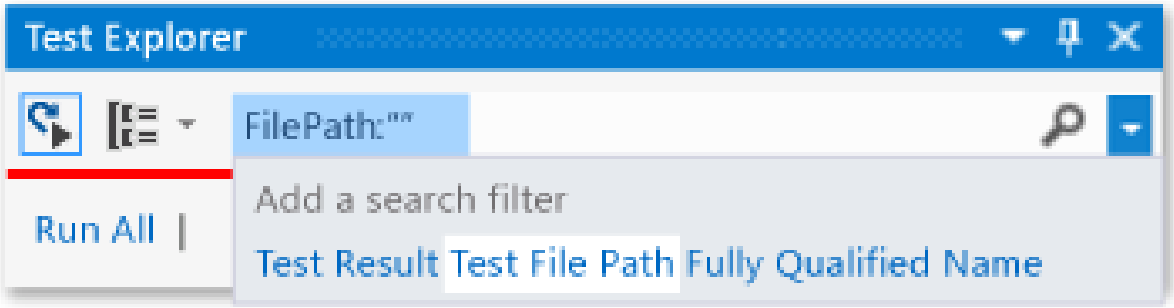

Рис. 2.3. Выбор фильтра из списка

Для группирования тестов по категориям, нажмите кнопку Группировка.

Типы тестов в VS 2010

Инструмент разработки программного обеспечения Microsoft VS 2010 предоставляет множество способов тестирования приложений с целью проверки их функциональности и производительности. В VS 2010 используются следующие основные виды тестов: 1) Basic Unit Test – модульный программный тест, при выполнении которого вызываются методы класса и проверяются возвращаемые ими значения; 2) Unit Test – то же самое, что и Basic Unit Test.

Разница между Basic Unit Test и Unit Test заключается в том, что Unit Test предоставляет разработчику более сложный конструктор для создания модульного теста; 3) Unit Test Wizard – мастер создания модульного теста. Позволяет быстро сгенерировать тест с помощью мастера, затем полученный код можно редактировать; 4) Coded Ui Test – данный вид тестов предназначен для автоматизации функциональных тестов и тестирования пользовательского интерфейса; 5) Generic Test – специальный вид тестов, который позволяет выполнять внешние тестирующие приложения; 6) Web Performance Test – функциональное тестирование веб-приложений в рамках организации нагрузочных тестов (Load Test); 7) Load Test – тесты для проведения нагрузочного тестирования веб-приложений; 8) Ordered Test – позволяют организовать выполнение всех написанных автоматических тестов в определенном порядке; 9) Database Unit Test – модульные тесты для тестирования баз данных.

Как видно, для автоматизации тестирования приложений на VS есть фактически весь необходимый набор тестов. В случае, если вам не будет хватать возможностей VS для создания автоматизированных тестов, вы можете выполнять внешние тестирующие приложения с помощью Generic Test. Для создания нового теста необходимо в главном меню выбрать “Test – New Test…”, после этого появится форма мастера создания нового теста. На данной форме отображается перечень доступных типов тестов для проекта. Сперва необходимо выбрать требуемый вид теста из списка, затем новый тест нужно связать с существующим проектом или создать для него новый проект тестирования кода, написанного на языке C++, C# или Microsoft Visual Basic (рис. 2.4). Для каждого вида тестов используется свой конструктор, а в некоторых случаях – мастер, который проводит вас через процедуру создания теста.

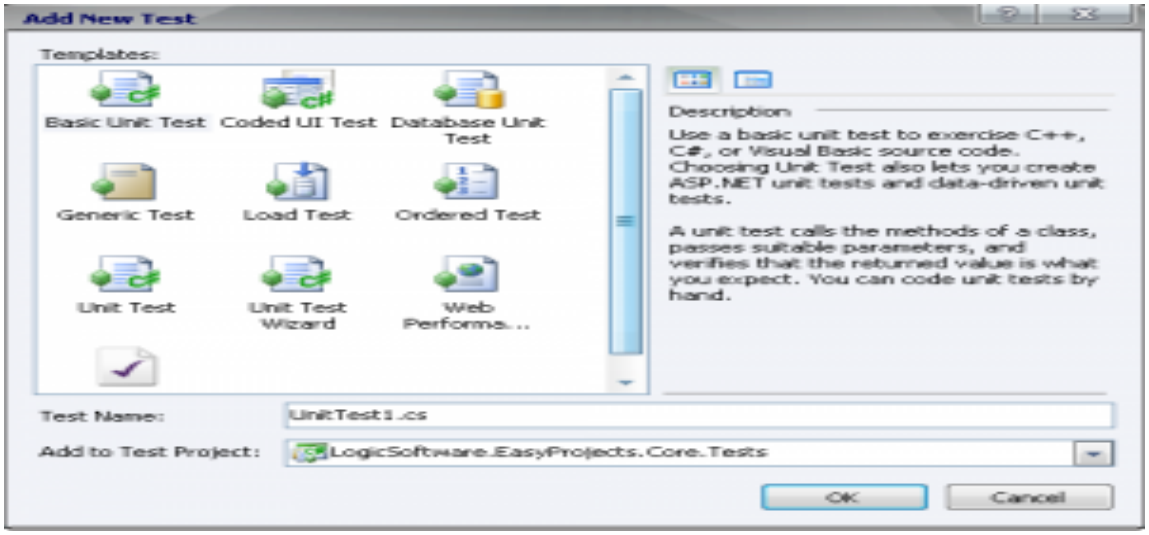

Рис. 2.4. Мастер создания нового теста для проекта типа C#

Программы VS Ultimate и VS Test Professional позволяют повысить продуктивность жизненного цикла тестирования, включая этапы планирования, тестирования и отслеживания хода выполнения работ. Эти средства интегрированы с сервером Team Foundation Server, который позволяет определять работы по тестированию на основе командных проектов, используемых другими подразделениями организации [5].

В состав Microsoft VS 2010 Ultimate и VS Test Professional 2010 теперь входит новое приложение Microsoft Test Manager, помогающее определять работы по тестированию и управлять ими посредством планов тестирования. Пользователь создает план тестирования и включает в него все необходимые наборы тестов, тестовые случаи и конфигурации, как показано на следующем рисунке.

