

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования

**ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ
УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ**

Факультет дистанционного обучения (ФДО)

Ю. В. Морозова

ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**Методические указания
по выполнению лабораторных работ
и организации самостоятельной работы для студентов,
обучающихся с применением
дистанционных образовательных технологий**

Томск 2020

Корректор: А. Н. Миронова

Морозова Ю. В.

Тестирование программного обеспечения : методические указания по выполнению лабораторных работ и организации самостоятельной работы для студентов, обучающихся с применением дистанционных образовательных технологий / Ю. В. Морозова. – Томск : ФДО, ТУСУР, 2020. – 53 с.

© Морозова Ю. В., 2020

© Оформление.

ФДО, ТУСУР, 2020

ОГЛАВЛЕНИЕ

Введение	4
1 Методические указания по выполнению лабораторных работ	6
1.1 Правила выполнения лабораторных работ	6
1.2 Лабораторная работа № 1 «Тестирование черного ящика»	7
1.3 Лабораторная работа № 2 «Тестирование белого ящика»	18
2 Методические указания для организации самостоятельной работы	48
2.1 Общие положения	48
2.2 Проработка лекционного материала и подготовка к контрольной работе	48
2.3 Подготовка к лабораторным работам	50
Рекомендуемые источники	51
Приложение А Форма титульного листа отчета	53

ВВЕДЕНИЕ

Квалифицированные специалисты по тестированию и контролю качества программного обеспечения (Software Quality Assurance, SQA) до сих пор являются редкостью на IT-рынке. Именно поэтому они ценятся не меньше разработчиков программного обеспечения. Многие успешные IT-менеджеры начинали свою карьеру с должности специалиста по тестированию. Цель профессии «тестировщик» – помочь создать качественный продукт. Работа SQA-специалиста – предотвратить дефекты и, следовательно, обеспечить высокое качество процесса разработки и его результатов. Для этого необходимо обладать профессиональными знаниями о видах тестирования, об инструментах и библиотеках для автоматизации тестирования; умением пользоваться специальным программным обеспечением для автоматизированного тестирования и регистрации ошибок; навыками тест-дизайна.

Цель дисциплины «Тестирование программного обеспечения» – 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 освоения дисциплины студент должен приобрести следующие знания, умения и опыт, необходимые для качественного изменения компетенций. Обучающийся должен

знать:

техники тестирования (техники, базирующиеся на интуиции и опыте инженера; техники, базирующиеся на спецификации; техники, ориентированные на код; тестирование, ориентированное на дефекты; техники, базирующиеся на условиях использования; тестирование, базирующееся на надежности инженерного процесса; техники, базирующиеся на природе приложения);

системы автоматизированного тестирования;
теорию тестирования (модели тестирования, планирование тестирования, тест-дизайн, проектирование тестов);
стандарты в области тестирования; стандарты и методологии, применяемые к необходимым приложениям;
жизненный цикл тестов, оценки надежности;
типы дефектов, классификации и статистики возникновения;
теорию различных стратегий тестирования;
базовые понятия качества программного продукта и качества процесса разработки программного обеспечения;

уметь:

понимать процесс тестирования программного обеспечения и жизненный цикл программного продукта;
оценивать покрытия требований тестовыми случаями;
определять наиболее значимые критерии качества программного продукта;

владеть навыками:

выполнения необходимых видов тестирования в соответствии с планом тестирования проведения автоматизированного тестирования, получения статистики о выполнениях тестов;
анализа полученных результатов;
определения целей, уровня тестирования, ролей и обязанностей каждого члена команды, требования к тестовым данным, определения инструментальных средств для достижения целей тестирования, требований к окружению и программному обеспечению.

Данные методические указания предназначены для организации самостоятельной работы и выполнения лабораторных работ для обучающихся по направлению подготовки «Программная инженерия» (уровень бакалавриата), профиль «Индустриальная разработка программных продуктов».

1 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ПО ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ

1.1 Правила выполнения лабораторных работ

Лабораторные занятия выполняются студентами самостоятельно.

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

Отчет оформляется в соответствии с образовательным стандартом университета ОС ТУСУР 01–2013 «Работы студенческие по направлениям подготовки и специальностям технического профиля. Общие требования и правила оформления» (введен приказом ректора от 03.12.2013 № 14103) (<https://regulations.tusur.ru/documents/70>).

Форма титульного листа представлена 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 представления фрагмента кода – Courier New, размер 12 пт. 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 тексте документа.

Для выбора варианта задания на лабораторные работы необходимо воспользоваться общим правилом с использованием следующей формулы:

$$V = (N \times K) \text{ div } 100,$$

где V – искомый номер варианта,

N – общее количество вариантов,

K – код варианта,

div – целочисленное деление.

При $V = 0$ выбирается максимальный вариант.

1.2 Лабораторная работа № 1

«Тестирование черного ящика»

Цель работы: получение практических навыков по разработке тестовых сценариев (тест-кейсов) и тестирования методом черного ящика.

