Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)**

Кафедра автоматизации обработки информации (АОИ)

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Методические указания к лабораторным работам и
организации самостоятельной работы
для студентов направления
«Программная инженерия»
(уровень бакалавриата)

2018

Морозова Юлия Викторовна
Тестирование программного обеспечения: Методические указания к лабораторным работам и организации самостоятельной работы для студентов направления «Программная инженерия» (уровень бакалавриата) / Ю.В. Морозова. – Томск, 2018. – 46 с.

Оглавление

1 Введение 4
2 Методические указания к проведению лабораторных работ....5
2.1 Правила выполнения лабораторных работ 5
2.2 Лабораторная работа «MindMap»....6
2.3 Лабораторная работа «Тест-кейсы»....7
2.4 Лабораторная работа «Ручное тестирование»....12
2.5 Лабораторная работа «Локализация дефектов» 13
2.6 Лабораторная работа «Классификация тестов» 15
2.7 Лабораторная работа «Попарное тестирование» 20
2.8 Лабораторная работа «Модульное тестирование»....23
2.9 Лабораторная работа «Автоматизированное тестирование» 34
3 Методические указания для организации самостоятельной работы....39
3.1 Общие положения 39
3.2 Проработка лекционного материала и подготовка к контрольным работам....39
3.3 Подготовка к лабораторным работам....41
3.4 Самостоятельное изучение тем теоретической части курса 42
4 Рекомендуемые источники 45
Приложение А 46

1 Введение

Двадцать с лишнем лет назад в бизнесе программирования считалось нормальным, что разработчики самостоятельно тестировали свой код. Сегодня тестирование стало обязательной частью процесса производства программного обеспечения.

Целью проведения лабораторных работ и самостоятельной работы является получение практических навыков тестирования программного обеспечения.

В результате изучения дисциплины студент должен:

- **знать:** основные понятия и методы тестирования; условия применения тестирования; приемы тестирования на разных фазах разработки качественного программного продукта;
- **уметь:** разрабатывать тестовые сценария в проекте; разрабатывать тестовую документацию; выполнять тест-кейсы для разных видов тестирования;
- **владеть:** основными методиками тестирования программного обеспечения; одним либо несколькими прикладными программами для тестирования программного обеспечения.

Данные методические указания предназначены для организации самостоятельной работы и выполнения лабораторной работы по дисциплине «Тестирование программного обеспечения» подготовки бакалавров направления «Программная инженерия».

2 Методические указания к проведению лабораторных работ

2.1 Правила выполнения лабораторных работ

В ходе выполнения лабораторной работы студент должен строго выполнять весь объем самостоятельной подготовки, указанный в описаниях соответствующих лабораторных работ. Выполнению каждой работы предшествует проверка готовности студента, которая проводится преподавателем.

Лабораторные занятия выполняются студентами самостоятельно, преподаватель в ходе занятия осуществляет научное и методическое руководство действиями студентов.

После выполнения работы студент должен представить отчет о проделанной работе с обсуждением полученных результатов и выводов.

Защита отчета по лабораторной работе заключается в предъявлении преподавателю полученных результатов, демонстрации полученных навыков в ответах на вопросы преподавателя.

Темы лабораторных работ:

1. MindMap.
2. Тест-кейсы.
3. Ручное тестирование.
4. Локализация дефектов.
5. Классификация тестов.
6. Попарное тестирование.
7. Модульное тестирование.
8. Автоматизированное тестирование.

Отчет оформляемся согласно образовательному стандарту вуза. Титульный лист представлен в приложении А. Изложение должно быть последовательным, логичным, конкретным. В текст отчета могут быть включены небольшие фрагменты программного кода, обязательно с комментариями. Рекомендуемый шрифт для выполнения фрагмента кода – Courier New, размер 12пт. На материалы, взятые из литературы и других источников должны быть даны ссылки с указанием номера источника по списку использованной литературы. В приложениях размещаются листинг, схемы программы, скриншоты интерфейса. Приложения нумеруются русскими буквами в порядке появления ссылок на них в основном тесте документа.

2.2 Лабораторная работа «MindMap»

Цель работы: получение практических навыков по созданию интеллект-карты (MindMap) на проект и применение ее в тестировании ПО.

Форма отчетности: отчет с представленной картой и пояснениями данных на карте.

Теоретические основы

В начале тестирования каждого нового программного продукта всегда возникает множество вопросов: что тестировать, как тестировать, когда тестировать, каковы приоритеты тестирования и т.п. При этом обычно руководство желает узнать ответы на эти вопросы от отдела тестирования как можно быстрее.

Mind mapping – это способ создать подробный перечень задач для проекта. Вместо записи перечня действий списком, в линейном, пошаговом формате, используется двумерная (часто цветная) диаграмма, которая представляет мысли, идеи и планы нелинейным образом. Это и называется *интеллект-картой* (*ментальные карты, карты ума, mind map*). Методика была разработана психологом Тони Бьюзеном в конце 1960-х годов. Такой способ записи позволяет визуализировать ассоциативное мышление мозга и развивать обсуждение центральной проблемы. Для тестировщиков – это детально проработанная интеллект-карта – это как раз и есть тот самый план, следуя которому, выполняются все необходимые шаги по обеспечению качества тестируемого продукта.

Примеры их использования:

1. В проектировании тестов.

2. В налаживании коммуникаций.

3. В построении команды.

4. В подготовке стендов.

5. В повышении мотивации команды.

6. В повышении личной мотивации.

Главные правила представления данных на карте следующие:

1. В центре располагается обсуждаемый вопрос, проблема, идея.

2. От центра исходят ветви с узлами, подписанными ключевыми словами, позволяющими вспомнить обсуждаемое понятие – это основные предполагаемые ответы на вопрос, решение проблемы или развитие идеи.

3. Каждый следующий уровень вложенности даёт развитие идеи родительского узла или содержит варианты решения.

4. Создание и чтение справа сверху.

5. Не более 7-8 элементов каждого уровня.

6. Ветки можно помечать разными цветами, указывать приоритеты, расставлять метки и связи, сопровождать картинками и файлами.

7. Каждый блок – это отдельная функциональность, которую необходимо протестировать.

Примерный перечень блоков:

1. Установка ПО.

2. Запуск и работа ПО.

3. Удаление ПО.

4. Обновление ПО

5. Расширение экрана.

6. Выполнение заявленных функций.

7. Пользовательский интерфейс

Конечно, этот перечень можно и нужно расширять. И чем больше аспектов будет учтено, тем более полным будет проведенное тестирование.

Порядок выполнения работы

1. Выберите проект, который будете тестировать (сайт, игра, приложение).

2. Исследуйте проект и нарисуйте его карту.

3. Напишите отчет с разработанной картой с пояснениями.

Контрольные вопросы

1. Что такое интеллект-карта?

2. Когда и зачем они появились?

3. Почему интеллект-карты лучше воспринимаются человеческим мозгом, чем текстовые документы и таблицы?

4. Области применений интеллект-карт.

2.3 Лабораторная работа «Тест-кейсы»

Цель работы: получение практических навыков по разработке тестовых сценариев (тест-кейсов).

Форма отчетности: отчет в виде таблицы.

Теоретические основы

Существует международный стандарт написания тестовой документации – IEEE 829 (IEEE (Institute of Electrical and Electronic Engineers) – организация, созданная в США в 1963 году, разработчик стандартов для

локальных вычислительных систем, в том числе по кабельной системе, физической топологии и методам доступа к среде передачи данных.).

Данный стандарт содержит в себе разделы информации по всем необходимым документам процесса тестирования. Стандарт также определяет форму и содержание тестовых документов. Этот стандарт разрабатывался с 1977 года и был утвержден в 1983 году, а затем вновь подтвержден в 1991,1998, 2008 гг. Несмотря на свою зрелость, он до сих актуален.

Тестовый сценарий или *тест-кейс* (*test-case*) – набор входных значений, предусловий выполнения, ожидаемых результатов и постусловий выполнения, разработанный для определенной цели или тестового условия, таких как выполнения определенного пути программы или же для проверки соответствия определенному требованию [IEEE 610].

Почему тест-кейсы обязательно должны быть на проекте?

1. Не тратится время на «вспоминание» и формулировку шагов теста, а можно просто следовать заранее написанной инструкции.

2. Тест-кейсы являются тем документом, который удобно продемонстрировать заказчику, чтобы показать, что именно тестировалось и каким образом.

3. Тест-кейсы облегчают ввод в процесс новых специалистов, ранее не знакомых с тестируемой системой.

4. Тест-кейсы служат хорошей обучающей базой для неопытных специалистов по тестированию.

5. Наличие тест-кейсов значительно ускоряет регрессионное тестирование.

Недостатки тест-кейсов:

1. Очень много копирования. Тест-кейсы очень похожи друг на друга, первые шаги одинаковые.

2. Сложно поддерживать. Чтобы актуализировать тест-кейсы, надо внести изменения в сотни сценариев, поэтому тест-кейсы должны быть независимые.

