Министерство науки и высшего образования Российской Федерации

Томский государственный университет
систем управления и радиоэлектроники

Ю. В. Морозова

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБСЕПЕЧЕНИЯ

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

Томск
2022

УДК 004.415.53(075.8)
ББК 32.973.26-018.2я73
М 801

Рецензент:
Пермякова Н.В., доцент кафедры автоматизации обработки информации
Томского государственного университета
систем управления и радиоэлектроники, канд. техн. наук

Морозова Юлия Викторовна
М-801 Тестирование программного обеспечения: методические указания к лабораторным работам и организации самостоятельной работы для студентов направления «Программная инженерия» (уровень бакалавриата) / Ю. В . Морозова. – Томск : Томск. гос. ун-т систем упр. и радиоэлектроники, 2022. – 38 с.

В настоящее время программное обеспечение (ПО) используется во всех сферах человеческой деятельности, и сегодня тестирование – это обязательная часть процесса разработки программного продукта. Цель тестирования ПО – создавать качественный продукт, предотвратить дефекты и, следовательно, обеспечить высокое качество процесса разработки и его результатов. Для этого необходимо обладать: знаниями о видах тестирования; об инструментах и библиотеках для автоматизации тестирования, умением пользоваться специальным ПО для автоматизированного тестирования и регистрации ошибок, навыками тест-дизайна.

Для студентов высших учебных заведений, обучающихся по направлению «Программная инженерия».

Одобрено на заседании кафедры АОИ, протокол № 1 от 20.01.2022

УДК 004.415.53(075.8)
ББК 32.973.26-018.2я73

Оглавление

ВВЕДЕНИЕ

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

Цель дисциплины – познакомить с основными понятиями, принципами и законами тестирования и контроля качества программного обеспечения, необходимыми для работы с современными методологиями тестирования. А также закрепить полученные знания на практике.

В результате изучения дисциплины студент должен:

- знать: основные понятия тестирования; уровни и классификацию видов тестирования; техники тест-дизайна на разных этапах разработки программного продукта;
- уметь: разрабатывать тестовую документацию; выполнять тестирование для разных видов тестирования;
- владеть: основными методиками тестирования программного обеспечения; одним либо несколькими прикладными программами для тестирования программного обеспечения.

1 МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРОВЕДЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

1.1 Лабораторная работа «MindMap»

Цель работы

Получение практических навыков по созданию интеллект-карты (Mind-Map) на проект и применение ее в тестировании ПО.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности

Защита отчета 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.

Предлагаю разделять функции по видам сущности, и по действиям которые с ними можно произвести.

Порядок выполнения работы

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

2. Исследуйте проект и нарисуйте его карту.

В задание достаточно описать основной функционал.

Определите, какие есть функции и/или части приложения.

Предлагаю разделять функции по видам сущности, и по действиям которые с ними можно произвести.

3. Напишите отчет с пояснениями о проекте: назначение проекта, что за проект.

Например: Объектом тестирования был выбран сайт для изучения английского языка LinguaLeo (lingualeo.com)

Lingualeo.com – онлайн-сервис для изучения и практики английского языка, которым пользуются 17 миллионов человек. Главной особенностью данного сервиса является поддержание высокой мотивации. Пользователи изучают английский по материалам на английском языке (фильмы, музыка, книги), проходят тренировки для закрепления словарного запаса, осваивают курсы под свои цели (для работы, общения, путешествий), выполняют различные задания. Lingualeo является мультиплатформенным. Он доступен на веб-платформе и в виде бесплатных мобильных приложений для iOS, Android, Windows Phone и расширений для браузеров.

Варианты заданий

Можно выбрать объект тестирования из предложенных вариантов:

1 https://2droida.ru/
2 https://www.sportmaster.ru/
3 https://www.ozon.ru/
4 https://www.dns-shop.ru/
5 https://www.wildberries.ru/
6 https://www.decathlon.ru/
7 http://knigi.tomsk.ru/
8 https://postelka.ru/tomsk/
9 https://stroypark.su/
10 https://www.citilink.ru/
11 https://0323.ru/
12 https://tomsk.e2e4online.ru/
13 https://pzd-online.ru/

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.

1.2 Лабораторная работа «Локализация дефектов»

Цель работы

Получение практических навыков написания отчетов об инцидентах.

Форма проведения

Выполнение индивидуального задания.

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, про-блема, приведшая к временному падению сервера или приводящая в не-рабочее 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 нали-чие 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 два дефекта в объекте тестирования.
2. Оформить по предложенной структуре.

Контрольные вопросы

1. Что такое дефект?
2. Какие виды дефектов?
3. Что такое локализация дефекта?
4. Что указывает серьезность дефекта?

1.3 Лабораторная работа «Тестовая комбинаторика»

Цель работы

Составление набора входных данных для тестирования.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности.

Отчет с таблицей данных.

Теоретические основы

Чтобы облегчить жизнь тестировщику, были разработаны различные техники и методы тест-дизайна, которые позволяют приблизиться 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 понимания, 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.

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 методики, 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.

Согласно данным сайта pairwise.org, есть множество программ для получения пар, есть и онлайновые сервисы: hexawise, testcover.

Но pict, allpairs и онлайн при одних входных данных могут выдать разные результаты. pict и allpairs почти одно и тоже показывают, а вот онлайн порой сильно отличается. Чем больше значений, тем больше разница. Сложно сказать, какой инструмент работает более точно. Это значит, что в данном примере PICT способен сгенерировать меньшее количество сценариев, чем Allpairs, при одинаковом покрытии. Можно использовать онлайн инструмент,: https://pairwise.teremokgames.com/

Порядок выполнения работы

1. Найдите в своем объекте тестирования место, где может быть применим pairwise (много переменных, мало значений в каждой). Отлично подойдет фильтр параметров выбора товаров.

2. Опишите отдельные параметры.

Для каждого параметра выполните разбиение на классы эквивалентности, определите граничные и особые значения.

3. Составьте входную таблицу данных.

4. Используйте онлайн сервисы для генерации данных для попарного тестирования. Получите выходную таблицу данных.

5. Проанализируйте результат.

6. Заполните 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.

1.4 Лабораторная работа «Тестирование черного ящика»

Цель работы

Получение практических навыков тест-дизайна.

Форма проведения

Выполнение индивидуального задания.

Форма отчетности.

Отчет с тест-кейсами и таблицей решением для формы авторизации.

Теоретические основы

Таблица решений (Decision Table) – техника, помогающая наглядно изобразить комбинаторику условий из требований.

Decision Table всегда состоит из «Условий» («Conditions») и «Действий» («Actions»), информация о которых берется из требований. Как составлять таблицу (рисунок 1.1):

- По горизонтали записываем условия, которые влияют на результат. А чуть ниже – сам результат, в оригинале Action – действие, которое нужно выполнить.
- По вертикали – правила: конкретная комбинация входных условий.

То есть интерпретируем условия как входы, а действия как выходы, т.е ожидаемый результат. Таким образом, колонки – это и будут тест-кейсы!

	Правило 1	**Правило 2**	**...**	**Правило N**
Условия				
Условие 1				
Условие 1				
...				
Условие N				
Действия				
Действие 1				
Действие 2				
...				
Действие N				

Рисунок 1.1 – Таблица решений

Плюсы подхода

1. Наглядность – таблица нагляднее текста. Можно взять таблицу и подойти к аналитику с каким-то вопросом. Или к разработчику. Им будет проще понять, о чём речь, чем если вы принесете стену текста.

2. Нарисовал таблицу = записал тест-кейсы. Поменял в заголовках слово «правило» на «тест-кейс», и вот они, готовые тесты! И это будут основные позитивные тесты, которые мы проводим в первую очередь.

Такой подход подойдет для тестирования формы, которая отправляется для обработки на сервер: Регистрация, Авторизация, Заявка на покупку, Работа фильтра какого-либо списка с большим кол-вом параметров.

Порядок выполнения работы

1. Разработайте таблицу решений для формы авторизации в своем объекте тестирования.
2. Оптимизируйте таблицу решений и определите количество тестов, необходимых для достижения минимальных критериев покрытия.
3. Запишите по полученной таблице тест-кейсы.
4. Заполните отчет с полной и оптимизированной таблицами.

Контрольные вопросы

1. Что такое тест-дизайн?
2. Назовите основные методы проектирования тестов.
3. Какие существуют виды и уровни тестирования?
4. Какое тестирование проводят после внесения исправлений, чтобы убедиться, что проблема действительно решена?

5. В чем принципиальная разница между тестированием черного ящика и белого ящика?

6. Назовите техники метода черного ящика.

1.5 Лабораторная работа «Модульное тестирование»

Цель работы

Овладение практических навыков выполнять модульное тестирование.

Форма проведения

Выполнение индивидуального задания.

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. С 2006 г. использовалась версия JUnit 4, спустя почти 10 лет, в июле 2016 г. ее сменила новая версия – JUnit 5.