После определения всех этих элементов можно приступать к тестированию. Если требования, описания функциональности пользователей или функции готовы к тестированию, можно выполнить тесты для каждой заданной конфигурации. Созданный план позволяет измерять ход тестирования при выполнении тестов и формировать отчеты по оставшейся работе. В Microsoft Test Manager можно выполнять ручные тесты, используя Microsoft Test Runner. Если с тестовым случаем

связана автоматизация, из Microsoft Test Manager можно также выполнять автоматические тесты. Результаты выполнения этих тестов будут связаны с планом тестирования.

Кроме того, автоматические тесты можно выполнять из среды VS, не связанной с планом. Выбор тестов для запуска осуществляется индивидуально, в составе политики возврата или на основе категорий тестов. Их также можно выполнять как часть построения, созданного с помощью Team Foundation Build, и из командной строки.

Поскольку средства тестирования интегрированы с другими компонентами VS Ultimate, можно сохранять результаты тестов в базе данных, создавать отчеты о тенденциях и исторические отчеты, а также сравнивать различные типы данных. Например, можно использовать данные для определения числа ошибок, обнаруженных в тесте, и более подробного анализа этих ошибок.

Нагрузочное тестирование с VS 2010

Для многих тестировщиков наличие такой возможности становится откровением. Видимо, это связано с тем, что такая возможность предоставляется только с Ultimate редакцией. В первую очередь выбираем Test — New Test. Перед нами открывается окно, в котором выбираем тип LoadTest, и видим, что наш тест может храниться в виде C#, C++ или VB проекта (рис. 2.5).

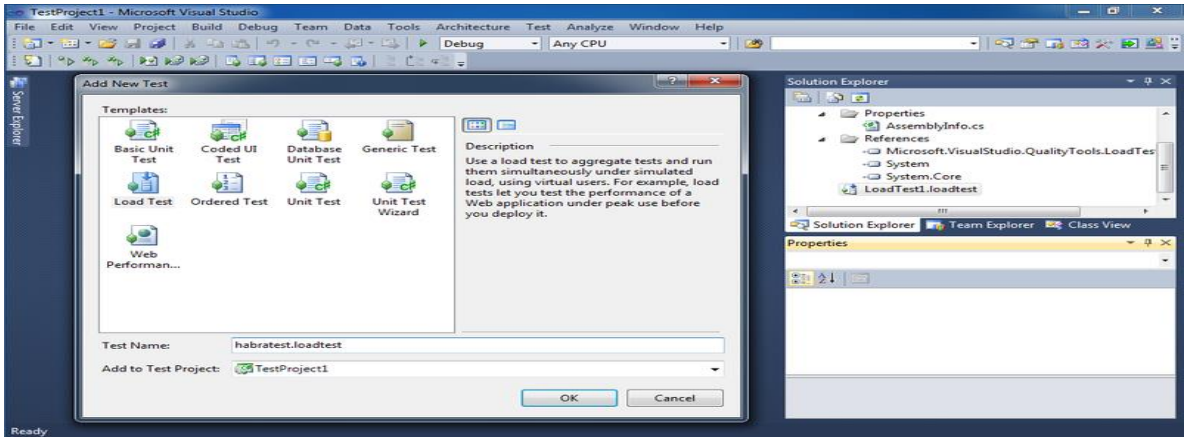

Рис. 2.5. Окно выбора теста

Выбираем любой тип и продолжаем. Дальше нас поведет Wizard, во время прохождения шагов которого нам будет предложено установить Think Time между выполнением запросов, выбрать Load Pattern. Следует отметить, что в Wizard'e представлены не все модели нагрузки (рис. 2.6). Самым необычным является шаблон нагрузки «Goal», при котором автоматически увеличивается или уменьшается число активных VUser'ов в зависимости от загрузки процессора целевой машины (сервера, на котором крутится тестируемое веб-приложение).

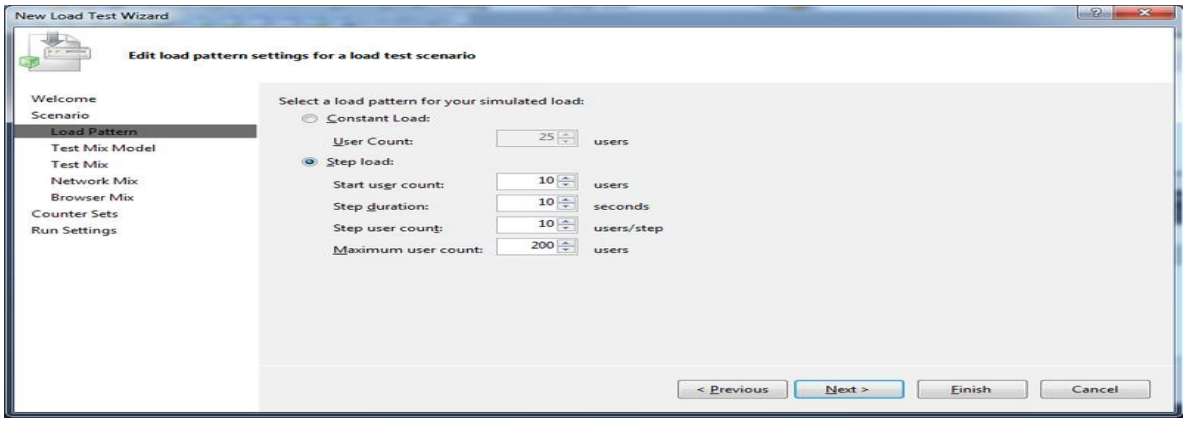

Рис. 2.6. Окно настройки теста

Затем Wizard предлагает выбрать Test Mix Model, то есть то, в каком порядке VUser'ы будут выполнять тестовые сценарии, которых может быть более одного (рис. 2.7).