3. Неактуальное состояние. Тест-кейсы копируются друг от друга, и часто в них остаются неактуальные части из исходного кейса, которые забыли изменить.

Тест-кейс должен обязательно содержать хотя бы ожидаемый результат (даже, может быть, без описания действий, которые к нему ведут). Кроме ожидаемого результата, необходимо еще пошаговое описание действий, которые позволят прийти к результату фактическому и

сравнить его с ожидаемым. Краткое описание тест-кейса имеет смысл вынести в заголовок.

При написании тест-кейсов следует придерживаться основных правил:

- Начинайте с коротких тест-кейсов.
- Перед написанием детализированных тест-кейсов запишите все, что можно протестировать в вашем приложении в вольной форме.
- Используйте активный залог: («open», «paste», «click»). В русском языке используйте безличную форму: «открыть» (вместо «откройте»).
- Описывайте поведение системы: «появляется окно», «приложение закрывается».
- Используйте простой технический стиль.
- Обязательно указывайте точные названия всех элементов приложения.
- Не объясняйте базовые понятия работы с операционной системой.
- Нужно избегать лишней информации в тест-кейсах, все должно быть коротко и ясно.
- Ни в коем случае нельзя связывать тест-кейсы между собой, так как это создаст дополнительные неудобства при роботе с кейсом, а, тест-кейс, с которым вы сделали привязку могут просто удалить из базы данных, или просто провести в нем корректривы. Так как тест-кейс – это независимый документ.

Что должно быть в тест кейсе?

Приведенный список – это лишь *рекомендация*, каждый тестировщик пишет, как ему удобнее или как это принято в компании.

- ***Уникальное краткое название (ID)***

В это поле записывается номер кейса или номер вместе с какой-то аббревиатурой к примеру «1-П» служит для их уникальной идентификации среди других кейсов.

- ***Предусловие (Pre Conditions)***

Список действий или критерии, которые приводят систему к состоянию пригодному для проведения основной проверки. Сюда можно записать и предварительные шаги, что не относиться к самому тесту. На них можно ссылаться и из других тестов, но сам тест-кейс должен быть независим.

- ***Описание (Summary)***

Это краткое описание проблемы. Описание должно содержать ответ на вопрос что произошло и при каких условиях работает не верно.

- ***Шаги(Steps)***

Здесь описывают шаги, для того чтобы воспроизвести баг. Степы рекомендуют максимально сокращать, то есть найти кратчайший путь для воспроизведения бага и описать в степах, и очень важно чтобы они оставались максимально понятными для разработчиков.

- ***Ожидаемый результат (Expected Result)***

В этом поле описываем ожидаемый результат после хождения по шагам или возможно после конкретных шагов, что бывает реже. Это результат шагов. Он может быть один или несколько. Их можно группировать.

- ***Статус (Pass/Fail)***

Поле служит для проставления статуса каждому тест-кейсу. Если ожидаемый результат совпадает с реальным, то проставляем *pass* (прошёл), в противном случае ставим *fail* (ошибка). Возможно еще несколько статусов в зависимости от процессов и правил в IT-компании.

- ***Постусловие (Post Conditions)***

Список действий, переводящих систему в первоначальное состояние. Не является обязательной частью. Это скорее всего правило хорошего тона. Это особенно актуально при автоматизированном тестировании, когда за один прогон можно наполнить базу данных сотней или даже тысячей некорректных документов.

Тест-кейсы бывают *позитивные* и *негативные*. Но запомните, что сначала проводят позитивное тестирование, а только потом негативное!

Позитивный тест-кейс использует только корректные данные и проверяет, что приложение правильно выполнило вызываемую функцию.

Негативный тест-кейс оперирует как корректными, так и некорректными данными (минимум 1 некорректный параметр) и ставит целью проверку исключительных ситуаций.

Пример 1.

Заголовок: «Проверка того, что программа умеет показывать файлы формата PNG».

Шаг 1. Нажать кнопку «Выбрать файл».

Шаг 2. Выбрать файл с расширением PNG.

Шаг 3. Нажать кнопку «Открыть».

Ожидаемый результат: содержимое файла показано в графическом виде, в полноэкранном режиме.

Здесь сравнение ожидаемого результата и фактического осуществить довольно просто, и критерий соответствия не нужен.

Пример 2.

Рассмотрим позитивный тест-кейс.

ID	Название сценария	Шаги	Ожидаемый результат
1-П	Переключение на русскоязычную версию сайта	1. Открыть сайт http://epayservices.com/	1. Открылась главная страница сайта http://epayservices.com/
		2. Кликнуть на меню смены языка.	2. Кликнув на меню смены языка, выпал список для смены локализации страницы.
		3. Выбрать в выпадающем списке элемент "RU" и нажать.	3. По нажатию на элемент списка "RU" загрузилась русскоязычная версия сайта http://epayservices.com/ru/index.html

Пример 3.

Рассмотрим негативный тест-кейс.

ID	Название сценария	Шаги	Ожидаемый результат
3-Н	Проверка имени на кириллице в регистрационной форме	1. Открыть сайт	1. Открылся сайт
		2. Перейти по ссылке "Sign Up"	2. Нажав ссылку "Sign Up" открылась страница регистрации нового пользователя
		3. Перейти по ссылке "New user registration"	3. После загрузки регистрационной формы по ссылке "New user registration" заполнили поле First Name "Линда"
		4. Ввести в поле First Name "Линда"	4. Активировав кнопку "Register", появилась надпись под полем First Name "may consist of: Latin letters, spaces".
		5. Активировать кнопку "Register"	

В результате мы проверим обработку ошибки!

Порядок выполнения работы

Написать 10 тест-кейсов (5 позитивных и 5 негативных) на любой функционал. Среди них хотя бы один тест-кейс, в котором результат идет на каждый шаг. И один, в котором результат общий после всех шагов. Помним, что одно и то же тестировать смысла нет (10 похожих кейсов, везде проверяя результат на каждый шаг, бесполезны).

Тест-кейсы представить в виде таблицы.

Контрольные вопросы

1. Что такое тест-кейс?
2. Что должен включать в себя тест-кейс?
3. Почему необходимо сначала выполнять позитивное тестирование, а не негативное?
4. Опишите плюсы и минусы тест-кейсов.

2.4 Лабораторная работа «Ручное тестирование»

Цель работы: овладение навыками ручного тестирования.

Форма отчетности: отчет, в котором описывается ход и результаты ручного тестирования.

Теоретические основы

Ручное тестирование (manual testing) – часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно проводится тестировщиками путем моделирования возможных сценариев действия пользователя.

Преимущества ручного тестирования:

- Вручную можно протестировать практически любое приложение, в то время как автоматизировать стоит только стабильные системы.
- Ручное тестирование – самый низкоуровневый и простой тип тестирования, не требующих большого количества дополнительных знаний.
- Ручное тестирование пользовательского интерфейса удобно тем, что контроль корректности интерфейса проводится человеком, т.е. основным «потребителем» данной части программной системы.

При этом ручное тестирование имеет и существенный *недостаток* – для его проведения требуются значительные человеческие и временные ресурсы. Особенно сильно этот недостаток проявляется при проведении

регрессионного тестирования, когда на каждой итерации тестирования пользовательского интерфейса требуется участие тестировщика.

Ad hoc тестирование (разовое тестирование) направлено на проверку одного аспекта программы. Для такого типа тестирования нет формально определённых правил, оно проводится импровизационно – тестировщик может использовать любые доступные средства для поиска дефектов. Фактически adhoc-тесты – это попытка «угадать» возможную ошибку.

Порядок выполнения работы

1. Выполнить тестовые сценарии (взять тест-кейсы из лабораторной «Тест-кейсы»).

2. Проверить, соответствует ли полученный результат ожидаемому.

3. Описать дефект, если был найден.

4. Выполнить описанные шаги и попробовать воспроизвести дефект повторно.

Контрольные вопросы

1. Что такое ручное тестирование?

2. Укажите преимущество и недостатки ручного тестирования.

3. Для каких видов тестирования можно проводить ручное тестирование?

2.5 Лабораторная работа «Локализация дефектов»

Цель работы: получение практических навыков написания отчетов об инцидентах.

Форма отчетности: отчет об инцидентах.

Теоретические основы

Дефект (баг, bug) – изъян в разработке программного обеспечения, который вызывает несоответствие ожидаемых результатов выполнения программы и фактически полученных результатов.

Основная проблема при описании бага – это его локализация. *Локализация бага* – найти и описать такие условия, при котором он повторяется. Поиск причины возникновения бага!!!

Отчет об инциденте (баг репорт, Bug Report) – это документ, описывающий ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

В отчете необходимо указать серьезность и приоритет дефекта.

Выставляя *серьезность* (*severity*) дефекта тестировщик оценивает его влияние на работоспособность ПО. Чем выше *severity*, тем масштабнее негативные последствия данного дефекта.