Одно из наиболее значимых изменений JUnit 5 – это то, что теперь фреймворк состоит с нескольких компонентов:

1. JUnit Platform – основа платформы, которая позволяет запускать разные тест-фреймворки на JVM.
2. JUnit Jupiter – сердце платформы. Предоставляет новые возможности для создания тестов и разработки собственных расширений.
3. JUnite Vintage, который предоставляет поддержку старых тестов, – это тестовый фреймворк, на котором можно запускать все тесты JUnit 3 и JUnit 4.

Дополнительные функции:

- JUnit больше не требует, чтобы методы были публичными.
- Появился метод для работы с Iterable: assertIterableEquals();
- Новые аннотации. @BeforeAll, @BeforeEach, @AfterAll, @AfterEach.
- Аннотация @Nested позволяет использовать внутренние классы при разработке тестов
- Аннотация @RepeatedTest для повторного запуска теста несколько раз.
- Параметризированные тесты позволяют запускать тест несколько раз с различными входными данными.
- Появилась поддержка default-методов в интерфейсах.

Модуль JUnit входит во многие популярные IDE, поэтому достаточно просто подключить библиотеку к проекту (рисунок 1.2).

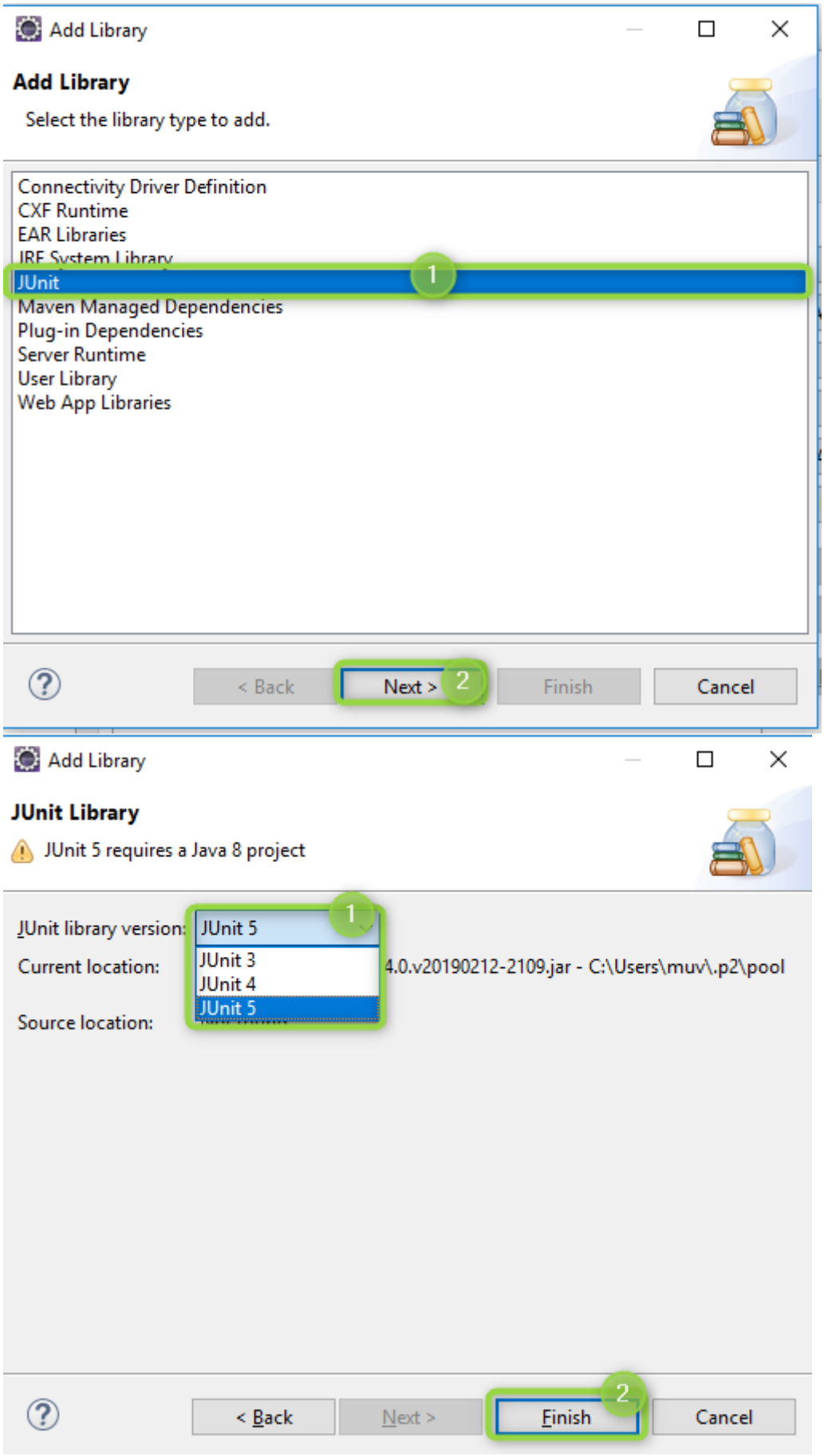


Рисунок 1.2 – Подключение системной библиотеки JUnit5

Теперь можно создать тест, но даже для самого простого теста необходимо указать соответствующую аннотацию @Test (табл.1.1).

Таблица 1.1 – Основные аннотации JUnit

Аннотация	Описание
@BeforeEach	Аннотированный метод будет запускаться перед каждым тестовым методом (аннотированным @Test) в текущем классе (JUnit 5)
@AfterEach	Аннотированный метод будет запускаться после каждого тестового метода (с аннотацией @Test) в текущем классе (JUnit 5)
@BeforeAll	Аннотированный метод будет запущен перед всеми тестовыми методами в текущем классе (JUnit 5)
@ AfterAll	Аннотированный метод будет запущен после всех тестовых методов в текущем классе (JUnit 5)
@Test	Аннотация @Test определяет, что метод или класс является тестовым
@DisplayName	Определяет отображаемое имя класса теста или метода теста (JUnit 5)
@Disable	Используется для отключения или игнорирования тестового класса или метода (JUnit 5)
@Nested	Используется для объявления вложенных тестовых классов (JUnit 5)
@Tag	Объявление тега для обнаружения и фильтрации тестов (JUnit 5)
@TestFactory	Обозначает, что метод – это фабрика тестов для динамических тестов (JUnit 5)
@Ignore	Аннотация заставляет инфраструктуру тестирования проигнорировать данный тестовый метод (JUnit 4)
@Rule	Аннотация позволяет более гибко работать с классами-утилитами, определяющими правила работы с тестами
@RunWith	Аннотация @RunWith задает способ запуска теста. В случае использования @RunWith (Suite.class) осуществляется запуск набора тестов, перечисляемых в аннотации @Suite.SuiteClasses
@Parameters	Аннотация @Parameters маркирует статический метод, который возвращает данные для теста, представленные типом Collection для параметризованного теста
@TestedOn	Аннотация @TestedOn принимает массив значений, которые будут использоваться в качестве точек данных для аннотированных (JUnit 5)
@Category	Используется для группировки тестов по категориям
@RepeatedTest	Повторные тесты (JUnit 5)

Тесты не были бы тестами без проверок. Верификация полученного фактического результата, проверка совпадения с ожидаемым результатом – это один из ключевых моментов тестирования. Проверки чаще всего выполняются с помощью класса *Assert*, хотя иногда используют ключевое слово *assert*.

Давайте рассмотрим пример.

Программа, которая рассчитывает зарплату для сотрудников:

```
public class Salary
{
        private double salary;
        private String name;

        public Salary(String name, double salary) {
                this.name = name;
                this.salary = salary;
                }

        public Salary() {
                this.name = "";
                this.salary = 0;
                }

        public double getSalary() {
          return salary;
        }

        public String getName() {
                return name;
              }

        public void setName(String name) {
                if (name == null) {
                  throw new IllegalArgumentException("Negative name is invalid.");
                }
                this.name = name;
              }

        public void setSalary(double salary) {
          if (salary < 0) {
            throw new IllegalArgumentException("Negative salary is invalid.");
          }
          this.salary = salary;
        }

        public double getGrossSalary() {
          return this.salary + getSocialInsurance() + getAdditionalBonus();
        }

        public double getSocialInsurance() {
          return this.salary * 25 / 100;
        }

        public double getAdditionalBonus() {
          return this.salary / 10;
        }
```

```
                @Override
                public String toString() {
                   return "{" +
                                "name=" + name +
                         ", salary=" + salary +
                         '}';
                }

   public static void main( String[] args )
   {
         System.out.println(new Salary("Иванов", 15000));
   }
}
```

Напишем несколько тестов для основных методов:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class SalaryTest {
         private Salary app = new Salary();
         @Test
         void test_assert() {
                double salary = 4000;
                app.setSalary(salary);
          double expectedSocialInsurance = salary * 0.5;
          assert expectedSocialInsurance==app.getSocialInsurance();
         }
         @Test
         void testSocialSalaryWithValidSalary() {
                double salary = 4000;
                app.setSalary(salary);
          double expectedSocialInsurance = salary * 0.5;
          assertEquals(expectedSocialInsurance, app.getSocialInsurance());