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) используется, когда 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, как

- 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 использовании *черного ящика*:

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

Порядок выполнения

Есть консольное приложение для выполнения математических вычислений.

Спецификация программы. На вход программа принимает два параметра: x – целое число, y – целое число. Результат вычисления выводится на консоль.

Значения x и y должны лежать в заданном диапазоне.

Если числа, подаваемые на вход, лежат за пределами указанных диапазонов, то должно выдаваться сообщение об ошибке.

При этом необходимо учесть технологическую границу переменной **result** (например, тип `int` занимает 32 бита и имеет верхнюю границу до 2147483647).

Варианты для выполнения задания представлены в таблице 1.1.

Таблица 1.1 – Варианты задания

Вариант	Тестовый метод	Ограничения для x	Ограничения для y
1	//метод, вычисляющий степень числа. public int test(int a, int b){ int result = 1; for (int i=1; i<=b; i++){ result = result*a; } return result; }	[0..999]	[1..100]
2	//метод, вычисляющий среднее геометрическое. public double test(int a, int b){ double result = 0; result = Math.sqrt(a*b); return result; }	[0..100]	[1..100]
3	//метод, вычисляющий гипотенузу прямоугольного треугольника. public double test(int a, int b){ double result = 0; result = Math.sqrt(a*a+b*b); return result; }	[1..100]	[1..100]
4	//метод, определяющий <u>наибольший общий делитель двух чисел</u> . public int test(int a, int b){ while (a != b) { if (a > b) { a = a - b; } else { b = b - a; } } return a; }	[0..100]	[0..100]
5	//метод, определяющий наименьшее общее кратное двух чисел. public int test(int x, int y){ int j; if (x<y) j=y; else j=x; boolean t=true; while (t) { if ((j%x == 0) && (j%y == 0)) break ; else j+=1; }	[0..100]	[0..100]

Продолжение таблицы 1.1

Вариант	Тестовый метод	Ограничения для x	Ограничения для y
	<pre> } return j; } </pre>		
6	//метод, определяющий большую сумму цифр одного из двух целых неотрицательных чисел. <pre> public int test(int a, int b){ int sumNumbersA = 0; int sumNumbersB = 0; int rez=0; while (a != 0) { sumNumbersA = sumNumbersA + a % 10; a = a / 10; } while (b != 0) { sumNumbersB = sumNumbersB + b % 10; b = b / 10; } if (sumNumbersA > sumNumbersB) rez= sumNumbersA; else if (sumNumbersA < sumNumbersB) rez= sumNumbersB; return rez; } </pre>	[0..1000]	[0..1000]
7	//метод, определяющий является ли число x делителем y. <pre> public boolean test(int a, int b){ boolean result = false; if (x % y == 0) result = true; return result; } </pre>	[2..100]	[2..100]
8	//метод, определяющий факториал суммы двух чисел. <pre> public int test(int a, int b){ int sum=a+b; int res = 1; for (int i = 1; i <=sum; i++){ res = res*i; } return res; } </pre>	[0..100]	[0..100]

Окончание таблицы 1.1

Вариант	Тестовый метод	Ограничения для x	Ограничения для y
9	<pre>//метод, определяющий количество символов в произведении двух положительных чисел. public int test(int a, int b){ int sum=a*b; int rez = 0; while (sum != 0) { rez ++; sum = sum / 10; } return rez; }</pre>	[0..1000]	[0..1000]
10	<pre>//метод, определяющий сумму всех четных чисел в диапазоне [a;b], a,b>0, a>b. public int test(int a, int b){ int sum=0; for (int i = a; i <= b; i++){ if (0==(i%2)){ sum += i; } } return sum; }</pre>	[1..100]	[1..100]

Код программ:

```
import java.util.Scanner;

public class Test{

    static int x,y;

    static Scanner scan=new Scanner(System.in);

    //тестовый метод из таблицы

    public int test(int a, int b){

    }

    public int getX(){

        System.out.println("Введите x:");

        int rez=0;
```

```

Scanner scan1=new Scanner(System.in);
int num=scan1.nextInt();

if((num <= 999)&& (num >=0)) // диапазон из таблицы
{
    rez=num;
} else {
    System.out.println("Вы допустили ошибку при вводе числа. Попробуйте еще раз.");
    rez = getX();//рекурсия
}

    return rez;
}

public int getY(){
    System.out.println("Введите y:");
    int rez=0;
    Scanner scan2=new Scanner(System.in);
    int num=scan2.nextInt();

    if((num<= 100)&&(num >= 1)) // диапазон из таблицы
    {
        rez = num;

    } else {
        System.out.println("Вы допустили ошибку при вводе числа. Попробуйте еще раз.");
        rez = getY();//рекурсия
    }
}