Градация серьезности дефекта (*Severity*) следующая:

- *S1. Блокирующая (Blocker).* Блокирующая ошибка, приводящая приложение в нерабочее состояние, в результате которого дальнейшая работа с тестируемой системой или ее ключевыми функциями становится невозможна.
- *S2. Критическая (Critical).* Критическая ошибка, неправильно работающая ключевая бизнес логика, дыра в системе безопасности, проблема, приведшая к временному падению сервера или приводящая в нерабочее состояние некоторую часть системы, без возможности решения проблемы, используя другие входные точки. Решение проблемы необходимо для дальнейшей работы с ключевыми функциями тестируемой системой.
- *S3. Значительная (Major).* Значительная ошибка, часть основной бизнес логики работает некорректно. Ошибка не критична или есть возможность для работы с тестируемой функцией, используя другие входные точки.
- *S4. Незначительная (Minor).* Незначительная ошибка, не нарушающая бизнес логику тестируемой части приложения, очевидная проблема пользовательского интерфейса.
- *S5. Тривиальная (Trivial).* Тривиальная ошибка, не касающаяся бизнес логики приложения, плохо воспроизводимая проблема, малозаметная посредствам пользовательского интерфейса, проблема сторонних библиотек или сервисов, проблема, не оказывающая никакого влияния на общее качество продукта.

Приоритет (Priority) дефекта – это инструмент менеджера по планированию работ. Чем выше *priority*, тем быстрее нужно исправить дефект.

Например, слово, напечатанное с ошибкой, может иметь самый низкий уровень *severity*, но перед выпуском продукта этот дефект может иметь наивысший приоритет и должен быть экстренно исправлен (если вдруг окажется, что на вашем интернет портале имя владельца компании напечатано с ошибкой.

Градация приоритета дефекта (*Priority*) следующая:

- *P1 Высокий (High).* Ошибка должна быть исправлена как можно быстрее, т.к. ее наличие является критической для проекта.

• *P2 Средний (Medium)*. Ошибка должна быть исправлена, ее наличие не является критичной, но требует обязательного решения.

• *P3 Низкий (Low)*. Ошибка должна быть исправлена, ее наличие не является критичной, и не требует срочного решения.

Описание дефекта должно иметь следующую структуру:

1. Уникальный номер (ID).

2. Краткое название (Title, Summary или Short Description): короткий текст, который помогает сразу понять, что это за дефект.

3. Описание (Description): полное описание дефекта включая шаги для воспроизведения.

4. Окружение: ОС, версия продукта на котором был найден дефект, браузер, патчи, т.е. конфигурация системы, на который дефект был обнаружен.

5. Attachments: некоторые дефекты сложно описать, для простоты делаются скриншоты, видео, или лог-файлы и прикладываются к описанию ошибки для наглядности.

6. Серьезность (Severity) дефекта.

7. Приоритет (Priority).

Если дефект описан согласно данной схеме, то он вызовет меньше всего вопросов, и разработчик, не теряя время на дополнительные разъяснения, приступит к его исправлению.

Порядок выполнения работы

1. Найти и локализовать дефекты в проекте.

2. Оформить по шаблону: Тема – заголовок бага / улучшения, Описание – описание задачи (шаги, результат, ожидаемый в баге или все описание улучшения), Ссылка на баг-исходник. Исполнитель – «назначить (Ю. Морозова)».

Контрольные вопросы

1. Что такое дефект?

2. Какие виды дефектов?

3. Что такое локализация дефекта?

2.6 Лабораторная работа «Классификация тестов»

Цель работы: получение практических навыков по разработке тест-кейсов для разных видов тестирования.

Форма отчетности: отчет с таблицей тест-кейсов.

Теоретические основы

Классификация тестов позволяет упорядочить знания и значительно ускоряет процессы планирования тестирования и разработки тест-кейсов.

Для каждого уровня тестирования может быть определено: цели, артефакты процесса разработки, на основании которых будут разработаны тестовые сценарии, объекты тестирования, типичные дефекты и отказы, которые могут быть найдены во время тестирования.

Все виды тестирования можно условно разделить по *состоянию системы* на две большие группы:

1. статическое тестирование (*static testing*);

2. динамическое тестирование (*dynamic testing*).

Статическое тестирование – это процесс анализа самой разработки программного обеспечения, т. е. тестирование без запуска программы.

Динамическое тестирование – это тестовая деятельность, предусматривающая эксплуатацию (запуск) программного продукта. Динамическое тестирование предполагает запуск программы, выполнение всех её функциональных модулей и сравнение фактического её поведения с ожидаемым.

Тестирование ПО можно классифицировать по следующим признакам:

1. По знанию системы.

2. По позитивности.

3. По целям (объекту).

4. По исполнителям (субъекту).

5. По времени проведения.

6. По степени автоматизации.

1. По знанию системы тестирование можно проводить методами *черного* и *белого ящиков*.

Метод чёрного ящика (black box testing, closed box testing, specification-based testing) – у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их понимания, либо он сознательно не обращается к ним в процессе тестирования.

Метод белого ящика (white box testing, open box testing, clear box testing, glass box testing) – у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.

2. По критерию «*позитивности*» сценариев:

- позитивное тестирование *(positive testing);*
- негативное тестирование *(negative testing).*

Позитивное тестирование проводится по сценариям, предполагающим нормальную, «правильную» работу системы в заведомо корректных условиях.

Негативное тестирование использует сценарии, проверяющие ситуации, связанные с потенциальными дефектами в системе. Например, проверка работы программы при вводе недействительных данных.

3. По *целям (или объекту)* разделяют:

- нефункциональное тестирование;
- функциональное тестирование.

Нефункциональное тестирование включает, но не ограничивается, нагрузочное тестирование, тестирование производительности, стресс-тестирование, тестирование удобства использования, тестирование восстановления, тестирование надежности и тестирование переносимости. Это тестирование того, «как» система работает.

Тестирование производительности (performance testing) – исследование показателей скорости реакции приложения на внешние воздействия при различной по характеру и интенсивности нагрузке.

В рамках тестирования производительности выделяют следующие подвиды:

- *Нагрузочное тестирование (load testing, capacity testing)* – исследование способности приложения сохранять заданные показатели качества при нагрузке в допустимых пределах и некотором превышении этих пределов (определение «запаса прочности»).

- *Стрессовое тестирование (stress testing)* – исследование поведения приложения при нештатных изменениях нагрузки, значительно превышающих расчётный уровень, или в ситуациях недоступности значительной части необходимых приложению ресурсов. Стрессовое тестирование может выполняться и вне контекста нагрузочного тестирования: тогда оно, как правило, называется «*тестированием на разрушение*» (*destructive testing*) и представляет собой крайнюю форму негативного тестирования.

Тестирование интерфейса пользователя (GUI testing) – это тестирование, при котором проверяются элементы интерфейса пользователя.

Тестирование удобства использования (юзабилити, usability testing) выполняется с целью определения, насколько органично используется пользовательский интерфейс целевыми пользователями, т. е. проверяется интуитивность интерфейса. Юзабилити-тестирование часто проводится путем привлечения группы потенциальных пользователей с целью собрать впечатления от работы с системой.

Важно понимать разницу между тестированием интерфейса пользователя и тестированием удобства использования. *Тестирование удобства использования (usability testing) и тестирование интерфейса пользователя (GUI testing)* – не одно и то же! Например, корректно работающий интерфейс может быть неудобным, а удобный может работать некорректно.

Тестирование интернационализации (internationalisation testing) – вид тестирования, при котором проверяется готовность приложения к работе с различными языковыми настройками, в частности способность корректно отображать шрифты, пункты меню, производить поиск, сортировку, способность приложения обрабатывать файлы, поименованные на различных языках.

Тестирование совместимости (*compatibility testing*) – это вид тестирования, основной целью которого является проверка качественной работы разрабатываемого программного средства с другим ПО (операционными системами, браузерами и т. д.). Тестирование с разными браузерами называется *кросс-браузерным тестированием (cross-browser testing)*. Тестирование с разными операционными системами называется *кросс-платформенным тестированием (cross-platform testing)*.

Функциональное тестирование (functional testing) - тестирование, основанное на анализе спецификации функциональности компонента или системы.

Функциональные тесты основываются на функциях, выполняемых системой, и могут проводиться на всех уровнях тестирования (*модульном, интеграционном, системном, приемочном*).

Компонентное тестирование (*модульное*) занимается поиском дефектов и верификацией функционирования программных модулей, программ, объектов, классов и т.п., которые можно протестировать *изолированно*. Это может быть сделано изолированно от остальной части системы, в зависимости от контекста ЖЦ разработки и системы. В процессе могут быть использованы заглушки, драйвера и эмуляторы.

Следующий уровень интеграционное тестирование.

Интеграционное тестирование проводится после компонентного тестирования и направлено на выявление дефектов *взаимодействия* различных подсистем на уровне потоков управления и обмена данными.