         }
         @Test
         void testAdditionalSalaryWithValidSalary() {
          double salary = 4000;
          app.setSalary(salary);

          double expectedAddionalBonus = salary * 0.1;
          assertEquals(expectedAddionalBonus, app.getAdditionalBonus());
         }

         @Test
         void testSalaryWithValidSalary() {
          double salary = 4000;
          app.setSalary(salary);

          double expectedAddionalBonus = salary * 0.1;
          double expectedSocialInsurance = salary * 0.25;
```

```
        double expectedGross = salary + expectedSocialInsurance + expectedAddionalBonus;
        assertEquals(expectedGross, app.getGrossSalary());
    }

}
```

Запустив JUnit и просмотрев отчет о тестировании online_tusur.unit_online_tusur.SalaryTest.txt, мы увидим, что в тесте test_assert() с ключевым слово assert нет диагностики, нет подробностей того, что случилось, а вот в тесте с тестовым методом класса Assert уже показано: *org.opentest4j.AssertionFailedError: expected: <2000.0> but was: <1000.0>*, т. е. ожидаемое 2000, фактическое – 1000 (рис. 1.3).

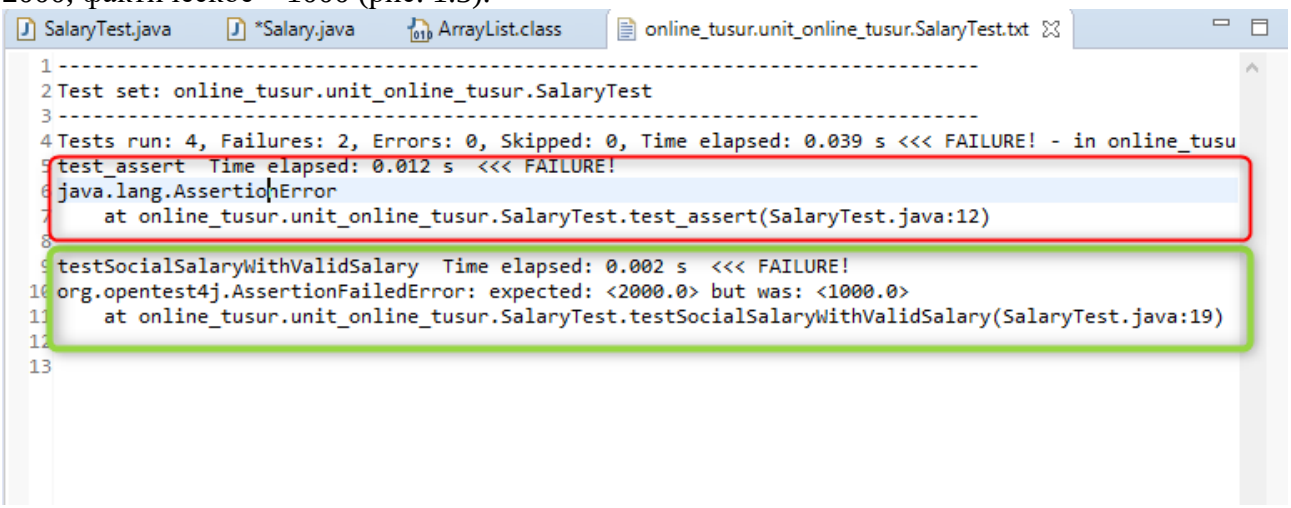


Рисунок 1.3 – Отчет JUnit5

Поэтому рекомендуется использовать тестовые методы класса Assert, тем более существует больше их количество на все случаи (табл. 1.2).

Таблица 1.2 – Тестовые методы класса Assert

Метод	Описание
fail(String)	Указывает на то, что тестовый метод завалился, при этом выводя текстовое сообщение
assertTrue(boolean condition, message)	Проверяет, что логическое условие истинно
assertsEquals(expected, actual, message)	Проверяет, что два значения совпадают
assertNull(object, message)	Проверяет, что объект является пустым **null**
assertNotNull(object, message)	Проверяет, что объект не является пустым **null**
assertSame(expected, actual, message)	Проверяет, что обе переменные относятся к одному объекту
assertNotSame(expected, actual, message)	Проверяет, что обе переменные относятся к разным объектам
assertNotEquals(expected, actual, message)	Проверяет, что два объекта не равны
assertFalse(boolean condition, message)	Проверяет, что логическое условие «Ложь»

assertArrayEquals([] expected, [] actual)	Проверяет, что expected и actual массивы равны
assertAll(String heading, <Executable> executables)	Проверяет, что все поставленные executables не генерируют исключения
assertThrows(Class expectedType, Executable executable)	Проверяет, что ожидаемое исключение совпадает с возвращаемым фактическим сгенерированным исключением

Написать автотесты – это еще полдела, необходимо проверить, а весь ли код покрыт тестами. Тесты должны покрывать 100% функционала. Если первый пункт является организационной задачей, то второй – технической. И под «100%» нужно понимать не наличие теста на каждое требование, а то, что каждая строчка кода будет исполнена в результате исполнения хотя бы одного теста. Данная характеристика называется code coverage и буквально означает степень покрытия кода тестами.

Различаются следующие показатели:

- покрытие функций (Function Coverage) – подсчет по вызовам методов;
- покрытие решений (Decision Coverage) – подсчет по возможным направлениям исполнения кода (then-else или case-case-default в управляющих структурах). Учитывает единственное ветвление в каждом конкретном случае;
- покрытие состояний (Statement Coverage) – подсчет по конкретным строчкам кода;