```

```

    return rez;

}

public void printRez(int x, int y){
    System.out.println("Результат вычисления равен" + test(x,n));
}

public static void main(String[] args){

    Test app=new Test();
    x = app.getX();
    y = app.getY();
    app.printRez(x,y);
}

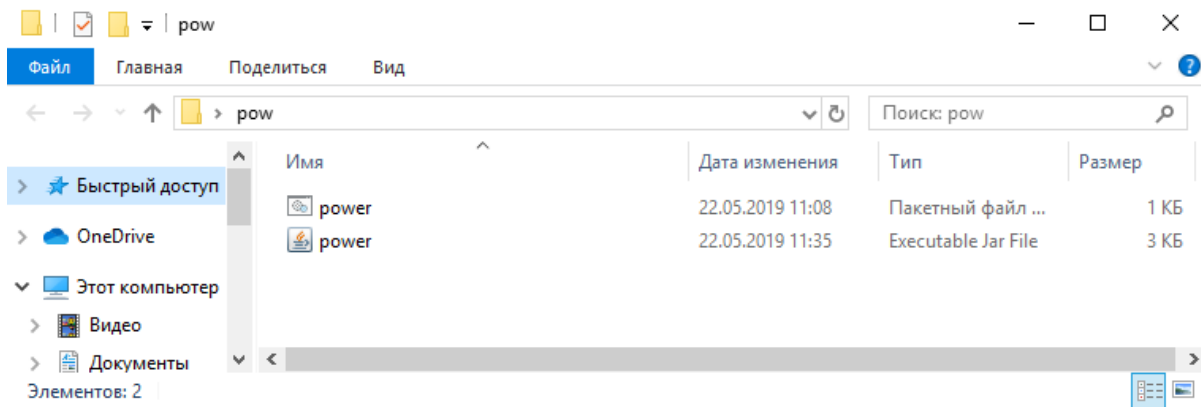
}

```

Для приведенной программы воспроизведите последовательность действий, необходимых для тестирования.

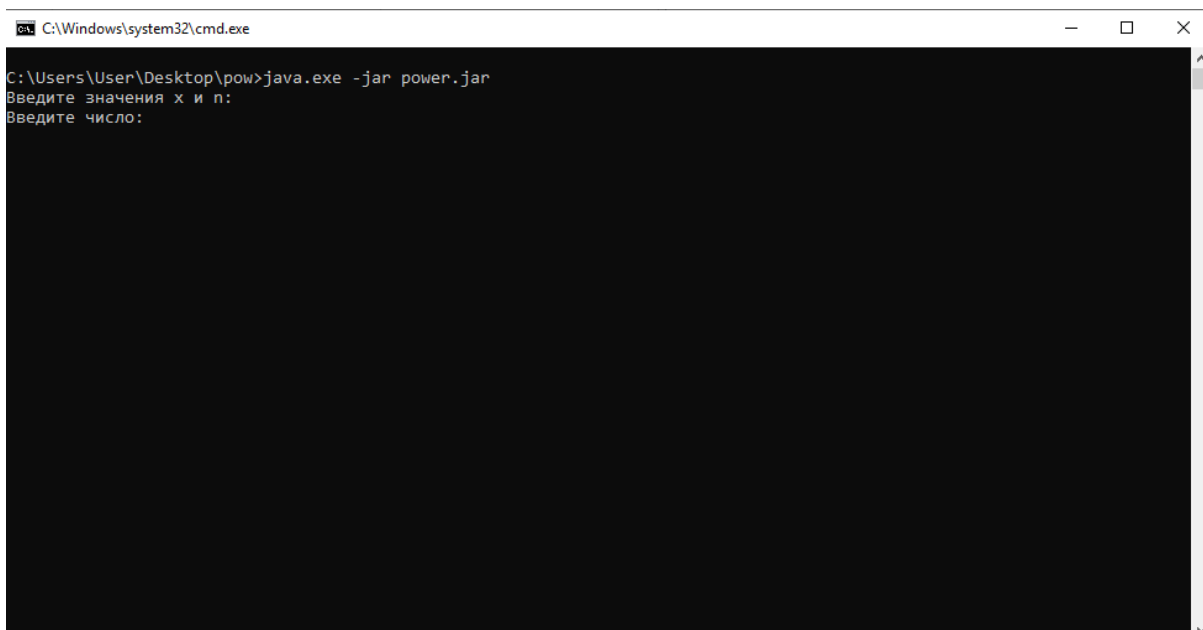
Скачать программу можно в электронном курсе по дисциплине: ВариантN.zip, где N – номер варианта. В архиве лежат исполняемый файл TestN.jar и пакетный файл test.bat.

Для запуска распакуйте архив на Рабочий стол.



Пакетный файл представляет собой текстовый файл, содержащий последовательность команд для операционной системы компьютера. Именно в нем прописаны команды для запуска TestN.jar.

Запустите его, кликнув по нему два раза. Откроется командная строка (консоль).



Далее можно вводить данные из тест-кейсов.

Сначала введите значение *x* и нажмите Enter.

```

C:\Windows\system32\cmd.exe
C:\Users\User\Desktop\pow>java.exe -jar power.jar
Введите значения x и n:
Введите число:
2
Введите число:

```

Далее введите значение n и снова Enter.

```

C:\Programs\Programs\Power\power.jar
Введите значения x и n:
Введите число:
5
Введите значения x и n:
Введите число:
C:\Programs\Programs\Power\power.jar

```

Разработка тестов

1. Определите области эквивалентности входных параметров.