Основная задача интеграционного тестирования – поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения интеграционное тестирование является количественным развитием модульного, поскольку так же, как и модульное тестирование, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (*stub*) на месте отсутствующих модулей.

Заглушка – это имитация вызываемой функции, возвращающая те же данные, но ничего больше не делающая.

Следующий уровень – это системное тестирование.

Системное тестирование (system testing) – процесс тестирования системы в целом с целью проверки того, что она соответствует установленным требованиям. Системное тестирование производится над проектом в целом с помощью метода «черного ящика». Системное тестирование чаще всего выполняет независимая тестовая команда.

Еще можно выделить приемочное тестирование.

Приёмочное тестирование (acceptance testing) – формальное тестирование по отношению к потребностям, требованиям и бизнес процессам пользователя, проводимое с целью определения соответствия системы критериям приёмки и дать возможность пользователям, заказчикам или иным авторизированым лицам определить, принимать систему или нет. [IEEE 610].

4. По *исполнителям* (субъекту):

- тестирование разработчиками или специалистами до передачи пользователю или *альфа-тестирование (alpha testing)*;
- тестирование пользователями после передачи пользователю или *бета-тестирование (beta testing)*.

5. По *времени проведения* тестирования выделяют:

- Дымное (*smoky*)
- Санитарное (*sanity*)
- Регрессионное (*regression*)

Дымное тестирование – проверка самой важной, самой ключевой функциональности, неработоспособность которой делает бессмысленной

саму идею использования приложения. Ежедневная сборка и «тест на дым» являются передовыми практическими методами.

Санитарное тестирование относится к виду тестирования, которое используется с целью доказательства работоспособности конкретной функции или модуля согласно заявленным техническим требованиям.

6. По *степени автоматизации*:

- Ручное (*manual testing*).
- Автоматизированное (*automated testing*).

Порядок выполнения работы

1. Написать 10 тест-кейсов (5 для функционального и 5 для нефункционального тестирования).

2. Выполнить тест-кейсы.

3. Заполнить отчет об инцидентах, если обнаружены были дефекты.

4. Отчет выслать в виде таблицы с тест-кейсами с указанием вида тестирования.

Контрольные вопросы

1. Чем отличаются функциональное тестирование от нефункционального?

2. На какие уровни можно разделить функциональное тестирование?

3. В чем заключается разница между юзабилити тестированием и тестирование GUI?

4. Какие тесты лучшие кандидаты для автоматизации?

2.7 Лабораторная работа «Попарное тестирование»

Цель работы: получение практических навыков тестирования методом черного ящика.

Форма отчетности: отчет с таблицей тест-кейсов, полученных путем попарного тестирования.

Теоретические основы

Чтобы облегчить жизнь тестировщику были разработаны различные техники и методы тест-дизайна, которые позволяют приблизится к исчерпывающему (полному) тестированию.

Метод чёрного ящика (black box testing, closed box testing, specification-based testing) – у тестировщика либо нет доступа к внутренней структуре и коду приложения, либо недостаточно знаний для их по-

нимания, либо он сознательно не обращается к ним в процессе тестирования. Тестировщик не знает, как устроена тестируемая система.

Целью этой техники является поиск ошибок в таких категориях:

- неправильно реализованные или недостающие функции;
- ошибки интерфейса;
- ошибки в структурах данных или организации доступа к внешним базам данных;
- ошибки поведения или недостаточная производительности системы.

*Преимущества***:**

- тестирование производится с позиции конечного пользователя и может помочь обнаружить неточности и противоречия в спецификации;
- тестировщику нет необходимости знать языки программирования и углубляться в особенности реализации программы;
- тестирование может производиться специалистами, независимыми от отдела разработки, что помогает избежать предвзятого отношения;
- можно начинать писать тест-кейсы, как только готова спецификация.

Недостатки:

- тестируется только очень ограниченное количество путей выполнения программы;
- без четкой спецификации (а это скорее реальность на многих проектах) достаточно трудно составить эффективные тест-кейсы;
- некоторые тесты могут оказаться избыточными, если они уже были проведены разработчиком на уровне модульного тестирования;

Техники тест-дизайна, основанные на использования *черного ящика*:

- разбиение на классы эквивалентности;
- анализ граничных значений;
- попарное тестирование;
- таблицы решений.

Pairwise testing (*all-pairs analysis, попарное тестирование или попарный анализ, анализ всех пар комбинаций*) – это современная и эффективная методика тестирования, основанная на том предположении, что большинство дефектов возникает при взаимодействии не более двух факторов. Тестовые наборы, генерируемые при использовании данной мето-

дики, охватывают все уникальные пары комбинаций факторов, что считается достаточным для обнаружения большего числа дефектов.

Для попарного тестирования используются алгоритмы, основанные на построении ортогональных массивов или на *All-Pairs алгоритме*, которые опираются на теоретические исследования в области комбинаторных алгоритмов, алгоритмов дискретной математики.

All-Pairs algorithm (алгоритм всех пар) – это комбинаторная методика, которая была специально создана для попарного тестирования. В её основе лежит выбор возможных комбинаций значений всех переменных, в которых содержатся все возможные значения для каждой пары переменных.

Плюсы попарного тестирования следующие:

- Данный тип проверки уменьшает количество тест-кейсов необходимых для проверки продукта.
- Попарное тестирование ускоряет выполнение самого процесса контроля качества продукта.
- Практика показывает, что количество багов, обнаруженных с помощью попарного тестирования, будет больше, чем при проверке всех значений для каждого параметра ввода.

Есть множество инструментов для попарного тестирования. Есть и онлайновые сервисы: hexawise, inductive, testcover. Некоторые работают через консоль, другие через GUI. Главное – подать им на вход грамотный набор данных. Удобно использовать консольную программу *Allpairs*, она не привязана ни к одной операционной системе.

Allpairs – программа, подбирающая уникальные пары для входящего набора данных. Работает из командной строки.

Порядок выполнения работы

1. Найти в своем проекте место, где может быть применим pairwise (много переменных, мало значений в каждой).

2. Составить входной файл, разбив входные данные на классы эквивалентности.

3. Выполнить программой *Allpairs* попарное тестирование.

4. Проанализировать результат.

5. Отправить отчет с входной и выходной таблицами с пояснениями.

Контрольные вопросы

1. В чем особенность метода черного ящика?

2. Когда применяют метод черного ящика?

3. Перечислите и опишите техники черного ящика.

4. Преимущества и недостатки попарного тестирования.

2.8 Лабораторная работа «Модульное тестирование»

Цель работы: овладение практических навыков выполнять модульное тестирование.

Форма отчетности: отчет с листингом программы и юнит-тестами.

Теоретические основы

Метод белого ящика (white box testing, open box testing, clear box testing, glass box testing) – метод, когда у тестировщика есть доступ к внутренней структуре и коду приложения, а также есть достаточно знаний для понимания увиденного.

Разработка тестов методом белого ящика (white-box test design technique) – процедура разработки или выбора тестовых сценариев на основании анализа внутренней структуры компонента или системы.

Тестирование методом белого ящика, основывается на конкретной структуре программного продукта или системы:

- Компонентный уровень: структура компонента программного обеспечения, т.е. операторы, альтернативы, ветви или определенные пути.
- Интеграционный уровень: структура может быть представлена деревом вызовов (диаграмма, в которой модули вызывают другие модули).
- Системный уровень: структура может представлять собой структуру меню, бизнес-процессов или же схему веб-страницы.

Техники, основанные на структуре, или методе белого ящика:

- покрытие операторов;
- покрытие альтернатив;
- покрытие решений.

Покрытие кода (code coverage) – метод анализа, определяющий, какие части программного обеспечения были проверены (покрыты) набором тестов, а какие нет, например, покрытие операторов, покрытие альтернатив или покрытие условий.

Покрытие операторов (statement coverage) – процентное отношение операторов, исполняемых набором тестов, к их общему количеству.

При тестировании операторов тестовые сценарии создаются таким образом, чтобы выполнять определенные операторы и обычно увеличивать покрытие операторов. Величина покрытия операторов определяется

как отношение числа выполняемых операторов, покрытых тестовыми сценариями (разработанными или выполненными) к общему числу операторов в тестируемом коде.

Покрытие альтернатив (decision coverage) – процент результатов альтернативы, который был проверен набором тестов. Стопроцентное покрытие решений подразумевает стопроцентное покрытие ветвей и стопроцентное покрытие операторов.

В методе тестирования альтернатив тестовые сценарии создаются для выполнения определенных результатов альтернатив. Ветви исходят из точек альтернатив в программном коде и показывают передачу управления различным участкам кода. Покрытие альтернатив определяется отношением числа всех результатов альтернатив, покрытых разработанными или выполненными тестовыми сценариями к числу всех возможных результатов альтернатив в тестируемом коде.

Модульное тестирование, или *юнит-тестировние* (*unit testing*) – процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии*, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель *модульного тестирования* – изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Существует различные инструменты как *JUnit, PHPUnit, TestNG, PyTest*, которые позволяют создавать и поддерживать качественные юнит-тесты.