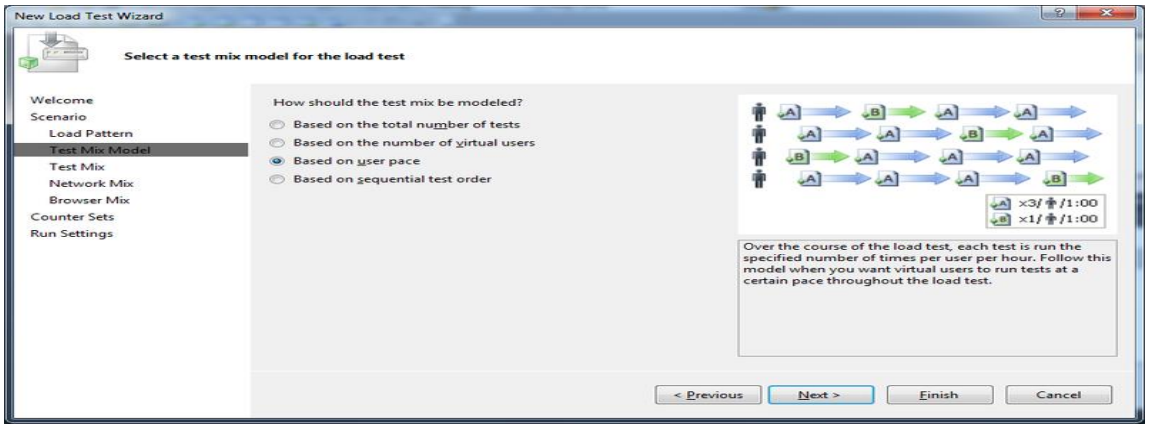

Рис. 2.7. Окно выбора модели теста

На следующем шаге предлагается выбрать те сценарии, которые будут выполняться виртуальными пользователями. После этого можно выбрать подключения через те сети,

которые будут эмулироваться (их пропускную способность) (рис. 2.8).

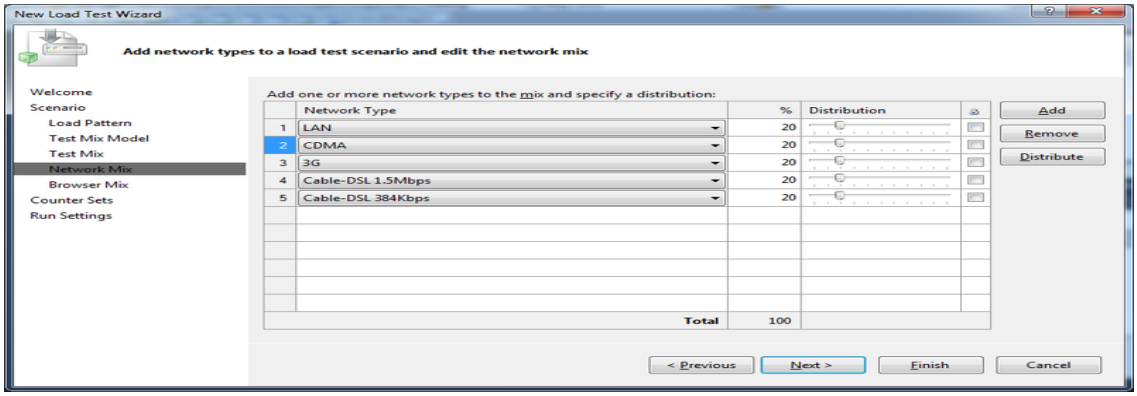

Рис. 2.8. Выбор эмулируемых сетей

Следующим шагом является выбор того, в каком процентном соотношении распределятся User Agent'ы отправляемых запросов. Затем мы выбираем компьютеры, состояние которых будет мониториться. После установки всех параметров, мы попадаем в созданный нами проект (рис. 2.9).

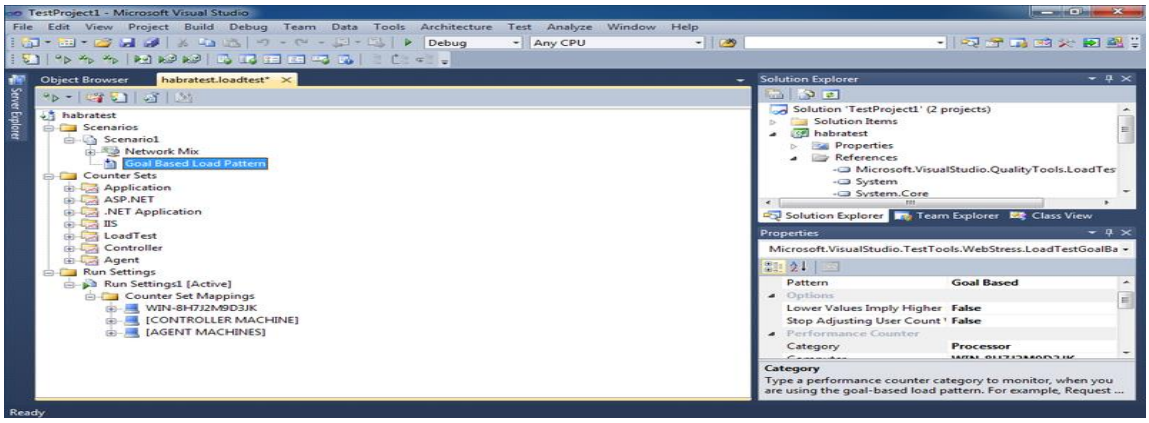

Рис. 2.9. Окно созданного проекта

Следующим шагом является выбор того, в каком процентном соотношении распределятся User Agent'ы отправляемых запросов. Затем выбираются компьютеры, состояние которых будет мониториться. После установки всех параметров, произойдет попадание в созданный проект. Можно рассмотреть создание и запись конкретного сценария нагрузочного тестирования с помощью Test Manager, сбор и анализ метрик с нагружаемой машины, а можно и разобрать сценарий с msdn.

Юнит тестирование программы на C#

Возможности VS. VS 2010 с пакетом обновления 1 (SP1) поддерживает следующие базовые возможности модульного тестирования, ориентированные на .NET Framework 3.5: 1. можно создавать проекты модульного тестирования и указывать для них в качестве требуемой версии .NET Framework 3.5; 2. можно выполнять модульные тесты, ориентированные на .NET Framework 3.5, из VS 2010 с пакетом обновления 1 (SP1) на локальном компьютере; 3. можно выполнять модульные тесты, ориентированные на .NET Framework 3.5, из командной строки с помощью программы MSTest.exe.