- покрытие путей (Path Coverage) – подсчет по возможным путям исполнения кода. Более широкое понятие, чем decision coverage, так как учитывает результат всех ветвлений;
- покрытие условий (Conditional Coverage) – подсчет по возможным результатам вычисления значений булевских выражений и подвыражений в коде.

Современные IDE поддерживают покрытие кода. Существует ряд расширений и плагинов для разных языков программирования:

- Java: Jcov, EclEmma – Java Code Coverage for Eclipse, eCobertura;
- C++ : Gcov, Tcov;
- C# : OpenCover;
- встроенный в Visual Studio Test Coverage.

Мы рассмотрим *EclEmma – Java Code Coverage for Eclipse.*

EclEmma – бесплатный инструмент покрытия кода на Java для среды Eclipse, доступный по лицензии Eclipse Public License. Он делает доступным анализ покрытия кода прямо в Eclipse.

Возможности EclEmma:

- Быстрый цикл разработки/тестирования. Запуски тестов типа JUnit в Eclipse могут быть проанализированы напрямую на предмет наличия покрытия кода.
- Богатый анализ покрытия. Результаты покрытия мгновенно приводятся и выделяются в окне редактора кода на Java.
- Бесконтактный. EclEmma не требует модификации проектов или выполнения всяких установок.

Разработка EclEmma основана на библиотеке EMMA, разработанной Владом Рубцовым, поэтому и получила свое название от Eclipse Emma. Если коротко, то EMMA – это набор инструментов с открытым исходным кодом для измерения и отчетов по покрытию кода на Java.

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.

Анализ производится по требованию или после того, как приложение было выполнено и завершено. В среде Eclipse автоматически появляется информация по покрытию кода:

- списки Coverage view, содержащие информацию по покрытию кода Java-проекта, позволяющие по клику перейти непосредственно в код;
- выделение кода цветом: 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;
- разные счетчики: можно выделить, будет ли EclEmma анализировать инструкции, ветвления, просто строки, методы, типы или циклы;
- несколько проходов оценки покрытия: возможно переключение между данными о покрытии из нескольких сессий покрытия.

Также EclEmma позволяет экспортировать *отчеты о покрытии*: данные о покрытии могут быть экспортированы в формате html, xml или csv в качестве исполняемых файлов с данными JaCoCo (*.exec).

Для EclEmma необходимы Eclipse 3.5 (или выше) и Java 1.5 (или выше).
Новые версии Eclipse уже имеют встроенную EclEmma (рис. 1.4).

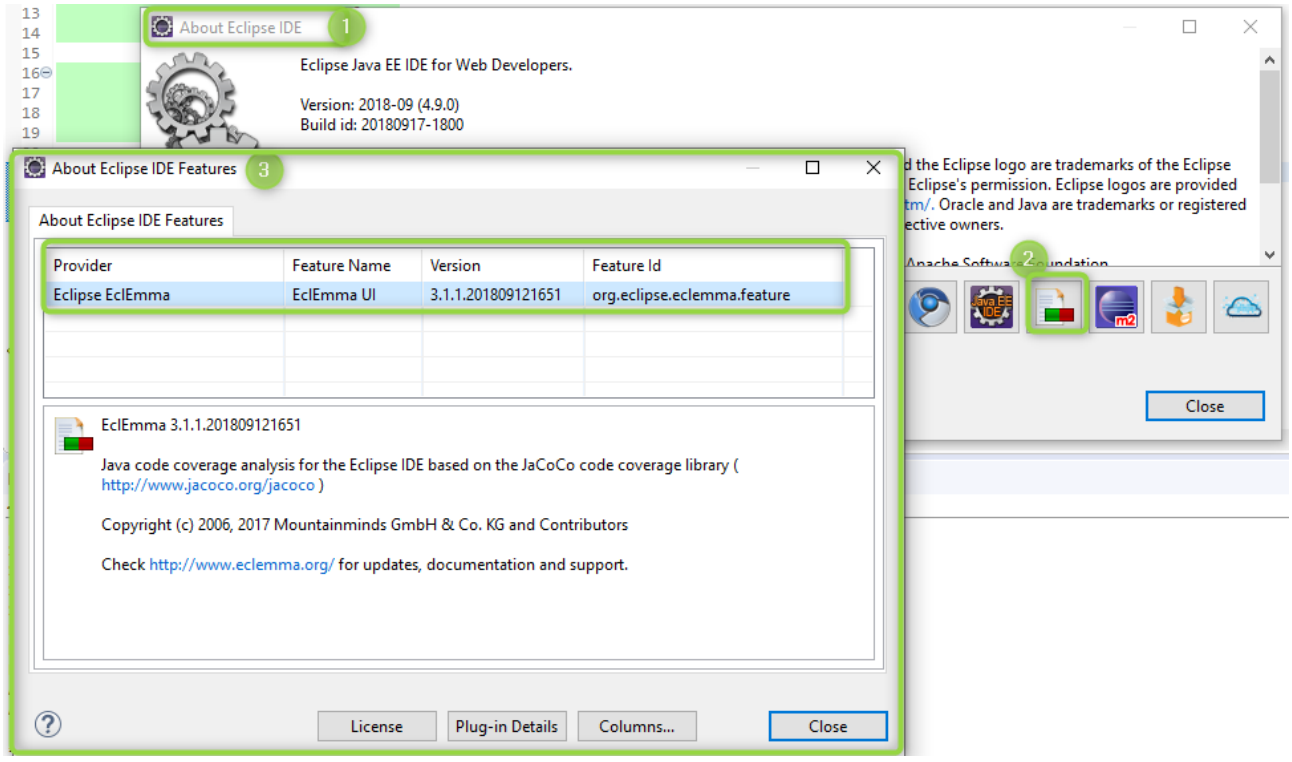


Рисунок 1.4 – Установка EclEmma

Метрики покрытия различных свойств кода (таких как строки, ветви и инструкции) конкретным тестом можно получить, выполнив анализ покрытия. Например, чтобы проверить свойства покрытия для контрольного теста NGTest.java, перейдите в Package Explorer, выберите этот файл и щелкните правой кнопкой мыши на Properties, чтоб отобразить вкладку Coverage. Свойства покрытия перечислены на рисунке 5.

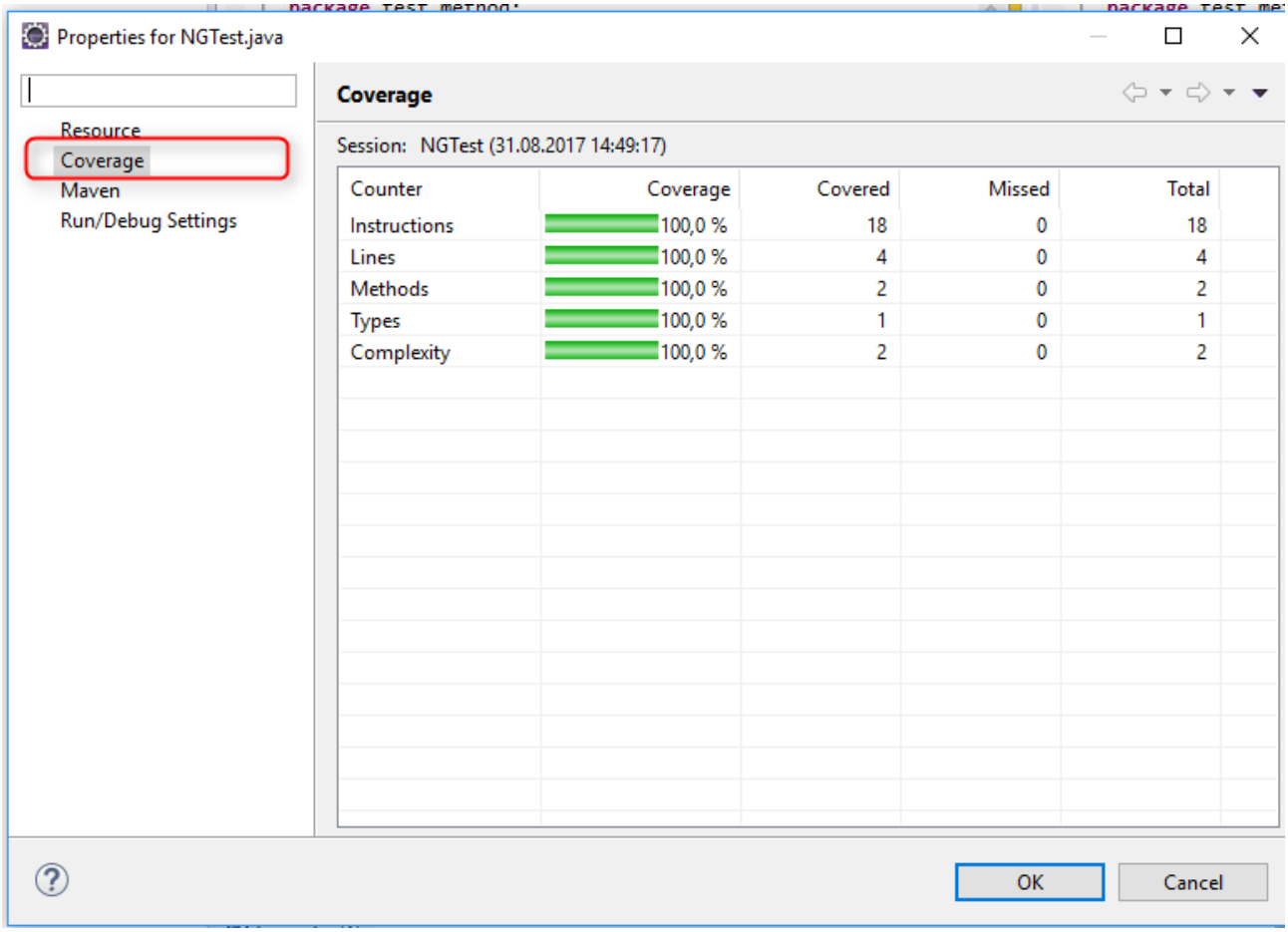


Рисунок 1.5 – Свойства покрытия в EclEmma

Набор покрытия создается каждый раз, когда выполняется тест или приложение с измерением покрытия кода. Каждый цвет в исходном коде имеет конкретное значение при определении покрытия для теста:

- зеленый: строки кода с полным покрытием;
- желтый: строки кода с частичным покрытием (некоторые инструкции или ветви пропущены);
- красный: строки с отсутствующим покрытием. Эти строки кода еще не были выполнены.

Кроме того, цветные бриллианты показаны слева для строк, содержащих ветви условий. Цвета бриллиантов имеют идентичную семантику с цветами линий:

- зеленый для полностью покрытых ветвей;
- желтый для частично покрытых ветвей;
- красный, если не было выполнено ни одной ветки в конкретной строке.

Эти цвета по умолчанию могут быть изменены в диалоге Window → Preferences → Editors → Text Editors → Annotations, где Annotations позволяет изменять визуальное представление подсветки покрытия. Соответствующие записи:

- Полный охват;
- Частичное покрытие;
- Нет покрытия.

Измерим покрытие кода нашей простой программы по расчету бонусов (рис. 1.6):

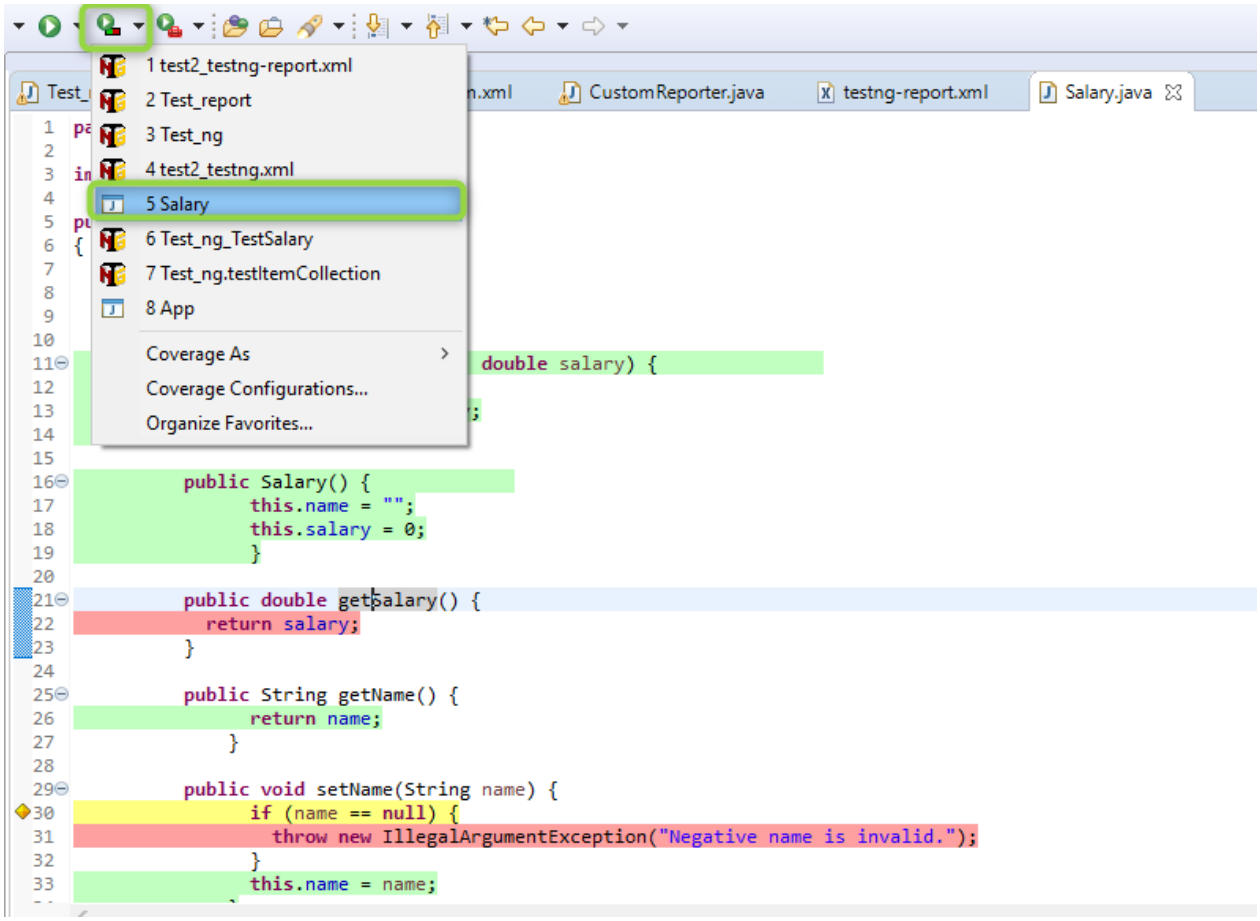
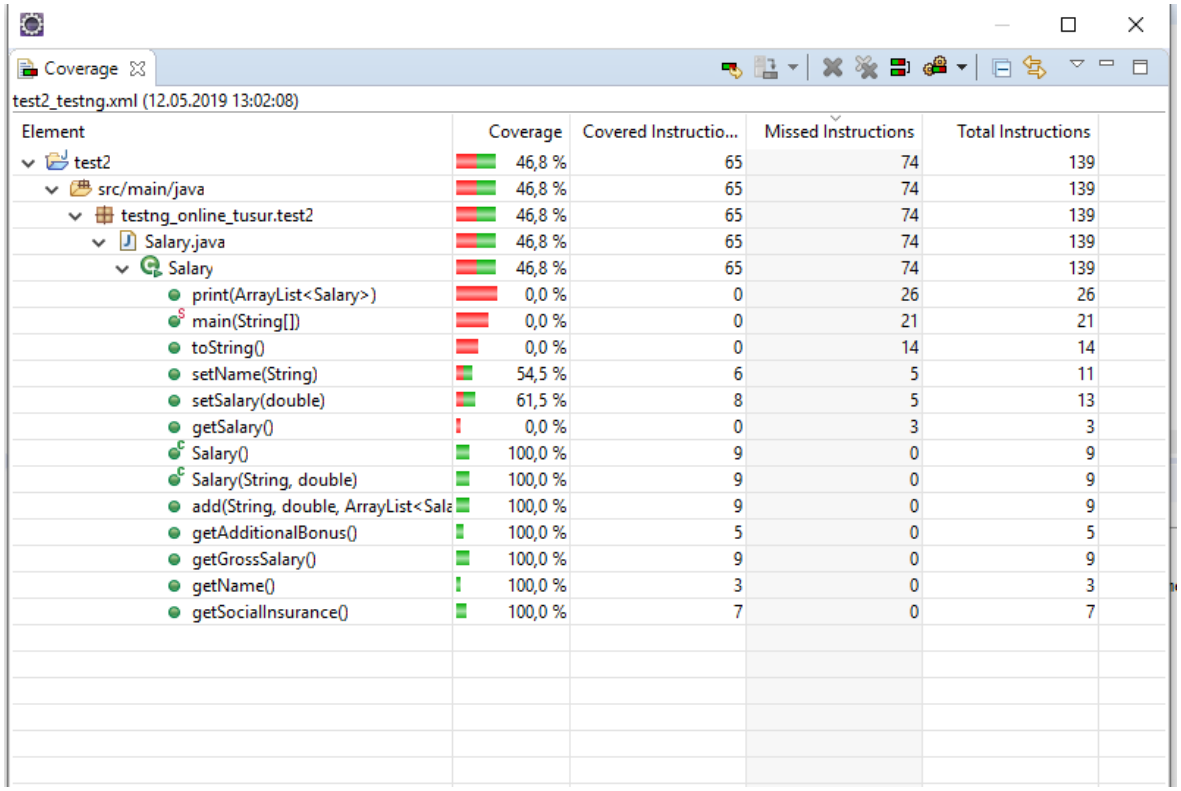


Рисунок 1.6 – Покрытие кода в EclEmma

Как видим, большая часть кода у нас не покрыта тестами (рис. 1.7).



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
test2	46,8 %	65	74	139
src/main/java	46,8 %	65	74	139
testng_online_tusur.test2	46,8 %	65	74	139
Salary.java	46,8 %	65	74	139
Salary	46,8 %	65	74	139
print(ArrayList<Salary>)	0,0 %	0	26	26
main(String[])	0,0 %	0	21	21
toString()	0,0 %	0	14	14
setName(String)	54,5 %	6	5	11
setSalary(double)	61,5 %	8	5	13
getSalary()	0,0 %	0	3	3
Salary()	100,0 %	9	0	9
Salary(String, double)	100,0 %	9	0	9
add(String, double, ArrayList<Sala	100,0 %	9	0	9
getAdditionalBonus()	100,0 %	5	0	5
getGrossSalary()	100,0 %	9	0	9
getName()	100,0 %	3	0	3
getSocialInsurance()	100,0 %	7	0	7

Рисунок 1.7 – Результат покрытия

Рассмотрим панель инструментов и раскрывающееся меню на Coverage:

Панель инструментов представления покрытия предлагает следующие действия:

- Последнее запущенное покрытие: повторно запустить текущий выбранный сеанс покрытия.
- Дамп данных выполнения из запущенного процесса и создание нового сеанса из данных. Активен, только когда хотя бы один процесс запущен в режиме покрытия.
- Удалить активный сеанс: удалить текущий выбранный сеанс покрытия.
- Удалить все сеансы: удалить все сеансы покрытия.
- Объединить сеансы: объединяет несколько сеансов в один.
- Выберите сеанс: выберите сеанс из выпадающего меню и сделайте его активным сеансом.
- Свернуть все: свернуть все развернутые узлы дерева.
- Связать с текущим выбором: если этот переключатель установлен, представление покрытия автоматически показывает элемент Java, выбранный в настоящее время в других представлениях или редакторах.

Некоторые из действий деактивируются, если нет сеанса или существует только один сеанс. Дополнительные настройки доступны из раскрывающегося меню вида покрытия:

- Показать элементы: выберите элементы Java, отображаемые как корневые записи в дереве покрытия: проекты, корни фрагментов пакетов (исходные папки или библиотеки), фрагменты или типы пакетов.
- Режим счетчика. В раскрывающемся меню представления можно выбрать различные режимы счетчика: инструкции байт-кода, ветви, линии, методы, типы и цикломатическая сложность.
- Скрыть неиспользуемые элементы: отфильтруйте все элементы из представления покрытия, которые не были выполнены вообще во время сеанса покрытия.

Исправим ситуацию и добавим в класс SalaryTest новые тесты, чтобы добиться результата 100%. Бывает проблематично тестировать методы, которые ничего не возвращают. Поскольку основной метод статичен и ничего не возвращает, мы должны проверить объект, который изменился после основного вызова. Основной метод вызывает разные методы для разных входов в консоль и вызывает разные функции для разных аргументов. Поэтому тестируем поток PrintStream.