Рекомендуется использовать следующие инструменты:

- JUnit.
- TestN.
- Code coverage.

JUnit – это Java фреймворк для тестирования, т. е. тестирования отдельных участков кода, например, методов или классов. Опыт, полученный при работе с JUnit, важен в разработке концепций тестирования программного обеспечения.

Junit предоставляет нам следующие возможности:

- Базовые классы и Аннотации для написания юнит-тестов.

- Базовый класс для запуска тестов – TestRunner class.
- Поддержка классов и аннотаций для написания тест-сьютов – @RunWith(Suite.Class).
- Отчет результатов тестирования.

JUnit Annotations – процесс добавления специальных синтаксических форм метаданных в Java. JUnit Annotations: переменные, параметры, пакеты, методы и классы.

Проверки чаще всего выполняются с помощью класса **Assert** хотя иногда используют ключевое слово assert.

Чтобы ускорить процесс и сделать его более автоматизированным используют параметризированные тесты (@Parameterized.Parameters). С их помощью можно создать тестовой класс, и тестировать модуль, используя различные данные с помощью тестового класса

С помощью @RunWith можно аннотировать тестовый класс, передав этой аннотации параметром значение *Parameterized.class*.

```
   @RunWith(value = Parameterized.class)
   public class ParametersTest {
// ...
}
```

TestNG – это фреймворк для тестирования, написанный Java, он взял много чего с JUnit и NUnit, но имеет более гибкий и расширенный функционал.

Принцип работы с TestNG очень схож с JUnit.

Написать автотесты – это еще полдела, необходимо проверить, а весь ли код покрыт тестами. Автоматические тесты должны покрывать 100% функционала. Данная характеристика называется «*code coverage*» и буквально означает степень покрытия кода тестами.

Различаются следующие показатели:

- Function Coverage – подсчёт по вызовам методов
- Decision Coverage – подсчёт по возможным направлениям исполнения кода (then-else или case-case-default в управляющих структурах). Учитывает единственное ветвление в каждом конкретном случае.
- Statement Coverage – подсчёт по конкретным строчкам кода
- Path Coverage – подсчёт по возможным путям исполнения кода. Более широкое понятие, чем decision coverage, так как учитывает результат всех ветвлений.

- Conditional Coverage – подсчёт по возможным результатам вычисления значениям булевских выражений и подвыражений в коде.

Покрытие кода (инструменты):

- Java (Jcov, *EclEmma – Java Code Coverage for Eclipse*).
- C++ (Gcov, Tcov).
- C# (OpenCover).
- встроенный в Visual Studio Test Coverage.

EclEmma – бесплатный инструмент покрытия кода на Java для среды *Eclipse*, доступный по лицензии Eclipse Public License.

Достоинство *EclEmma*:

- Доступный анализ покрытия кода прямо в IDE Eclipse.
- Запуски тестов типа JUnit в Eclipse могут быть проанализированы напрямую на предмет наличия покрытия кода.
- Не требует модификации проектов или выполнения всяких установок.
- Результат оценки покрытия виден в редакторе кода, при этом настраиваемый цвет кода показывает полностью покрытые строки кода, частично покрытые и не покрытые тестами строки кода.
- Позволяет экспортировать **отчёты о покрытии:** данные о покрытии могут быть экспортированы в формате HTML, XML или CSV.

Следующие типы кода для запуска поддерживают EclEmma:

- Local Java application.
- Eclipse/RCP application.
- Equinox OSGi framework.
- JUnit test.
- TestNG test.
- JUnit plug-in test.
- JUnit RAP test.
- SWTBot test.
- Scala application.

Пример

Рассмотрим пример как можно протестировать код.

Шаг 1. Создадим проект.

Шаг 2. Напишем код или метод, который будем тестировать.

```
package test_meth;
public class method {
public static int alg(int A, int B, int X) {
      //точка a
      if((A>1)&&(B==0))
      X=X/A;//точка c
      //точка b
      if((A==2)||(X>1))
      X=X++; //точка e
      //точка d
      return X;
      } }
```

Шаг 3. Напишем тесты. Постараемся добиться полного покрытия кода. Добавим в проект **JUnit**.

```
package test_meth;
import static org.junit.Assert.*;
import org.junit.Test;
public class methodTest {
      @Test
      public void test1() {
assertNotNull(null, new method().alg(2, 0, 4));
            }
      @Test
      public void test2() {
assertNotNull(null, new method().alg(2, 1, 1));
      }
      @Test
      public void test3() {
assertNotNull(null, new method().alg(1, 0, 2));
      }
      @Test
      public void test4() {
assertNotNull(null, new method().alg(1, 1, 1));
            }
      @Test
      public void test5() {
assertNotNull(null, new method().alg(0, 0, 0));
      } }
```

Запустим (рис. 1).

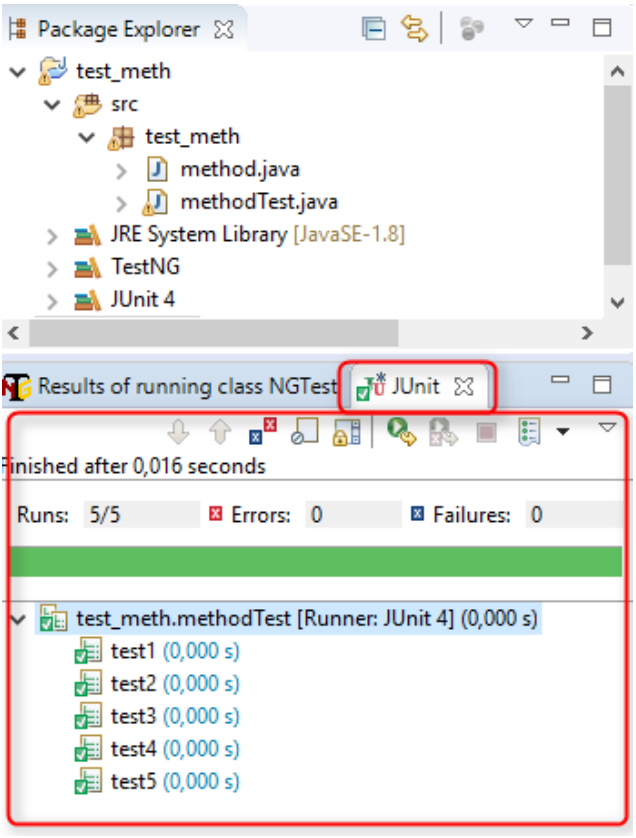

Рис. 1

Можно запускать тесты вручную с помощью программного кода. Для этого можно воспользоваться **Runner**. Бывают текстовый – junit.textui.TestRunner, графические версии – junit.swingui.TestRunner, junit.awtui.TestRunner, или класс JUnitCore.

Добавим следующий метод в метод main() в класс methodTest, получим:

```
package test_meth;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
public class methodTest {
```

```
      @Test
      public void test1() {
            new method();
assertNotNull(null, method.alg(2, 0, 4));
            }
      @Test
      public void test2() {
            new method();
assertNotNull(null, method.alg(2, 1, 1));
      }
      @Test
      public void test3() {
            new method();
assertNotNull(null, method.alg(1, 0, 2));
      }
      @Test
      public void test4() {
            new method();
assertNotNull(null, method.alg(1, 1, 1));
      }
      @Test
      public void test5() {
      new method();
assertNotNull(null, method.alg(0, 0, 0));
      }
    public static void main(String[] args) throws
Exception {
       JUnitCore runner = new JUnitCore();
Result result=runner.run(methodTest.class);

System.out.println("run tests: " +
result.getRunCount());
System.out.println("failed tests: " +
result.getFailureCount());
System.out.println("ignored tests: " +
result.getIgnoreCount());
System.out.println("success: " +
result.wasSuccessful());
      } }
```

Результат выводится на консоль IDE (рис. 2).