Изменение для проектов модульного тестирования Visual C#: Создайте новый проект модульного тестирования C#: в меню **Файл** выберите команду **Создать** и щелкните **Проект**. Откроется диалоговое окно **Новый проект**; в области **Установленные шаблоны** разверните узел **Visual C#**. Выберите **Тест**, а затем выберите шаблон **Тестовый проект**; В текстовом поле **Имя** введите имя тестового проекта Visual C# и нажмите кнопку **ОК**; в обозревателе решений щелкните созданный тестовый проект Visual C# правой кнопкой мыши и выберите пункт **Свойства**. Будут показаны свойства тестового проекта Visual C#; на вкладке **Приложение** с помощью раскрывающегося списка **Целевая рабочая среда** измените требуемую версию с **.NET Framework 4** на**.NET Framework 3.5**, как показано на рис. 2.10.

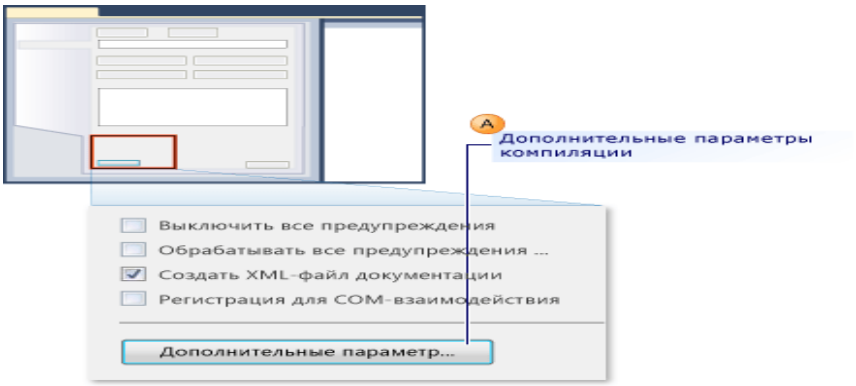

Рис. 2.10. Замена версии .NET

С помощью раскрывающегося списка **Заданная исполняющая среда (все конфигурации)** измените требуемую версию с

.NET Framework 4 на **.NET Framework 3.5**, как показано в выноске B на рис. 2.11.

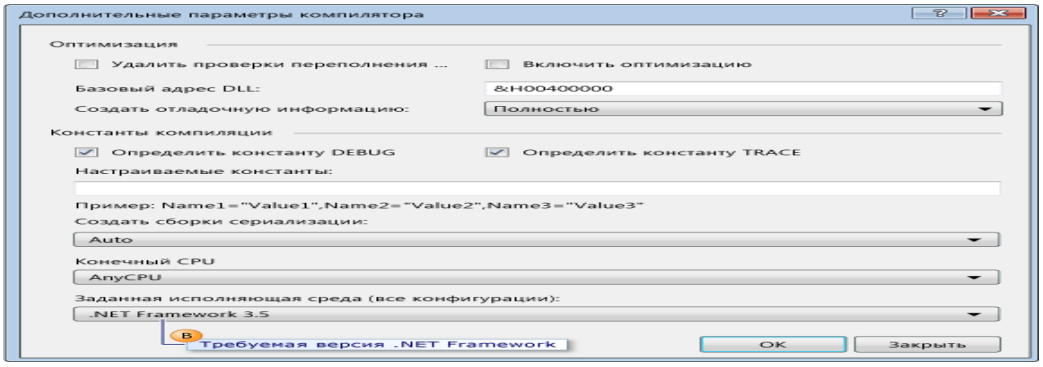

Рис. 2.11. Выбор версии .NET

Изменение требуемой версии на .NET Framework 3.5 для проектов модульного тестирования Visual C#: создайте новый проект модульного тестирования C#. В меню **Файл** выберите команду **Создать** и щелкните **Проект**; откроется диалоговое окно **Новый проект**; в области **Установленные шаблоны** разверните узел **Visual C#**. Выберите **Тест**, а затем выберите шаблон **Тестовый проект**; в текстовом поле **Имя** введите имя тестового проекта Visual C# и нажмите кнопку **ОК**; в обозревателе решений щелкните созданный тестовый проект Visual C# правой кнопкой мыши и выберите пункт **Свойства**. Будут показаны свойства тестового проекта Visual C#.

На вкладке **Приложение** с помощью раскрывающегося списка **Целевая рабочая среда** измените требуемую версию с **.NET Framework 4** на **.NET Framework 3.5**, как показано на рис. 2.12. Возможные дополнительные шаги по изменению требуемой версии тестовых проектов на .NET Framework 3.5 представлены в http://msdn.microsoft.com/ru-ru/library/vstudio/gg601487(v=vs.100).aspx

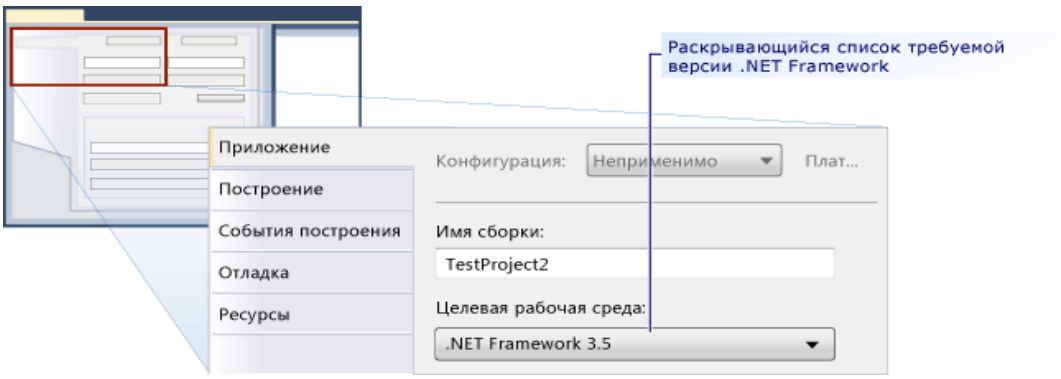

Рис. 2.12. Список требуемой версии .NET

Тестирование в VS 2012

VS 2012 обеспечивает совместимость тестовых проектов с тестовыми проектами VS 2010 SP1. Например, тестовые проекты, которые были созданы в VS 2010 SP1 можно открыть с помощью VS 2012 без какого-либо обновления. Следовательно, ваш коллектив может использовать и VS 2010 SP1, и VS 2012 для работы с одним и тем же тестовым проектом.