```
@Test
void testMain() {
  String Name = "Сидоров";
  double salary= 10000;
  double expectedAddionalBonus = salary * 0.1;
  double expectedSocialInsurance = salary * 0.25;
  double expectedGross = salary + expectedSocialInsurance + expectedAddionalBonus;
  String expectedFullName =Name+", "+expectedGross;


  ByteArrayOutputStream output = new ByteArrayOutputStream();
  PrintStream old=System.out;
        System.setOut(new PrintStream(output));

       Salary.main(new String[] {});

        assertEquals(expectedFullName, output.toString());
        System.setOut(old);
```

```
}
```
Точно также можно тестировать входящий поток Scanner:

```
@Test
void testMain(String[] args) {
//ожидаемое
String expectedValue ="Введите число:\r\nВведите степень числа:\r\nСтепень числа
2^10=1024";
ByteArrayOutputStream output = new ByteArrayOutputStream();
PrintStream old=System.out;
//подставляем два потока Scanner
ByteArrayInputStream testIn1 = new ByteArrayInputStream(("2\n").getBytes());
ByteArrayInputStream testIn2 = new ByteArrayInputStream(("10\n").getBytes());
//Объединяем эти потоки
SequenceInputStream in = new SequenceInputStream(testIn1,testIn2);
System.setIn(in);
System.setOut(new PrintStream(output));
//вызываем тестируемый метод main()
App.main(new String[] {});
//сравниваем ожидаемое с фактическим
assertEquals(output.toString(),expectedValue);
//возвращаем стандартные потоки
System.setOut(old);
System.setIn(in);
}
```

Хватило 10 тестов, 2 из которых негативные, чтобы добиться 100%-ного покрытия (рис. 1.8).

Мастер экспорта позволяет экспортировать сессии покрытия в одном из следующих форматов:

- html: подробный и просматриваемый отчет в виде набора файлов html;
- xml: данные покрытия в виде одного структурированного xml-файла;
- csv: данные о степени детализации на уровне класса в виде значений, разделенных запятыми;
- файл данных исполнения: собственный формат данных исполнения JaCoCo.

Для использования мастера экспорта должна быть выполнена хотя бы одна сессия покрытия (рис.1. 9).

Выбираем формат экспорта в html. Обновив проект, увидим новую папку jacoco-resourses и отчет:

С помощью плагина EclEmma разработчик и тестировщик могут получать информацию об уровне тестирования конкретного набора кода (или набора тестов в блоке или комплекте тестов). Подробную информацию о покрытии кода можно получить как на уровне проекта, так и на уровне каждого метода. Использование этих инструментов в процессе ежедневной разработки позволяет существенно повысить качество и производительность любого проекта.

Порядок выполнения задания

1. Установите **Eclipse (в пособие используется** Java EE IDE for Web Developers, Version: 2018-09 (4.9.0)**)**
2. Создайте **Java-проект.**
3. Добавьте в проект **JUnit5**.
4. Соберите проект.
5. Напишите код программы согласно варианту, которую будем тестировать.
6. Добавьте в проект в **AppTest.java** простейший класс с тестовым методом:

Запустите проект и убедитесь, что созданный тестовый метод запускается.

7. Разработанные ранее тесты (см. предыдущую лабораторную работу) добавьте в созданный проект.

Не нужно делать сложные тесты, со сложной логикой или сложной архитектурой. Наша цель - изучение возможностей тестового класса Assert и применение тестовых аннотаций. Поэтому делайте несложные тесты.

8. Оцените покрытие проекта тестами.

9. Добейтесь 100% покрытия.

В качестве результата выполнения задания приложите архив с проектом и отчетом последней сессии покрытия: подробный и просматриваемый отчет в виде набора файлов **html, используя мастер экспорта.** Комментарии в коде программы приветствуются!

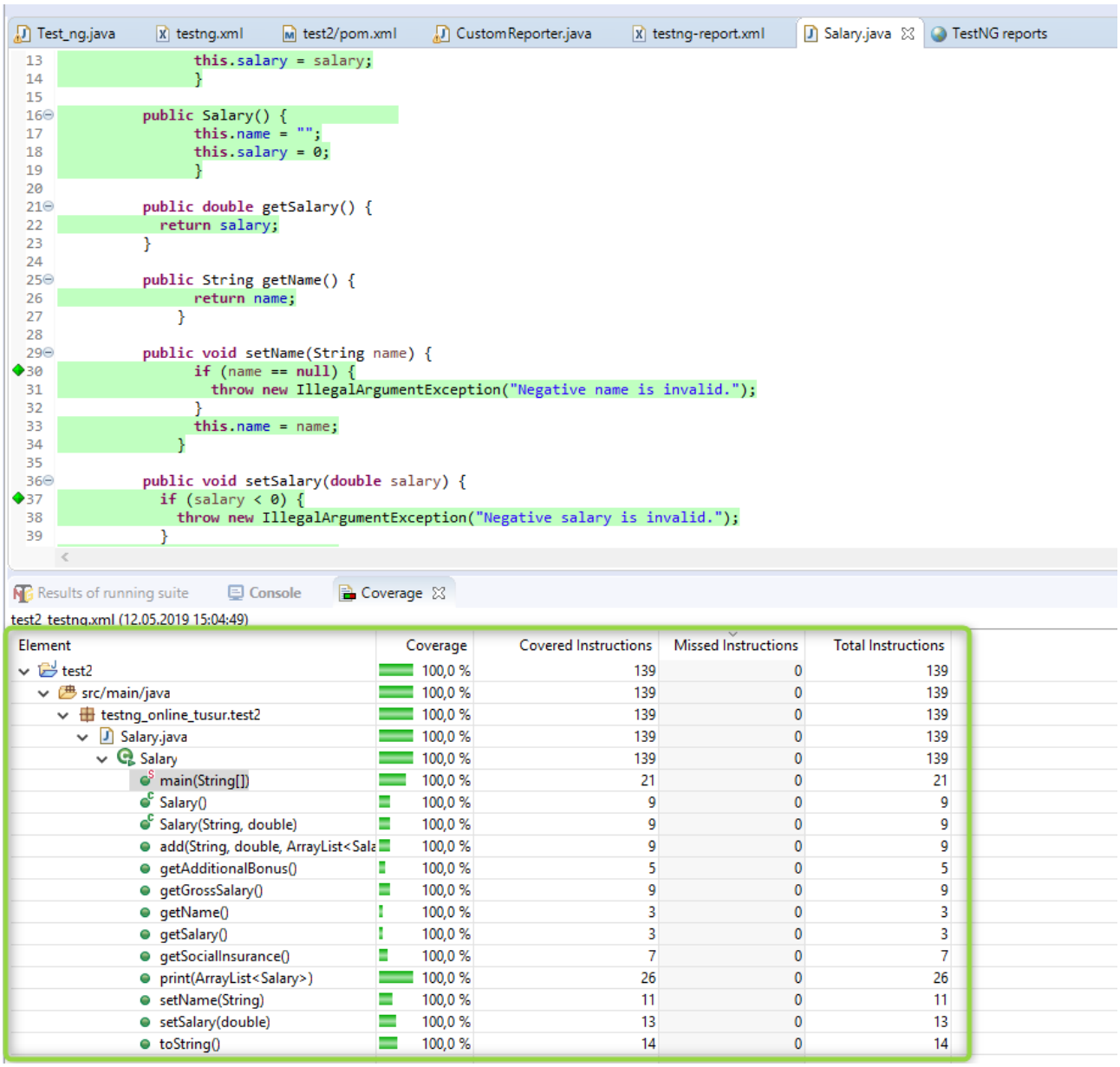


Рисунок 1.8 – 100%-ное покрытие кода

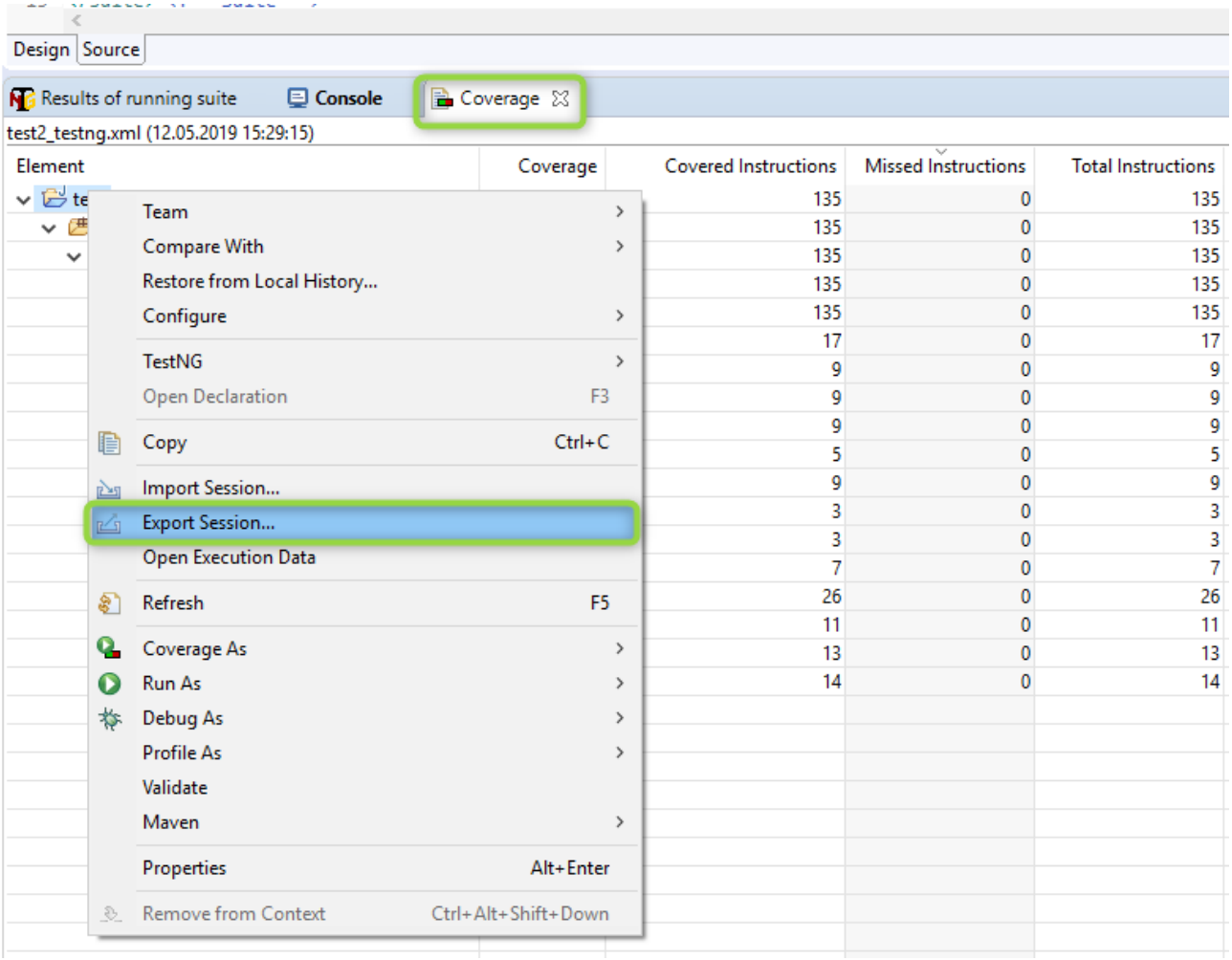


Рисунок 1.9 – Экспорт отчета о покрытии

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. Чем отличается отладка от тестирования?
2. На каком уровне находится модульное тестирование?
3. Перечислите возможности фреймворка JUnit.
4. Укажите назначение аннотации @Test.
5. Какие аннотации позволяют отключать тесты?
6. Что такое покрытие кода?
7. Назовите метрики покрытия кода.

1.6 Лабораторная работа «Нефункциональное тестирование»

Цель работы

Получение практических навыков нефункционального тестирования.

Форма отчетности

По результатам анализа напишите отчет в свободной форме. Приложите скриншоты результатов проверки.

Теоретические основы

Если в рамках функционального тестирования мы отвечаем на вопрос «Работает ли система?», то нефункциональное отвечает на вопрос: «Как хорошо работает система?». Нефункциональное тестирование направлено на проверку тех аспектов ПО, которые могут быть описаны в документации, но не относятся к конкретным функциям.

Нефункциональное тестирование состоит из подвидов:

- Тестирование стабильности приложения – Stability/Reliability testing – обнаружение крэшей системы во время использования.
- Юзабилити тестирование – Usability testing – исследование для определения удобства использования ПО.
- Нагрузочное тестирование – Load testing – как правило, проводится с целью определения поведения ПО под ожидаемым уровнем нагрузки.
- Тестирование производительности – Performance testing – проверка скорости работы ПО или его отдельных функций.
- Тестирование совместимости – Compatibility testing – тестирование системы во время работы в разных окружениях: «железо», софт и т. д.
- Тестирование безопасности – Security testing – проводится для ответа на вопрос «Является ли приложение безопасным/защищенным или нет?».
- Объемное тестирование – Volume testing – тестирование ПО с использованием баз данных определенного размера.

- Стресс-тестирование – Stress testing – это тестирование в ограниченных условиях, например проверка поведения системы (отсутствие крэшей) при условиях нехватки ресурсов ПК (ОЗУ или места на HDD/SSD дисках).
- Тестирование скорости восстановления – Recovery testing – проводится с целью определения скорости восстановления системы в случае софтверного крэша (крэша программного обеспечения) или ошибки «железа».
- Тестирование локализации, интернационализация – Localization testing – проверка ПО на соответствие языковых, культурных и/или религиозных норм. Локализация – проверка отображения всех переведенных в ПО текстов.

Нефункциональное тестирование предложено проводить с помощью инструмента разработчика DevTools Google Chrome. Инструменты разработчика доступны в современных браузерах. Mozilla firefox, в Opera, то есть во всех браузерах также есть подобные утилиты. Почему мы будем с вами изучать данные инструмент? Он достаточно распространен среди разработки, помогает также в тестирование.

DevTools (Tools for Web Developers) — приложение, которое позволяет:

- просматривать HTML-код страницы;
- контролировать процесс исполнения кода;
- просматривать системную информацию (логи).

Инструмент можно вызвать нажатием горячей клавишей или hotkey - F12. Точно также вы можете вызвать данную утилиту у других браузерах. Или через настройки в браузере через меню браузера: в правом верхнем углу три точки -> Дополнительные инструменты -> Инструменты разработчика.

Перед выполнением лабораторной работы рекомендует изучить документацию на официальном сайте Chrome DevTools - Chrome Developers.

Порядок выполнения задания:

1. В форме для авторизации определить локаторы в css и xpath для полей ввода логина и пароля, кнопки Submit.

2. При нажатии на кнопку Submit отправляется запрос для определения типа браузера и ip клиента. Вам нужно определить URL, куда отправляется запрос, и ввести его в поле ввода в качестве ответа.

3. Определить плейсхолдер (placeholder, подсказку внутри поля формы) для поля поиска по сайту.

4. Сменить цвета на странице:
#CDEDFF
Фон
#1F87EF

Верхняя панель (хедер, Header)
#1874CF

Нижняя панель (футер, footer)
Сохраните скриншот страницы размером 1080x780 в масштабе 75% средствами DevTools.

5. Сохраните все цвета, использованные в color, background-color, and border-color, используя следующий сниппет:

```
(function () {
  var allColors = {};
  var props = ["background-color", "color", "border-top-color", "border-right-color", "border-bottom-color", "border-left-color"];
```

```
  var skipColors = { "rgb(0, 0, 0)": 1, "rgba(0, 0, 0, 0)": 1, "rgb(255, 255, 255)": 1 };

  [].forEach.call(document.querySelectorAll("*"), function (node) {
    var nodeColors = {};
    props.forEach(function (prop) {
      var color = window.getComputedStyle(node, null).getPropertyValue(prop);
      if (color && !skipColors[color]) {
        if (!allColors[color]) {
          allColors[color] = {
            count: 0,
            nodes: []
          };
        }
        if (!nodeColors[color]) {
          allColors[color].count++;
          allColors[color].nodes.push(node);
        }
        nodeColors[color] = true;
      }
    });
  });

  var allColorsSorted = [];
  for (var i in allColors) {
    allColorsSorted.push({
      key: i,
      value: allColors[i]
    });
  }
  allColorsSorted = allColorsSorted.sort(function (a, b) {
    return b.value.count - a.value.count;
  });

  var nameStyle = "font-weight:normal;";
  var countStyle = "font-weight:bold;";
  var colorStyle = function (color) {
    return "background:" + color + ";color:" + color + ";border:1px solid #333;";
  };