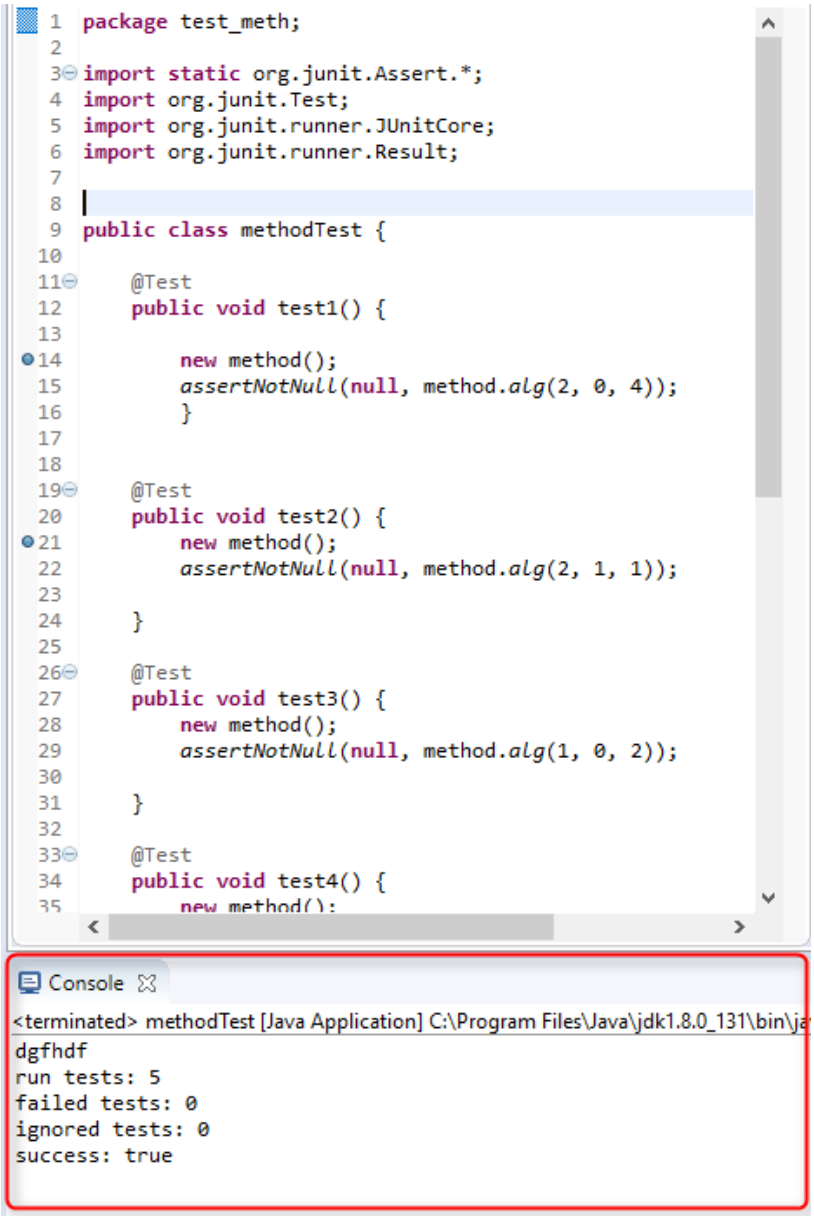
package test_meth;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
public class methodTest {
@Test
public void test1() {
new method();
assertNotNull(null, method.alg(2, 0, 4));
}
@Test
public void test2() {
new method();
assertNotNull(null, method.alg(2, 1, 1));
}
@Test
public void test3() {
new method();
assertNotNull(null, method.alg(1, 0, 2));
}
@Test
public void test4() {
Console
<terminated> methodTest [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\ja
dgfhdf
run tests: 5
failed tests: 0
ignored tests: 0
success: true

Рис. 2

Писать 5 тестов для одного метода слишком долго, а бывает их не 5, а гораздо больше. Лучше воспользоваться параметризированными тестами. Перепишем тесты:

```
package test_meth;
import static org.junit.Assert.*;
import org.junit.Test;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.testng.Assert;
import java.util.Arrays;
@RunWith(Parameterized.class)
public class methodTest {
    private int valueA;
    private int valueB;
    private int valueX;
    private int expected;
    public methodTest(int valueA, int valueB, int
valueX, int expected) {
        this.valueA = valueA;
        this.valueB = valueB;
        this.valueX = valueX;
        this.expected = expected;
    }
@Parameterized.Parameters(name = "{index}: re-
zOf({0},{1},{2}=={3}")
public static Iterable<Object[]> dataForTest() {
        return Arrays.asList(new Object[][]{
                {2, 0, 4, 2},
                {2, 1, 1, 1},
                {1, 0, 2, 2},
                {1, 1, 1, 1},
                {0, 0, 0, 0}
        });
    }
      @Test
        public void test_peram() {
        new method();

Asert.assertEquals(expected,method.alg(valueA,valueB
, valueX));
```

```
    }

    public static void main(String[] args) throws
Exception {
 JUnitCore runner = new JUnitCore();
 Result result = runner.run(methodTest.class);

 System.out.println("run tests: " + re-
sult.getRunCount());
 System.out.println("failed tests: " + re-
sult.getFailureCount());
 System.out.println("ignored tests: " + re-
sult.getIgnoreCount());
 System.out.println("success: " + re-
sult.wasSuccessful());


    } }
```

Результат выполнения параметризированных тестов показан на рисунке 3.

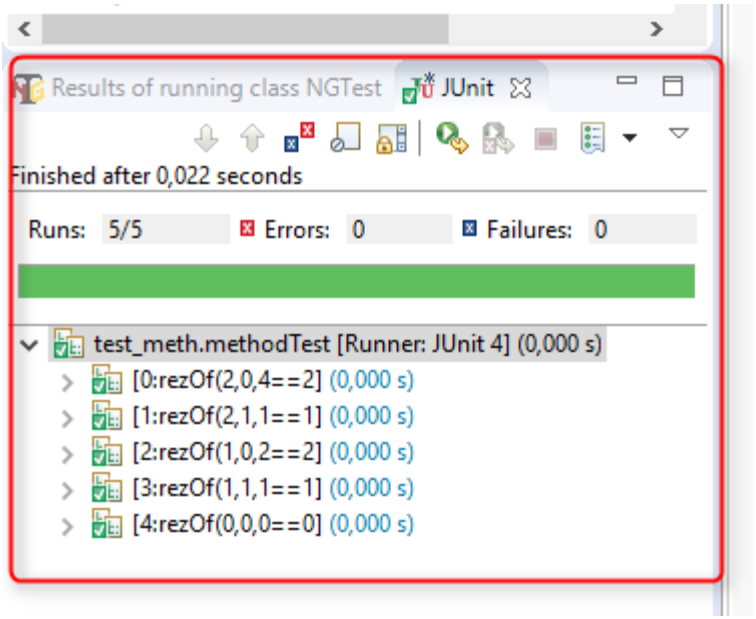

Рис. 3

Теперь давайте оценим покрытие кода (рис. 4).

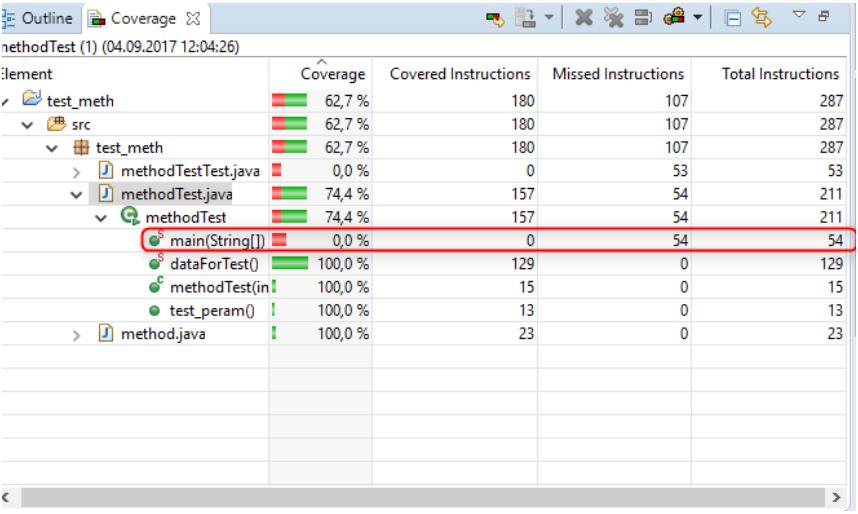

Рис. 4

Как видим, имеем покрытие нашего метода methodTest() 100%.

Отчет по лабораторной работе должен включать:

1. Вариант задания.
2. Листинг программы.
3. Листинг и описание юнит-тестов.
4. Скриншот прохождения тестов.
5. Скриншот анализа покрытия кода.
6. Вывод о результатах тестирования.

Варианты задания:

Вариант 1. Провести функциональное тестирование программы, которая решает квадратное уравнение.

Вариант 2. Провести функциональное тестирование программы, которая определяет вид треугольника, заданного длинами его сторон: равносторонний, равнобедренный, прямоугольный, разносторонний.

Вариант 3. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит максимальное значение элемента, минимальное значение элемента и их сумму.

Вариант 4. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит минимальное значение элемента, устанавливает, сколько раз это значение встречается в последовательности.

Вариант 5. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит значение элемента, повторяющееся большее число раз и выводит количество повторений в последовательности.

Вариант 6. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит разность между максимальным значением элемента и минимальным значением элемента.

Вариант 7. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит максимальное значение элемента, минимальное значение элемента и их произведение.

Вариант 8. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит минимальное значение элемента и проверяет, является ли это число простым.

Вариант 9. Провести функциональное тестирование программы, которая определяет вид четырехугольника, заданного координатами вершин на плоскости: квадрат, прямоугольник, параллелограмм, ромб, равнобедренная трапеция, прямоугольная трапеция, трапеция общего вида, четырехугольник общего вида.