В VS 2012 представлено несколько изменений в модульном тестировании. Вследствие этих изменений, важно понимать проблемы совместимости между предыдущими версиями VS и VS 2012. Из всех изменений, относящихся к модульному тестированию, значительным изменением является то, что VS 2012 содержит в себе больше, чем один шаблон тестового проекта, включая, в том числе и шаблон проекта модульного теста. Новые модульные тесты добавляются к новому шаблону проекта модульного теста. Модульные тесты можно также включать в другом новом шаблоне, называющимся "Проект с закодированными тестами пользовательского интерфейса".

Интеграционное тестирование

Основное назначение тестирования взаимодействий состоит в том, чтобы убедиться, что происходит правильный *обмен сообщениями* между объектами, классы которых уже прошли тестирование в автономном режиме (на модульном уровне

тестирования). Тестирование взаимодействия или *интеграционное тестирование* представляет собой тестирование собранных вместе, взаимодействующих модулей (объектов). В интеграционном тестировании можно объединять разное количество объектов - от двух до всех объектов тестируемой системы. *Интеграционное тестирование* отличается от системного тем, что: при *интеграционном тестировании* используется подход "белого ящика", а при системном - "черного ящика"; целью интеграционного тестирования является только проверка правильности взаимодействия объектов, тогда как целью системного - проверка правильности функционирования системы в целом.

Идентификация взаимодействий

Взаимодействие объектов представляет собой просто *запрос* одного объекта (**отправителя**) на выполнение другим объектом (**получателем**) одной из операций получателя и всех видов обработки, необходимых для завершения этого запроса.

В ситуациях, когда в качестве основы тестирования взаимодействий объектов выбраны только спецификации общедоступных операций, тестирование намного проще, чем когда такой основой служит реализация. Мы ограничимся тестированием общедоступного интерфейса. Такой подход вполне оправдан, поскольку мы полагаем, что классы уже успешно прошли *модульное тестирование*. Тем не менее, выбор такого подхода отнюдь не означает, что не нужно возвращаться к спецификациям классов, дабы убедиться в том, что тот или иной метод выполнил все необходимые вычисления. Это обусловливает необходимость проверки значений атрибутов внутреннего состояния получателя, в том числе любых агрегированных атрибутов, т.е. атрибутов, которые сами являются объектами. Основное внимание уделяется отбору тестов на основе спецификации каждой *операции* из общедоступного интерфейса класса.

Взаимодействия неявно предполагаются в спецификации класса, в которой установлены ссылки на другие объекты. В

разделе 4 рассматривалось тестирование примитивных классов. Такие объекты представляют собой простейшие компоненты системы и, несомненно, играют важную роль при выполнении любой программы. Тем не менее, в объектно-ориентированной программе существует сравнительно небольшое количество примитивных классов, которые реалистично моделируют объекты задачи и все отношения между этими объектами. Обычным явлением для хорошо спроектированных объектно- ориентированных программ является использование непримитивных классов; в этих программах им отводится главенствующая роль.

Выявить такие взаимодействующие классы можно, используя отношения ассоциации (в том числе отношения агрегирования и композиции), представленные на диаграмме классов. Ассоциации такого рода преобразуются в интерфейсы класса, а тот или иной *класс* взаимодействует с другими классами посредством одного или нескольких способов:

Тип 1. Общедоступная операция имеет один или большее число формальных параметров объектного типа. Сообщение устанавливает ассоциацию между получателем и параметром, которая позволяет получателю взаимодействовать с этим параметрическим объектом.

Тип 2. Общедоступная операция возвращает значения объектного типа. На *класс* может быть возложена задача создания возвращаемого объекта, либо он может возвращать модифицированный *параметр*.

Тип 3. Метод одного класса создает экземпляр другого класса как часть своей реализации.

Тип 4. Метод одного класса ссылается на глобальный экземпляр некоторого другого класса. Разумеется, принципы хорошего тона в проектировании рекомендуют минимальное использование глобальных объектов. Если реализация какого-

либо класса ссылается на некоторый глобальный *объект*, рассматривайте его как неявный *параметр* в методах, которые на него ссылаются.

Выбор тестовых случаев

Исчерпывающее тестирование, другими словами, прогон каждого возможного тестового случая, покрывающего каждое сочетание значений - это, вне всяких сомнений, надежный подход к тестированию. Однако во многих ситуациях количество тестовых случаев достигает таких больших значений, что обычными методами с ними справиться попросту невозможно. Если имеется принципиальная возможность построения такого большого количества тестовых случаев, на построение и выполнение которых не хватит никакого времени, должен быть разработан *систематический* метод определения, какими из тестовых случаев следует воспользоваться. Если есть выбор, то мы отдаем предпочтение таким тестовым случаям, которые позволяют найти ошибки, в обнаружении которых мы заинтересованы больше всего. Существуют различные способы определения, какое *подмножество* из *множества* всех возможных тестовых случаев следует выбирать. При любом подходе мы заинтересованы в том, чтобы систематически повышать уровень покрытия.

Задания

1. Определение модульного тестирования.
2. Перечислить преимущества модульного тестирования.
3. Описать процесс создания модульного теста.
4. Пример написания теста.
5. Описать запуск теста в Обозревателе тестов.
6. Виды тестов в VS 2010.
7. Виды тестов в VS 2012.
8. Нагрузочное тестирование в VS.
9. Юнит-тестирование на языке C#.
10. Особенности тестирования в последней версии VS.

11. Определение интеграционного тестирования.
12. Цели интеграционного тестирования.
13. Взаимодействия и их идентификация.
14. Типы взаимодействий классов.
15. Выбор тестовых случаев.

Практическое занятие № 3
СИСТЕМНОЕ ТЕСТИРОВАНИЕ
Теория

Системное тестирование качественно отличается от интеграционного и модульного уровней. *Системное тестирование* рассматривает систему в целом и применяется на уровне пользовательских интерфейсов, в отличие от последних фаз *интеграционного тестирования*, которое оперирует на уровне интерфейсов модулей, хотя набор модулей может быть аналогичным. Различны и цели этих уровней тестирования. На уровне системы часто сложно и малоэффективно анализировать прохождение тестовых траекторий внутри программы, а также отслеживать правильность работы конкретных функций. Основной *задачей системного тестирования* является выявление дефектов, связанных с работой системы в целом, таких как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство в использовании и тому подобное.