Целью данной техники является не только сокращение числа тестов, но и сохранение приемлемого тестового покрытия.

Примерный алгоритм использования:

- *Определить классы эквивалентности.* Это главный шаг техники. От него во многом зависит эффективность ее применения.
- *Выбрать одного представителя от каждого класса.* На этом шаге из каждого эквивалентного набора тестов выбирается один тест.
- *Выполнить тесты.* На этом шаге выполняются тесты от каждого класса эквивалентности.

Пример

Размер скидки в интернет-магазине рассчитывается в зависимости от потраченной суммы следующим образом:

Скидка = 10% при заказе на сумму больше или равную 5 000 руб.

Скидка = 5% при заказе на сумму меньше 5 000 руб.

Сумма заказа не может быть равной 0.

На какие классы эквивалентности можно разбить сумму для проверки правильности предоставляемой скидки?

Ответ:

Класс 1 = $0 < \text{сумма} < 5000$.

Класс 2 = $\text{сумма} \geq 5000$.

2. Проведите анализ тестовых случаев, а также учтите граничные значения. Необходимо добиться полного тестирования.

Примерный алгоритм использования техники анализа граничных значений:

- Выделить классы эквивалентности. Это очень важный шаг, и от правильности разбиения на классы эквивалентности зависит эффективность тестов граничных значений.
- Определить граничные значения этих классов.
- Определить, к какому классу будет относиться каждая граница.

Пример

Поле «пароль» принимает количество символов от 6 до 20. Каковы будут граничные значения?

Ответ:

Для нижнего – 5, 6.

Для верхнего – 20, 21.

3. Запишите позитивные и негативные тест-кейсы с входными значениями, покрывающие каждый класс эквивалентности.

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 хотя бы один некорректный параметр) и ставит целью проверку исключительных ситуаций.

4. Выполните тесты с заданными значениями. Проведите сравнение фактического с ожидаемым и укажите статус Fail/Pass.

5. Напишите рекомендации для улучшения/исправления программы.

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.

1.3 Лабораторная работа № 2

«Тестирование белого ящика»

Цель работы: овладение практическими навыками выполнения модульного тестирования.

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 используется, 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.

Покрывание 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-тестирование (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.

Существуют различные инструменты (например, *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.

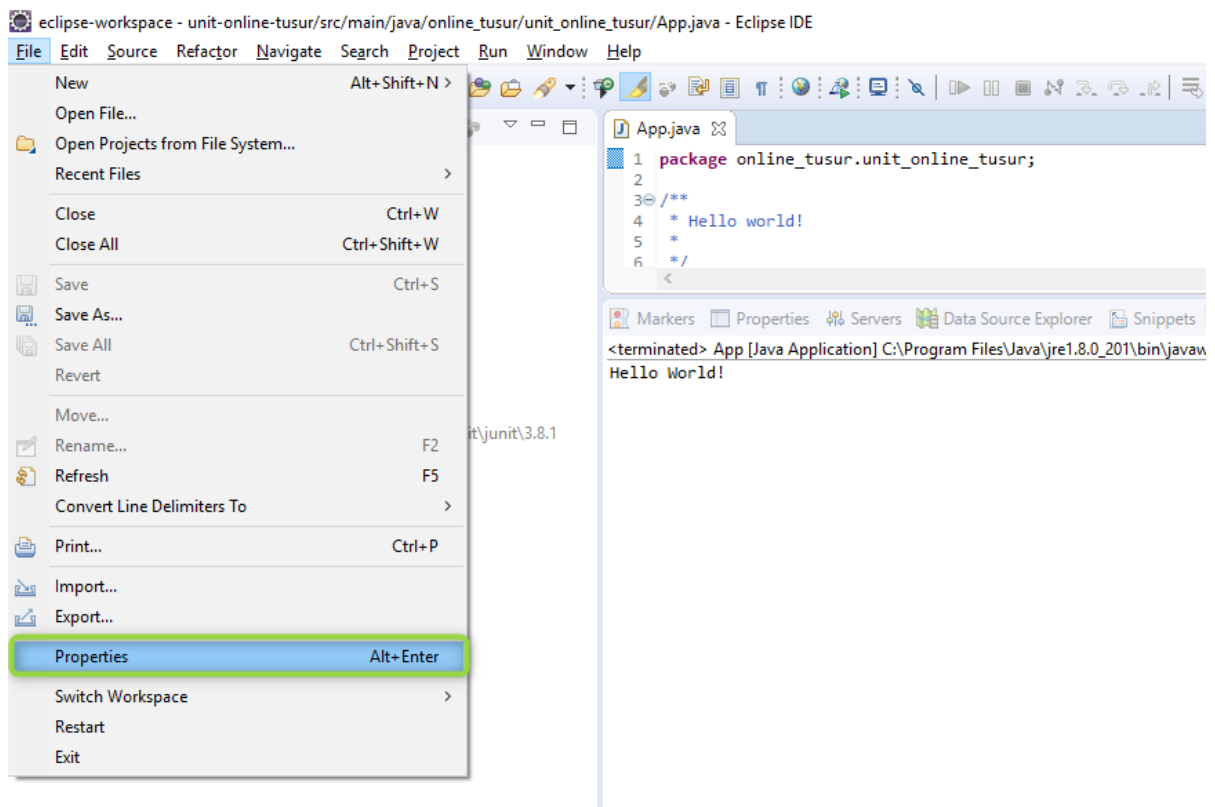
3. JUnit Vintage, который предоставляет поддержку старых тестов, – это тестовый фреймворк, на котором можно запускать все тесты JUnit 3 и JUnit 4.