Вариант 10. Провести функциональное тестирование программы, которая из последовательности 10 целых чисел выводит максимальное значение элемента и проверяет, является ли это число простым.

Контрольные вопросы

1. Назовите уровни функционального тестирования.

2. Что проверяют юнит-тестами?

3. Какие техники, основанные на структуре, или методе белого ящика?

4. Что означает 100% покрытие кода?

2.9 Лабораторная работа «Автоматизированное тестирование»

Цель работы: получение практических навыков по автоматизированному тестированию web-приложений с использованием инструмента Selenium IDE. Знакомство с локаторами и методами нахождения элементов в структуре документа.

Форма отчетности: отчет должен включать тест-сьют для тестируемого приложения.

Теоретические основы

Selenium IDE (Integrated Development Environment, интегрированная среда разработки) – это инструмент, используемый для разработки тестовых сценариев.

Он представляет собой простое в использовании дополнение к браузеру Firefox и, в целом, является наиболее эффективным способом разработки тестовых сценариев.

Тест-кейс в Selenium – набор команд прикладного уровня, имитирующих действия пользователя в web-приложении.

Selenium сохраняет файлы с тест-кейсами в обычных HTML-файлах с простой структурой, содержащей одну таблицу из трех колонок, что позволяет редактировать тесты в любом редакторе:

- *Test head* – заголовок теста,
- *Command* – команда языка Selenium,
- *Target* – цель, это элемент, над которым должно выполняться действие (обычно указывается как xPath на элемент),
- *Value* – параметр, при необходимости передаваемый в команду.

После прогонки тест-сьютов Selenium можно ознакомиться с лог-файлом – отчётом по тестированию, который включает в себя:

- общий результат прогонки тестового набора (passed / failed);
- общее время тестирования;
- общее число выполненных тест-кейсов, и число успешных и неуспешных из них;
- число успешных, неуспешных тестовых команд и команд с ошибками.

Панель инструментов (Toolbar) или панель управления тестами

На панели инструментов (рис. 5) находятся кнопки, с помощью которых можно управлять выполнением тестовых сценариев, в том числе пошаговым выполнением для отладки. Крайняя правая кнопка, на которой изображена красная точка – это кнопка записи.

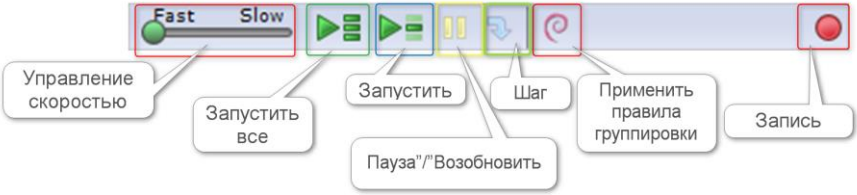

Рис. 5

Главное достоинство Selenium – запись действий пользователя в браузере.

Панель тестового сценария

На панели (рис. 6) отображается набор команд Selenium, составляющих тестовый сценарий. На ней расположены две вкладки, первая из которых, «Table», отображает команды и их параметры в удобном для восприятия табличном виде.

Command	Target	Value
open	/	
waitForPageToLoad		
clickAndWait	xpath=id('menu_download')/a	
assertTitle	Downloads	
verifyText	xpath=id('mainContent')/h2	Downloads

Рис. 6

Поля ввода данных «Command», «Target» и «Value» отображают выбранную в данный момент команду, а также ее параметры (рис. 7). С помощью этих полей можно модифицировать выбранную команду. Значение первого параметра, описанного во вкладке «Reference» нижней панели, указывается в поле «Цель». Если в «Справке» описан также второй параметр, то он всегда указывается в поле «Значение».

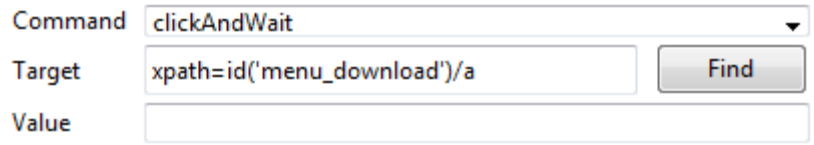

Рис. 7

Нижняя панель используется для четырёх различных функций: лога, справки, документации по UI-Element и группирования.

За ходом и результатом выполнения тестов можно следить с помощью поля log, в котором отражаются все выполняемые Selenium IDE действия (рис. 8).

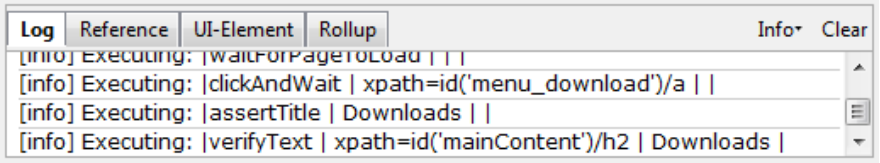

Рис. 8

Вкладка «Reference» выбирается по умолчанию каждый раз, когда пользователь вводит или модифицирует команды и параметры в табличном режиме и отображает информацию о текущей команде (рис. 9).

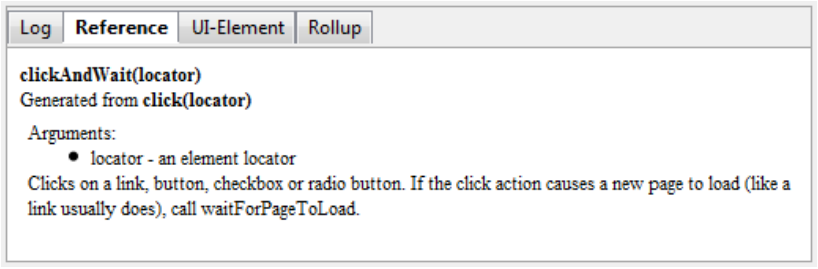

Рис. 9

BaseURL – это значение домена, для которого будет создаваться тест.

Список типичных команд, самых востребованных при создании тест-кейсов:

1. *Действия* – команды, которые обычно управляют состоянием приложения. Они совершают действия вроде «щелкнуть по той или иной ссылке» или «выбрать опцию».

2. *Считыватели* – анализируют состояние приложения и сохраняют результаты в переменные.

3. *Проверки* – проверяют соответствие состояния приложения ожидаемому.

Порядок выполнения работы

1. Познакомиться с панелями Selenium. Провести пошаговое выполнение тестового сценария по одной команде за раз или по шагам:

- Запустить тестовый сценарий с помощью кнопки «Run» на панели инструментов.
- Сразу же остановить выполнение тестового сценария, нажав на кнопку «Pause».
- Выполнить тест по шагам, нажав на кнопку «Step».

2. Разработать функциональные сценарии.

- Провести тестирование регистрации
- Провести тестирование поиска на сайте.
- Провести тестирование локализации.

Если нет возможности провести автоматизированное тестирование в проекте, возьмите сайт **tusur.ru.**

3. Выслать в качестве отчета тест-сьют Selenium IDE.

Контрольные вопросы

1. Что такое автоматизированное тестирование?

2. Какие команды можно выполнять в Selenium?

3. На каком языке можно экспортировать тест-сьют из Selenium?

3 Методические указания для организации самостоятельной работы

3.1 Общие положения

Целью самостоятельной работы является систематизация, расширение и закрепление теоретических знаний, использование материала, собранного и полученного в ходе самостоятельной подготовки к лабораторным работам.

Самостоятельная работа включает в себя подготовку к лабораторным работам, проработку лекционного материала и подготовку к контрольным работам, проработку тем дисциплины, вынесенных на самостоятельное изучение.

3.2 Проработка лекционного материала и подготовка к контрольным работам

Изучение теоретической части дисциплин призвано не только углубить и закрепить знания, полученные на аудиторных занятиях, но и способствовать развитию у студентов творческих навыков, инициативы и организовать свое время.

Проработка лекционного материала включает:

- чтение студентами рекомендованной литературы и усвоение теоретического материала дисциплины;
- знакомство с Интернет-источниками;
- подготовку к различным формам контроля (контрольные работы);
- подготовку ответов на вопросы по различным темам дисциплины в той последовательности, в какой они представлены.

Планирование времени, необходимого на изучение дисциплин, студентам лучше всего осуществлять весь семестр, предусматривая при этом регулярное повторение материала.

Материал, законспектированный на лекциях, необходимо регулярно прорабатывать и дополнять сведениями из других источников литературы, представленных не только в программе дисциплины, но и в периодических изданиях.

При изучении дисциплины сначала необходимо по каждой теме прочитать рекомендованную литературу и составить краткий конспект основных положений, терминов, сведений, требующих запоминания и яв-

ляющихся основополагающими в этой теме для освоения последующих тем курса. Для расширения знания по дисциплине рекомендуется использовать Интернет-ресурсы; проводить поиски в различных системах и использовать материалы сайтов, рекомендованных преподавателем.

Задачи, стоящие перед студентом при подготовке и написании контрольной работы:

- закрепление полученных ранее теоретических знаний;
- выработка навыков самостоятельной работы;
- выяснение подготовленности студентов к зачету.