Поскольку *системное тестирование* проводится на уровне пользовательских интерфейсов, то построение специальной тестовой системы становится технически необязательным. Однако объемы данных на этом уровне таковы, что обычно более эффективным подходом является полная или частичная *автоматизация тестирования*, что может потребовать создания тестовой системы намного более сложной, чем система тестирования на уровне модулей или их комбинаций.

Необходимо подчеркнуть, что существует два принципиально разных подхода к системному тестированию.

В первом варианте для построения тестов используются требования к системе, например, для каждого требования строится тест, который проверяет выполнение данного требования в системе. Этот подход особенно широко применяется при разработке военных и научных систем, когда заказчик вполне осознает, какая функциональность ему нужна, и составляет полный набор **формальных требований**. Тестировщик в данном случае только проверяет, соответствует ли разработанная система этому набору. Такой подход предполагает длинную и дорогостоящую фазу сбора требований, выполняемую до начала собственно проекта. В этом случае для определения требований обычно разрабатывается прототип будущей системы.

Во втором подходе основой для построения тестов служит *представление* о способах использования продукта и о задачах, которые он решает. На основе более или менее формальной модели пользователя создаются **случаи использования** системы, по которым затем строятся собственно **тестовые случаи**. **Случай использования (use case)** описывает, как субъект использует систему, чтобы выполнить ту или иную задачу. **Субъекты** или **актеры (actors)**могут исполнять различные роли при работе с системой. **Случаи использования** могут описываться с различной степенью абстракции. **Случаи использования** не обязательно охватывают каждое требование. Можно конкретизировать **случаи использования** и расширять их в наборы более **специфических случаев использования (пошаговое описание случая использования)**. В контексте конкретного **случая использования** можно определить один или большее число **сценариев**. **Сценарий** представляет конкретный экземпляр **случая использования - путь в пошаговом описании случая использования**. Каждый **путь (сценарий)** в **случае использования** должен быть протестирован (рис. 3.1).

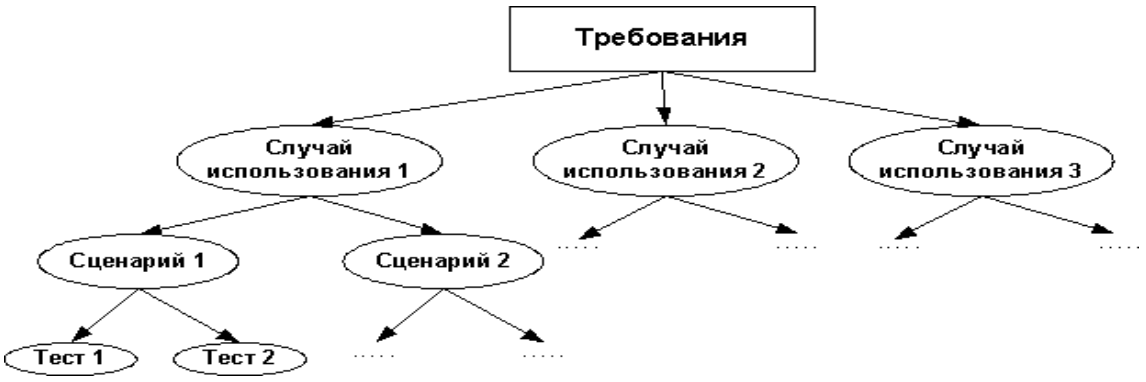

Рис. 3.1. Тестирование случаев использования

Входные данные для каждого **сценария** надо выбирать следующим образом: идентифицировать все значения (входные данные), которые могут задавать субъекты для **случая использования**; определить **классы эквивалентности** для каждого типа входных данных; построить таблицу со списком значений из различных классов эквивалентности; построить **тестовые случаи** на базе таблицы с учетом внешних ограничений.

Далее при построении **тестовых случаев** применялись оба подхода и при выполнении заданий необходимо действовать следующим образом: на основе требований определить случаи использования (use case); на основе каждого случая использования (use case) построить сценарии; для каждого сценария разработать тестовые случаи (набор тестов).

Описание случая использования "подбор подшипников"

Последовательно приходят два подшипника, поступает запрос от оси. При поступлении запроса от оси система подбирает два подшипника из имеющихся на складе и выдает их в выходную ячейку.

Рассмотрим этот **случай использования** подробнее. Согласно спецификации, система постоянно опрашивает склад и терминал оси. При поступлении подшипника (статус склада 32) система опрашивает терминал подшипника, формирует и посылает команду складу "принять подшипник" и получает ответ от склада о результатах выполнения команды. При поступлении

оси (поступлении параметров оси при опросе терминала оси) система должна подобрать подшипники из имеющихся на складе, сформировать команды для их выдачи, послать их складу и получить ответ о результате выполнения команд.

Далее приводится пошаговое описание этого случая использования: приняли на склад первый подшипник (1-10); приняли на склад второй подшипник (11-20); поступила ось (21-26); подбираем первый подшипник для оси (27-30); подбираем второй подшипник для оси (31-34); завершение выдачи команд (35-39).