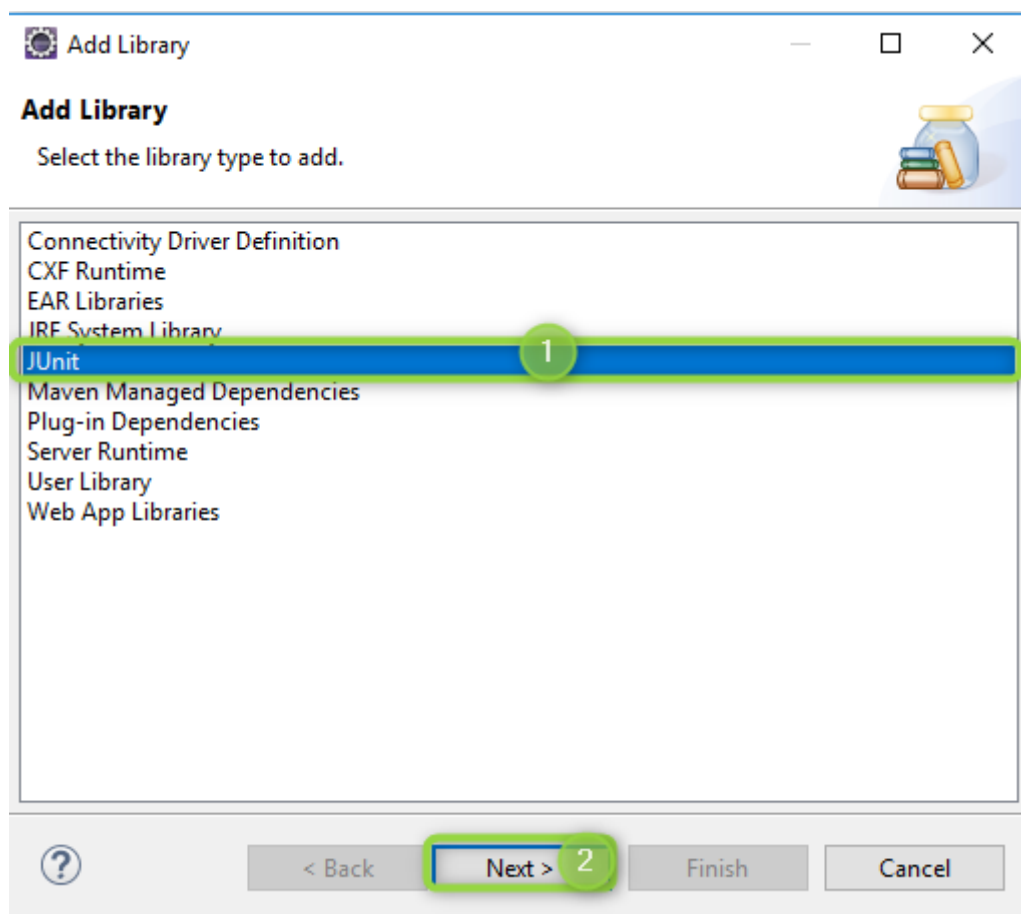
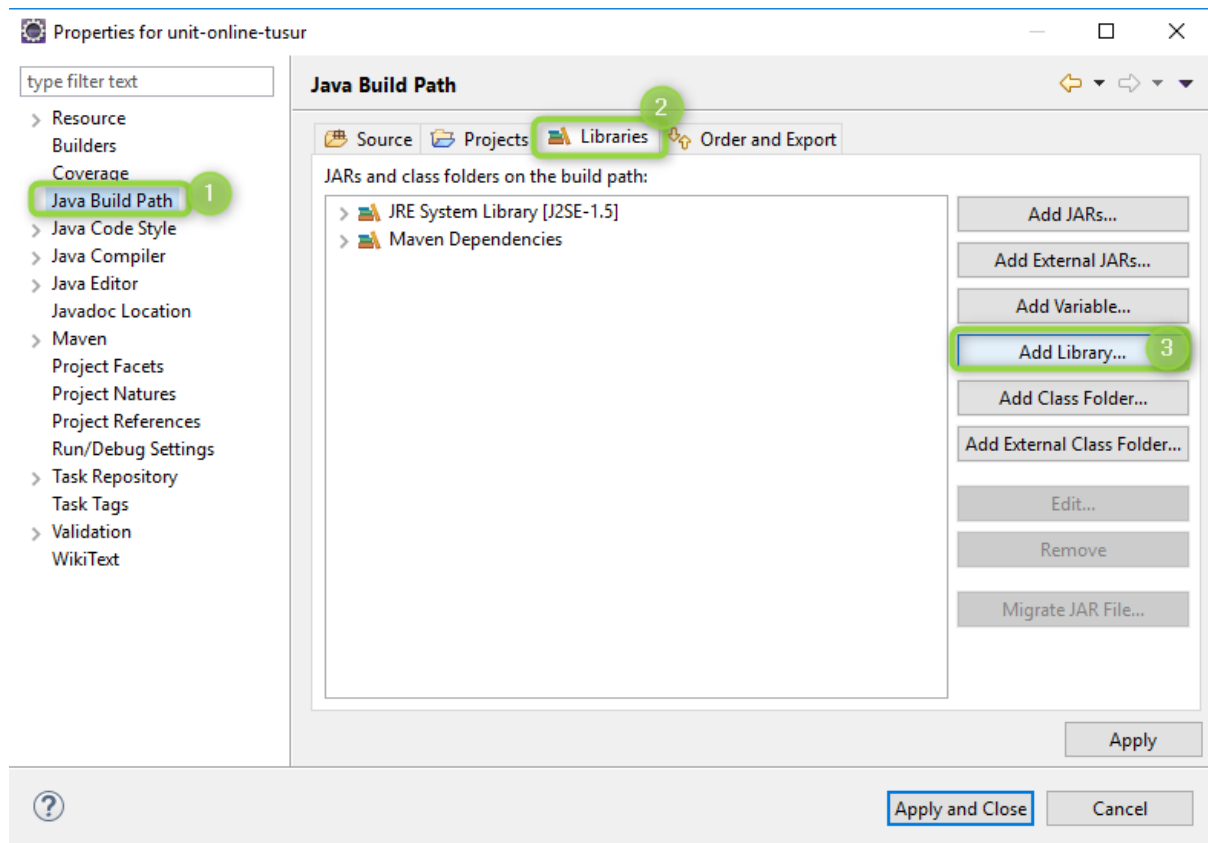
Дополнительные функции:

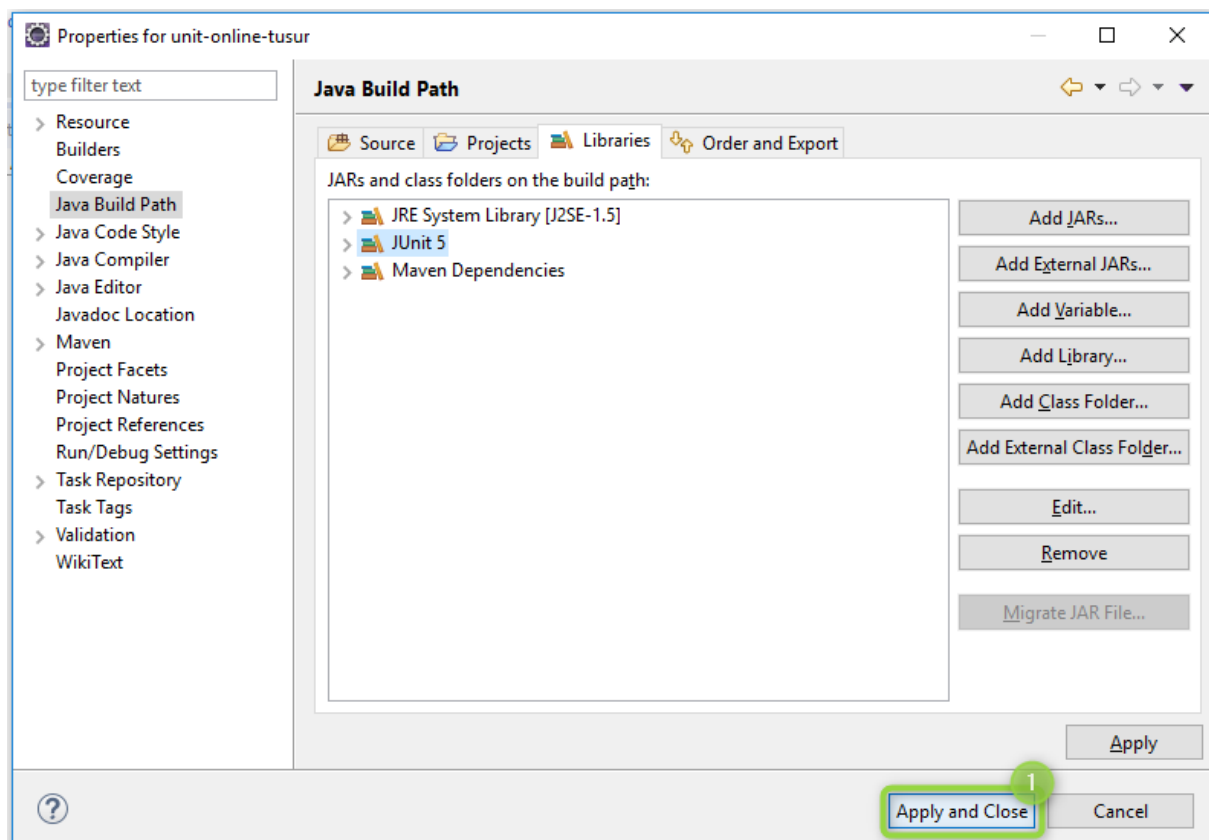
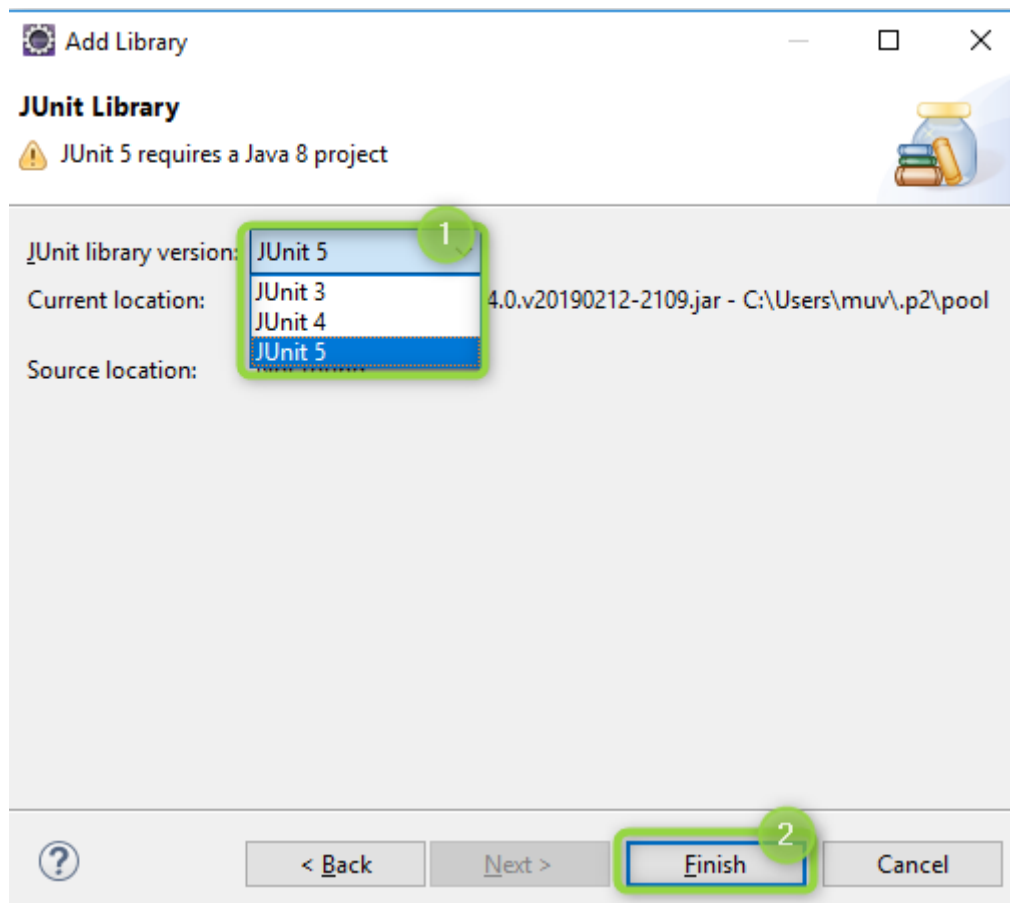
- JUnit больше не требует, чтобы методы были публичными.
- Опциональное сообщение сделали последним аргументом:
`assertEquals(expected, actual, "The optional assertion message");`
- Появился метод для работы с Iterable: `assertIterableEquals();`