  console.group("All colors used in elements on the page");
  allColorsSorted.forEach(function (c) {
    console.groupCollapsed("%c    %c " + c.key + " %c(" + c.value.count + " times)",
      colorStyle(c.key), nameStyle, countStyle);
    c.value.nodes.forEach(function (node) {
      console.log(node);
    });
    console.groupEnd();
  });
  console.groupEnd("All colors used in elements on the page");
})();
```

6. Проверить страницу в портретном и ландшафтном режимах на девайсах Galaxy Fold и MotoG4, Nexus 5.

7. Сгенерируйте отчет о производительности Perfomance, сравните результат скорости загрузки с результатом из задания 6.

Вставьте скриншоты в отчет и опишите какие проблемы возникли по каждому пункту.

Контрольные вопросы

1. Зачем тестировщику может пригодиться вкладка Console?
2. В какой вкладке можно посмотреть ответы от сервера?
3. В какой вкладке можно посмотреть время загрузки страницы?
4. Для чего служить вкладка Elements?

1.7 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. 1.10) 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.

Рисунок 1.10 – Панель инструментов Selenium

Главное достоинство Selenium – запись действий пользователя в браузере.

Панель тестового сценария

На панели (рис. 1.11) 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

Рисунок 1.11 – Набор команд

Поля ввода данных «Command», «Target» и «Value» отображают выбранную в данный момент команду, а также ее параметры (рис. 1.12). 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».

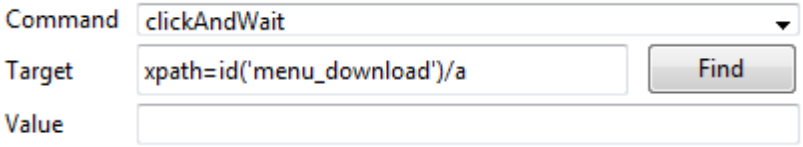


Рисунок 1.12 – Target

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. 1.13).

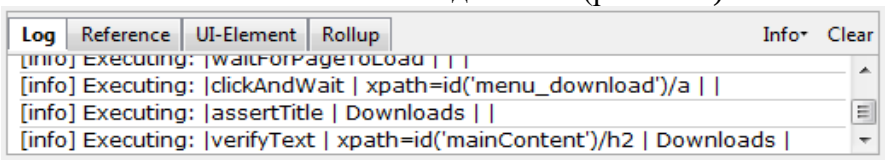


Рисунок 1.13 – Log

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. 1.14).

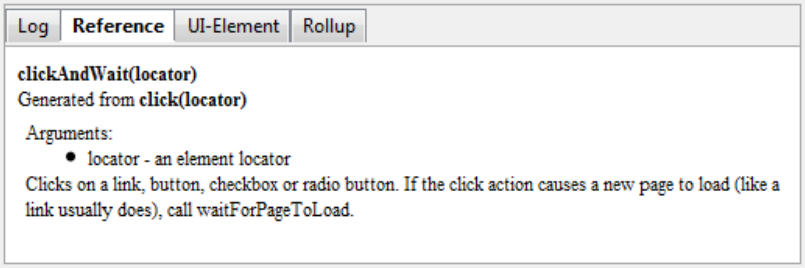


Рисунок 1.14 – Вкладка «Reference»

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.
2. Записать функциональные сценарии.
   - Провести тестирование регистрации
   - Провести тестирование поиска на сайте.
   - Провести тестирование локализации.

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

Контрольные вопросы

1. Что такое автоматизированное тестирование?
2. Какие команды можно выполнять в Selenium?
3. На каком языке можно экспортировать тест-сьют из Selenium?

1.8 Лабораторная работа «Тестирование API»

Цель работы

Получение практических навыков по тестирования клиент-серверных приложений.

Форма отчетности

Отчет должен включать запросы.

Теоретические основы

Многие уже сталкивались таким понятием как интерфейс.

- графическим интерфейсом – GUI (Graphical User Interface);
- интерфейсом командной строки – CLI (Command Line Interface).

Интерфейс – это граница между двумя функциональными системами, на которой происходит их взаимодействие и обмен информацией. Но при этом процессы внутри каждой из систем скрыты друг от друга. С помощью интерфейса можно использовать возможности разных систем, не задумываясь о том, как они обрабатывают наши запросы и что у них внутри. Через них пользователь взаимодействует с приложениями, компьютерами и другими устройствами.

Существует еще и API(Application Programming Interface) . Это тоже интерфейс – программный интерфейс приложения. API – это недооцененные посредники, которые значительно упрощают взаимодействие между приложениями и веб-серверами.

API (Application Programming Interface – программный интерфейс приложения) – это набор способов и правил, по которым различные программы общаются между собой и обмениваются данными. Тестирование API – это уже интеграционное тестирование.

Точно так же с помощью вызовов API можно выполнить определённые функции программы, не зная, как она работает. Поэтому API и называют интерфейсом.

Тестировать API без документации достаточно сложно, на реальных проектах она есть обязательно.

Документация позволяет изучить работу API компонента или приложения. Она описывает:

- Действия, которые можно произвести с системой, и данные, которые можно от неё получить.
- Ограничения передаваемых данных. Например, некоторые параметры могут иметь ограничение на тип данных (число или строка), на длину (от 2 до 50 символов) или на обязательность (нужно ли передавать или можно пропустить).

Структуру запросов, ответов и адреса, по которым запросы можно слать.

Чаще всего документацию пишут разработчики.

Все виды документации по API выглядят примерно одинаково. В любом из видов можно увидеть какой метод нужно использовать, какой URL, какие body, params headers и так далее. Это описание требований API. Задача тестировщика – убедиться, что реализация выполнена согласно требованиям. Чтобы это сделать, нужно провести тест-анализ и тест-дизайн на их основе, а затем протестировать.

Для практики можно воспользоваться тренировочными сайтами, например https://reqres.in/

Не всегда программа предоставляет именно графический интерфейс. Это может быть SOAP, REST интерфейс, или другое API. Чтобы использовать этот интерфейс, вы должны понимать:

- что подать на вход;
- что получается на выходе;
- какие исключения нужно обработать.

Если приложение построено по REST. Значит, оно принимает запросы по HTTP в форматах JSON или XML. Эта информация помогает определиться с инструментами: скорее всего, здесь лучше тестировать через Postman.

Порядок выполнения работы

1. Заведите аккаунт на официальном сайте и создайте рабочее пространство в веб-версии Postman.

2. Изучите документацию в обучающем центре Postman. Там есть небольшие видео с уроками, также примеры запросов. Создайте коллекцию.

Используйте свое API или тестовый сайт https://reqres.in/ , выполните следующие сценарии:

1) получите информацию. Необходимо проверить, что в нем есть пользователь с id=10. Сохраните id, name, email в переменных коллекции.

2) создайте информацию или зарегистрируйте пользователя.

Email: eve.holt@reqres.in

Password: pistol

Проверьте, что в ответе token= "QpwL5tke4Pnpja7X4".

Используйте значение name из переменной коллекции для поля "password".

Используйте значение email из переменной коллекции для поля "email".
Проверьте код статуса, что он равен 200.
3) обновите данные.
Для name: Eva и job: QA
Проверьте, что поле updatedAt не пустое.
4) удалите информацию или пользователя.
Проверить, что Body ответа пустое.
Тут можно проверить двумя способами:

- pm.response.text() - текст ответа пустой.
- тело pm.response пустое ""

3. Экспортируйте коллекцию.

Контрольные вопросы

1. Какой интерфейс применяют при взаимодействии приложений между собой?
2. Что такое код состояния в запросе?
3. Почему надо тестировать API?

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

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.

2.1 Проработка лекционного материала и подготовка к контрольной работе

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 являющихся 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.

Контрольная работа выполняются студентами в аудитории, под наблюдением преподавателя.

Тема контрольной работы: Особенности процесса и технологии 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?
10. Назовите и опишите уровни тестирования.
11. Перечислите известные вам виды и стратегии тестирования, опишите их (стратегий) основные характеристики.
12. Что такое функциональное тестирование?
13. Охарактеризуйте позитивное негативное и дымовое тестирование.
14. Что оценивает нефункциональное тестирование? Примеры (виды нефункционального тестирования).
15. Что такое регрессионное тестирование?
16. Укажите причины возникновения повторных ошибок.
17. Напишите типичные ошибки при проведении регрессионного тестирования.
18. Перечислите виды регрессионного тестирования.
19. Правила проведения регрессионного тестирования.
20. Что такое автоматизированное тестирование.
21. Укажите минусы и плюсы автоматизации.
22. Каковы цели автоматизации?
23. Каким проектам противопоказана автоматизация?
24. Порядок действий при проведении автоматизации.
25. Какие тесты – лучшие претенденты на автоматизацию.
26. Как выбирать инструменты на автоматизацию.

2.2 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.

2.3 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. Проект «Хабр». – Режим доступа: https://habr.com/ru/hub/it_testing/ (дата обращения: 24.03.2022).

4. Проект Software-Testing.ru. – Режим доступа: http://software-testing.ru/ (дата обращения: 24.03.2022).

2.3.1 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?

2.3.2 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?

2.3.3 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.

2.3.4 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.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Морозова, Ю. В. Тестирование программного обеспечения: учебное пособие [Электронный ресурс] / Ю. В. Морозова. – Томск: ТУСУР, 2019. – 120 с. – ISBN 978-5-4332-0279-5. – Режим доступа: https://edu.tusur.ru/publications/9014 (дата обращения: 15.01.2022).
2. Винниченко И.В. Автоматизация процессов тестирования : производственно-практическое издание / И. В. Винниченко. – СПб. : Питер, 2005. – 202[6] с.
3. Бахтизин В.В. Автоматизация тестирования программного обеспечения. : учебн.-метод. пособие / В.В. Бахтизин, С.С. Куликов, Е.П. Фадеева. – Минск: БГУИР, 2012. – 72 с.
4. DevTools Google Chrome [Электронный ресурс]. – Режим доступа: https://developer.chrome.com/docs/devtools/ (дата обращения: 15.01.2022).
5. Обучающий центр Postman [Электронный ресурс]. – Режим доступа: https://learning.postmancom/docs/getting-started/introduction/ (дата обращения: 15.01.2022).
6. 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: 15.01.2022).