Пошаговое описание случая использования (рис. 3.2)

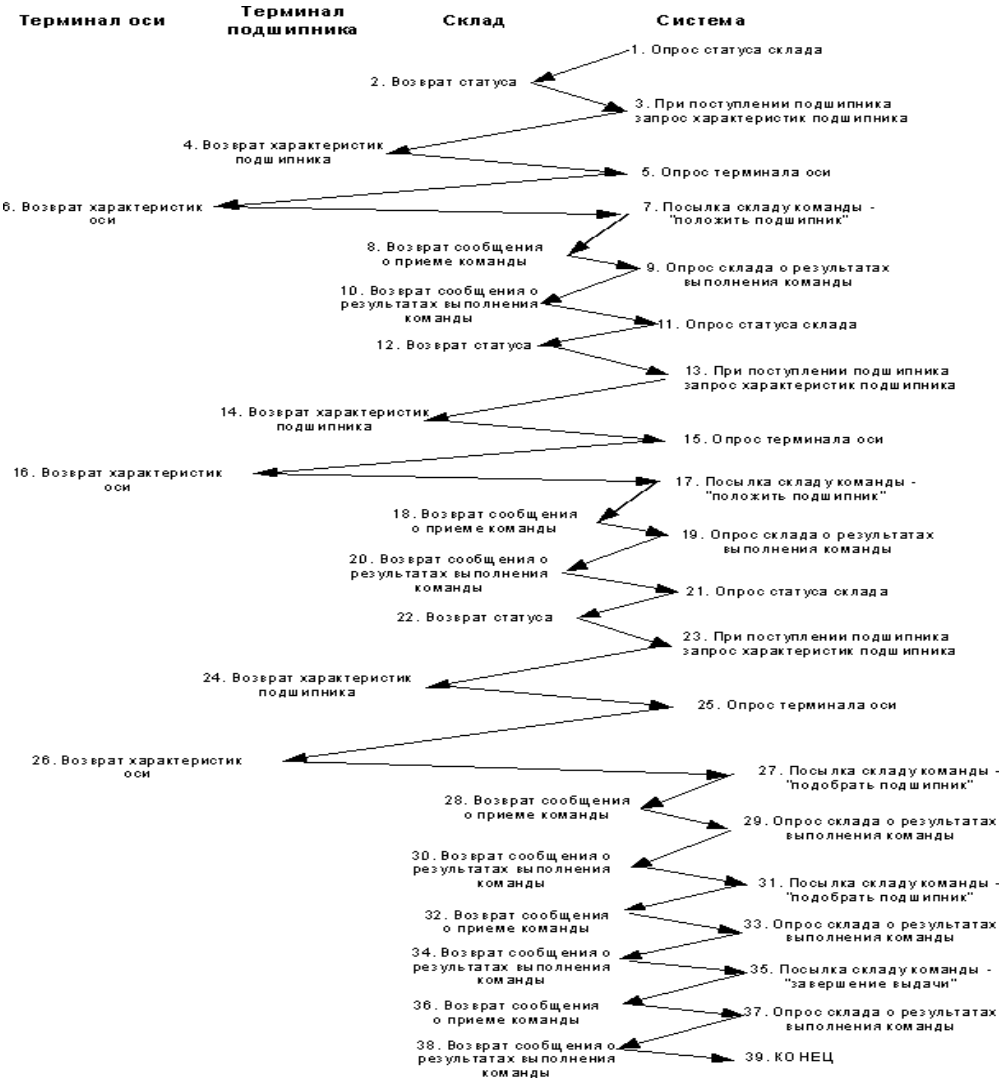

Рис. 3.2. Пример use case

Описание процесса системного тестирования

Анализ. Тестируемая система анализируется (проверяется) на наличие определенных свойств, которым надо уделить особое внимание, и определяются соответствующие **тестовые случаи**.

Построение. Выбранные на стадии анализа **тестовые случаи** переводятся на *язык программирования*.

Выполнение и анализ результатов. Производится выполнение **тестовых случаев**. Полученные результаты анализируются, чтобы определить, успешно ли прошла система испытания на **тестовом наборе**.

Процесс запуска тестовых случаев и анализа полученных результатов должен быть подробно описан в **тестовых процедурах**.

Задания

1. Определение системного тестирования.
2. Различия системного, модульного и интеграционного тестирования.
3. Подходы к системному тестированию.
4. Тестирование случаев использования.
5. Выбор входных данных для каждого случая.
6. Практический пример случая использования.
7. Пошаговое выполнение случая использования.
8. Процесс системного тестирования.

Практическое занятие № 4
РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ
Теоретическая часть

Регрессионное тестирование - цикл тестирования, который производится при внесении изменений на фазе *системного*

тестирования или сопровождения продукта. Главная проблема *регрессионного тестирования* - выбор между полным и частичным *перетестированием* и пополнением тестовых наборов. При частичном *перетестировании* контролируются только те части проекта, которые связаны с измененными компонентами. На ГМП это пути, содержащие измененные узлы, и, как правило, это методы и классы, лежащие выше модифицированных по уровню, но содержащие их в своем контексте. Пропуск огромного объема тестов, характерного для этапа *системного тестирования*, удается осуществить без потери качественных показателей продукта только с помощью регрессионного подхода.

Пример регрессионного тестирования

Получив отчет об ошибке, программист анализирует исходный код, находит ошибку, исправляет ее и модульно или интеграционно тестирует результат. В свою очередь тестировщик, проверяя внесенные программистом изменения, должен: проверить и утвердить исправление ошибки, выполнив указанный в отчете тест, с помощью которого была найдена ошибка; попробовать воспроизвести ошибку каким-нибудь другим способом; протестировать последствия исправлений. Возможно, что внесенные исправления привнесли ошибку (наведенную ошибку) в код, который до этого исправно работал.

Например, при тестировании класса TCommandQueue запускаем тесты:

```
// Тест проверяет, создается ли объект типа TCommand и  добавляется ли он в конец очереди.
private void TCommandQueueTest1()
// Тест проверяет добавление команд в очередь на указанную позицию. Также проверяется правильность удаления команд из очереди.
private void TCommandQueueTest2()
7.4. Набор тестов класса TCommandQueue
// Тест проверяет, создается ли объект типа TCommand и добавляется ли он в конец очереди.
void TCommandQueueTest1()
// Тест проверяет добавление команд в очередь на указанную позицию. Также проверяется правильность удаления команд из очереди.
```

```
void TCommandQueueTest2()
```
4.1. Набор тестов класса TCommandQueue (C++)

При этом первый тест выполняется успешно, а второй нет, т.е. команда добавляется в конец очереди команд успешно, а на указанную позицию - нет. Разработчик анализирует код, который реализует тестируемую функциональность:

```
...

if ((Position<-1)&&
    (Position<=this.Items.Count))
{
  this.Items.Insert(Position, Command);
}
else
{
  if (Position==-1)
  {
    this.Items.Add(Command);
  }
}
```
7.5. Фрагмент кода с зафиксированным при тестировании дефектом
```
if ((Position <-1)&&(Position<=this.Items.Count))
{
  this.Items.Insert(Position, Command);
}
else
{
  if (Position==-1)
  {
    this.Items.Add(Command);
  }
}
```
4.2. Фрагмент кода с зафиксированным при тестировании дефектом