- 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.







Теперь можно создать тест, но даже для самого простого теста необходимо указать соответствующую аннотацию `@Test` (табл. 1.2).

Таблица 1.2 – Основные аннотации JUnit

Аннотация	Описание
<code>@BeforeEach</code>	Аннотированный метод будет запускаться перед каждым тестовым методом (аннотированным <code>@Test</code>) в текущем классе (JUnit 5)
<code>@AfterEach</code>	Аннотированный метод будет запускаться после каждого тестового метода (с аннотацией <code>@Test</code>) в текущем классе (JUnit 5)
<code>@BeforeAll</code>	Аннотированный метод будет запущен перед всеми тестовыми методами в текущем классе (JUnit 5)
<code>@AfterAll</code>	Аннотированный метод будет запущен после всех тестовых методов в текущем классе (JUnit 5)
<code>@Test</code>	Аннотация <code>@Test</code> определяет, что метод или класс является тестовым
<code>@DisplayName</code>	Определяет отображаемое имя класса теста или метода теста (JUnit 5)
<code>@Disable</code>	Используется для отключения или игнорирования тестового класса или метода (JUnit 5)
<code>@Nested</code>	Используется для объявления вложенных тестовых классов (JUnit 5)
<code>@Tag</code>	Объявление тега для обнаружения и фильтрации тестов (JUnit 5)
<code>@TestFactory</code>	Обозначает, что метод – это фабрика тестов для динамических тестов (JUnit 5)
<code>@Ignore</code>	Аннотация заставляет инфраструктуру тестирования проигнорировать данный тестовый метод (JUnit 4)
<code>@Rule</code>	Аннотация позволяет более гибко работать с классами-утилитами, определяющими правила работы с тестами