Контрольные выполняются студентами в аудитории, под наблюдением преподавателя.

Темы контрольных работ:

1. Особенности процесса и технологии тестирования.

2. Основные понятия и разновидности тестирования.

Вопросы, выносимые на контрольную работу «Особенности процесса и технологии тестирования»:

1. Каковы цели тестирования?

2. Назовите 7 принципов тестирования и расшифруйте их значение.

3. Что такое дефект? Какие существуют виды дефектов (определения)?

4. Перечислите и поясните основные характеристики общих требований к качеству ПО.

5. Опишите ЖЦ дефекта.

6. Опишите схему, по которой должен быть описан дефект.

7. Какова схема действий в процессе тестирования? Опишите каждый этап.

8. Что такое тест-кейсы, для чего пишутся?

9. Что такое чек-лист, для чего пишется?

Вопросы, выносимые на контрольную работу «Основные понятия и разновидности тестирования»:

1. Назовите и опишите уровни тестирования.

2. Перечислите известные вам виды и стратегии тестирования, опишите их (стратегий) основные характеристики.

3. Что такое функциональное тестирование?

4. Охарактеризуйте позитивное негативное и дымовое тестирование.

5. Что оценивает нефункциональное тестирование? Примеры (виды нефункционального тестирования).

6. Что такое регрессионное тестирование?

7. Укажите причины возникновения повторных ошибок.

8. Напишите типичные ошибки при проведении регрессионного тестирования.

9. Перечислите виды регрессионного тестирования.

10. Правила проведения регрессионного тестирования.

11. Что такое автоматизированное тестирование.
12. Укажите минусы и плюсы автоматизации.
13. Каковы цели автоматизации?
14. Каким проектам противопоказана автоматизация?
15. Порядок действий при проведении автоматизации.
16. Какие тесты – лучшие претенденты на автоматизацию.
17. Как выбирать инструменты на автоматизацию.

3.3 Подготовка к лабораторным работам

Проведение лабораторных работ включает в себя следующие этапы:

- постановку темы занятий и определение задач лабораторной работы;
- определение порядка лабораторной работы или отдельных ее этапов;
- непосредственное выполнение лабораторной работы студентами и контроль за ходом занятий;
- подведение итогов лабораторной работы и формулирование основных выводов;
- оформление отчета и защиты лабораторной работы (демонстрация работы и ответы на вопросы по теме лабораторной работы).

При подготовке к лабораторным занятиям необходимо заранее изучить методические рекомендации по его проведению. Обратить внимание на цель занятия, на основные вопросы для подготовки к занятию, на содержание темы занятия.

Если в процессе лабораторной работы или над изучением теоретического материала у студента возникают вопросы, разрешить которые самостоятельно не удается, необходимо обратиться к преподавателю для получения у него разъяснений или указаний.

3.4 Самостоятельное изучение тем теоретической части курса

Темы, отводимые на самостоятельное изучение:

1. Исследовательское тестирование.

2. Гибкое тестирование.

3. Разработка через тестирование.

4. Системы учета дефектов (bug tracking system, BTS).

Рекомендуемая литература:

1. Кент Бек. Экстремальное программирование: разработка через тестирование : пер. с англ. / К. Бек ; пер. П. Анджан. – СПб. : Питер, 2003. – 224 с.

2. Криспин Л., Грегори Д. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд. – М. : – Вильямс, 2010. – 464 с.

3. Сайт «Про Тестинг». – Режим доступа: http://www.protesting.ru/ (дата обращения: 24.04.2018).

4. Проект Software-Testing.ru. – Режим доступа: http://software-testing.ru/ (дата обращения: 24.04.2018).

3.4.1 Исследовательское тестирование

Исследовательское тестирование (exploratory testing) – это одновременное изучение программного продукта, проектирование тестов и их исполнение.

Главное, что нужно помнить об исследовательском тестировании, это то, что само по себе оно не является методикой тестирования. Это, скорее, подход или образ мыслей, который можно применить к любой методике тестирования.

Перечень вопросов, подлежащих изучению

1 Когда следует применять исследовательское тестирование?

2. Что такое тест-туры?

3. Применение чек-листов при исследовательском тестирование.

4. В каких случаях исследовательское тестирование не походит?

3.4.2 Гибкое тестирование

Тестирование является ключевым компонентом *гибкой модели* разработки. Широкое внедрение гибких методов привело к необходимости помещения в центр внимания приемов эффективного тестирования, а

гибкие проекты существенно трансформировали роль тестировщиков ПО.

Перечень вопросов, подлежащих изучению

1. Как вовлечены тестировщики в процесс гибкой разработки ПО?

2. Какое место в гибкой команде занимают тестировщики и менеджеры по контролю качества?

3. Как совершить переход от традиционной циклической к гибкой разработке?

4. Как обеспечить полное выполнение всех действий по тестированию в течение коротких итераций?

5. Как использовать тесты для успешного управления процессом разработки?

3.4.3 Разработка через тестирование

Один из подходов функционального тестирования – составить автоматизированные тестовые сценарии до кодирования. Это подход называется разработкой через тестирование (*test-driven development*, TDD). Разработка через тестирование была тесно связана с концепцией «сначала тест» (*test-first*), применяемой в экстремальном программировании, однако позже выделилась как независимая методология.

Перечень вопросов, подлежащих изучению

1. Опишите стиль разработки.

2. Что такое рефакторинг?

3. Цикл разработки через тестирование.

4. Укажите недостатки и преимущества разработки через тестирование.

3.4.4 Системы учета дефектов

Все дефекты, найденные в системе, должны быть зафиксированы в специальной *Bug Tracking System (BTS)* – системе учета дефектов. Эти системы помогают разработчикам программного обеспечения учитывать и контролировать ошибки, найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий. На рынке ПО предложено огромное количество *bugtracker*, среди них есть и свободно распространяемые и платные.

Перечень вопросов, подлежащих изучению

1. Назначение систем учета дефектов.

2. Redmine.
3. Mantis.
4. Jira.
5. Что должен содержать bug report (отчет об инциденте) в BTS?
6. Какие статусы могут быть присвоены дефекту в BTS?
7. Назначение цветовой раскраски списка задач.

4 Рекомендуемые источники

1. Котляров, В.П. Основы тестирования программного обеспечения [Электронный ресурс] : учеб. пособие. – М. : , 2016. – 248 с. – Режим доступа: https://e.lanbook.com/book/100352 (дата обращения: 24.04.2018).

2. Стандартизация, сертификация и управление качеством программного обеспечения [Электронный ресурс] : учеб. пособие / Т.Н. Ананьева, Н.Г. Новикова, Г.Н. Исаев. – М.:НИЦ ИНФРА-М, 2016. - 232 с. – Режим доступа: http://znanium.com/bookread2.php?book=541003 (дата обращения: 24.04.2018).

3. Казарин О.В. Надежность и безопасность программного обеспечения [Электронный ресурс] : учеб. пособие / О.В. Казарин, И.Б. Шубинский. – М.: Издательство Юрайт, 2018. - 342 с. – Режим доступа: https://biblio-online.ru/book/6A637EC7-8B78-4DA6-B404-71DE0202E2EF (дата обращения: 24.04.2018).

4. Майерс Г. Искусство тестирования программ : Пер. с англ. / Гленфорд Дж. Майерс; Ред. пер. Б. А. Позин. – М. : Финансы и статистика, 1982. – 176 с.

5. Липаев В.В. Тестирование компонентов и комплексов программ [Текст] : учебник для вузов / В. В. Липаев ; Российская академия наук, Институт системного программирования. – М. : Синтег, 2010. – 399 с..

6. Калбертсон Р., Браун К., Кобб Г. Быстрое тестирование. – М. : – Вильямс, 2002. – 384 с.

7. Бейзер Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем. – СПб. : Питер, 2004. – 320 с.

8. Винниченко И.В. Автоматизация процессов тестирования : производственно-практическое издание / И. В. Винниченко. – СПб. : Питер, 2005. – 202[6] с.

9. Бахтизин В.В. Автоматизация тестирования программного обеспечения. : учебн.-метод. пособие / В.В. Бахтизин, С.С. Куликов, Е.П. Фадеева. – Минск: БГУИР, 2012. – 72 с.

10. Морозова Ю.В. Тестирование ПО для студентов направлений «Программной инженерии» и «Бизнес-информатика» [Электронный ресурс]. – Режим доступа: https://sdo.tusur.ru/course/view.php?id=41 (дата обращения: 24.04.2018).

Приложение А

Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

НАЗВАНИЕ ЛАБРАБОТЫ

Отчет по лабораторной работе по дисциплине
«Тестирование программного обеспечения»

Студент гр. ___
_____ И. О. Фамилия
«___»_____201_ г.

Руководитель
доцент каф. АОИ,
канд. техн. наук
_____ Ю. В. Морозова
«___»_____201_ г.

Томск 201_