Анализ показывает, что ошибка заключается в использовании неверного знака сравнения в первой строке фрагмента. Далее программист исправляет ошибку, например: следующим образом:

```
...

if ((Position>=-1)&&
    (Position<=this.Items.Count))
{
```

```
  this.Items.Insert(Position, Command);
}
else
{
  if (Position==-1)
  {
    this.Items.Add(Command);
  }
}
...
```

7.6. Исправленный фрагмент кода

```
if ((Position>=-1)&&
    (Position<=this.Items.Count))
{
  this.Items.Insert(Position, Command);
}
else
{
  if (Position==-1)
  {
    this.Items.Add(Command);
  }
}
```

4.3. Исправленный фрагмент кода

Для проверки скорректированного кода хочется пропустить только тест TCommandQueueTest2. Можно убедиться, что тест TCommandQueueTest2 будет выполняться успешно. Однако одной этой проверки недостаточно. Если мы повторим пропуск двух тестов, то при запуске первого теста,TCommandQueueTest1, будет обнаружен новый дефект. Повторный анализ кода показывает, что ветка else не выполняется. Таким образом, исправление в одном месте привело к ошибке в другом, что демонстрирует необходимость проведения полного *перетестирования*. Однако повторное *перетестирование* требует значительных усилий и времени. Возникает задача – отобрать сокращенный набор тестов из исходного набора (может быть, пополнив его рядом дополнительных - вновь разработанных - тестов), которого, тем не менее, будет достаточно для исчерпывающей проверки функциональности в соответствии с выбранным критерием. Организация повторного тестирования в условиях сокращения ресурсов, необходимых для

обеспечения заданного уровня качества продукта, обеспечивается *регрессионным тестированием*.

Комбинирование уровней тестирования

В каждом конкретном проекте должны быть определены задачи, ресурсы и технологии для каждого уровня тестирования таким образом, чтобы каждый из типов дефектов, ожидаемых в системе, был "адресован", то есть в общем наборе тестов должны иметься тесты, направленные на выявление дефектов подобного типа. Табл. 4.1 суммирует характеристики свойств модульного, интеграционного и системного уровней тестирования. Задача, которая стоит перед тестировщиками и менеджерами, заключается в оптимальном распределении ресурсов между всеми тремя типами тестирования. Например, перенесение усилий на *поиск* фиксированного типа дефектов из области системного в область *модульного тестирования* может существенно снизить сложность и *стоимость* всего процесса тестирования.

Таблица 4.1. Характеристики модульного, интеграционного и *системного тестирования*

	Модульное	**Интеграционное**	**Системное**
Типы дефектов	Локальные дефекты, такие как опечатки в реализации алгоритма, неверные операции, логические и математические выражения, циклы, ошибки в использовании локальных ресурсов, рекурсия и т.п.	Интерфейсные дефекты, такие как неверная трактовка параметров и их формат, неверное использование системных ресурсов и средств коммуникации, и т.п.	Отсутствующая или некорректная функциональность, неудобство использования, непредусмотренные данные и их комбинации, непредусмотренные или неподдерживаемые сценарии работы, ошибки совместимости, ошибки пользовательской документации, ошибки переносимости продукта на различные платформы, проблемы производительности, инсталляции и т.п.
Необходимость в системе тестирования	Да	Да	Нет (*)

Цена разработки системы тестирования	Низкая	Низкая до умеренной	Умеренная до высокой или неприемлемой
Цена процесса тестирования, то есть разработки, прогона и анализа тестов	Низкая	Низкая	Высокая

(*) *прямой* необходимости в системе тестирования нет, но цена процесса *системного тестирования* часто настолько высока, что требует использования систем автоматизации.

Задания

1. Определение регрессионного тестирования.
2. Практический пример регрессионного тестирования.
3. Анализ кода примера.
4. Оценка функциональности.
5. Исправление ошибок примера кода.
6. Анализ комбинаций уровней тестирования.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Орлик С. Основы программной инженерии - http://swebok.sorlik.ru/4_software_testing.html
2. Бейзер Б. Тестирование черного ящика: Технологии функционального тестирования ПО. - СПб.: Питер, 2004. - 318 с.
3. Бек К. Экстремальное программирование: разработка через тестирование. - СПб.: Питер, 2004. — 224 с.
4. Канер С. Тестирование ПО. Фундаментальные концепции менеджмента бизнес-приложений. – М.: Изд. Лори, 2011. – 544 с.
5. Рекс Блэк. Ключевые процессы тестирования. Планирование, подготовка, проведение. - СПб.: Питер, 2011. — 544 с.
6. Казиев В. Введение в практическое тестирование. - http://www.intuit.ru/studies/courses/1023/300/info
7. Налютин Н. и др. Верификация программного обеспечения. - http://www.intuit.ru/studies/courses/1040/209/info
8. Котляров В. Основы тестирования программного обеспечения. - http://www.intuit.ru/studies/courses/48/48/info
9. Тестирование приложения http://msdn.microsoft.com/ru-u/library/ms182409(d=printer,v=vs.100).aspx
10. Основы современного тестирования ПО, разработанного на C#, Учебное пособие. В.П.Котляров и др.- Санкт-Петербург, 2004, 170с.
11. С. Канер и др. Тестирование ПО. — К.: Диасофт, 2000. — 544 с.

Учебно-методическое издание

Методические указания
для практических занятий

Потапенко Владислав Яковлевич

Тестирование и отладка программного обеспечения

Редактор *Я.В.Максименко*

Подписано в печать 02.11.2016.
Формат 60x84 1/16. Бумага офсетная. Печать цифровая.
Усл. печ. л. 2,32. Уч.-изд. л. 2,5. Тираж 50 экз. Заказ ____.

Южно-Российский государственный политехнический университет
(НПИ) им. М.И. Платова
Редакционное-издательский отдел ЮРГПУ (НПИ)
346428, Новочеркасск, ул. Просвещения, 132

Отпечатано в ИД «Политехник»
346428, г. Новочеркасск, ул. Первомайская, 166
idp-npi@mail.ru