Окончание таблицы 1.2

Аннотация	Описание
@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 inva-
lid.");
    }
}

```

```

        this.salary = salary;
    }

    public double getGrossSalary() {
        return this.salary + getSocialInsurance() + getAdditionalBo-
nus();
    }

    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 expectedAdditionalBonus = salary * 0.1;
```

```

    assertEquals(expectedAdditionalBonus, app.getAdditionalBonus());

}

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

    double expectedAdditionalBonus = salary * 0.1;
    double expectedSocialInsurance = salary * 0.25;

    double expectedGross = salary + expectedSocialInsurance + expectedAdditionalBonus;
    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.

```

1 -----
2 Test set: online_tusur.unit_online_tusur.SalaryTest
3 -----
4 Tests run: 4, Failures: 2, Errors: 0, Skipped: 0, Time elapsed: 0.039 s <<< FAILURE! - in online_tusu
5 test_assert Time elapsed: 0.012 s <<< FAILURE!
6 java.lang.AssertionError
7     at online_tusur.unit_online_tusur.SalaryTest.test_assert(SalaryTest.java:12)
8
9 testSocialSalaryWithValidSalary Time elapsed: 0.002 s <<< FAILURE!
10 org.opentest4j.AssertionFailedError: expected: <2000.0> but was: <1000.0>
11     at online_tusur.unit_online_tusur.SalaryTest.testSocialSalaryWithValidSalary(SalaryTest.java:19)
12
13

```

Поэтому рекомендуется использовать тестовые методы класса Assert, тем более существует больше их количество на все случаи (табл. 1.3).

Таблица 1.3 – 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
assertEquals(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»

Окончание таблицы 1.3

Метод	Описание
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 [14]. 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.

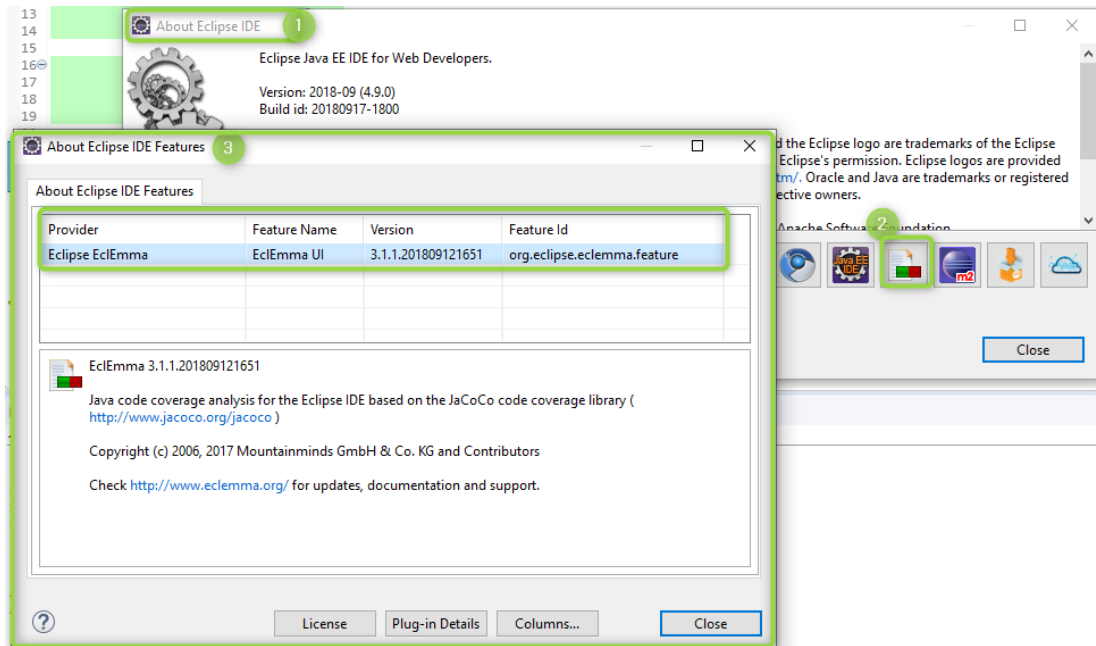
Анализ производится по требованию или после того, как приложение было выполнено и завершено. В среде Eclipse автоматически появляется информация по покрытию кода:

- 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).

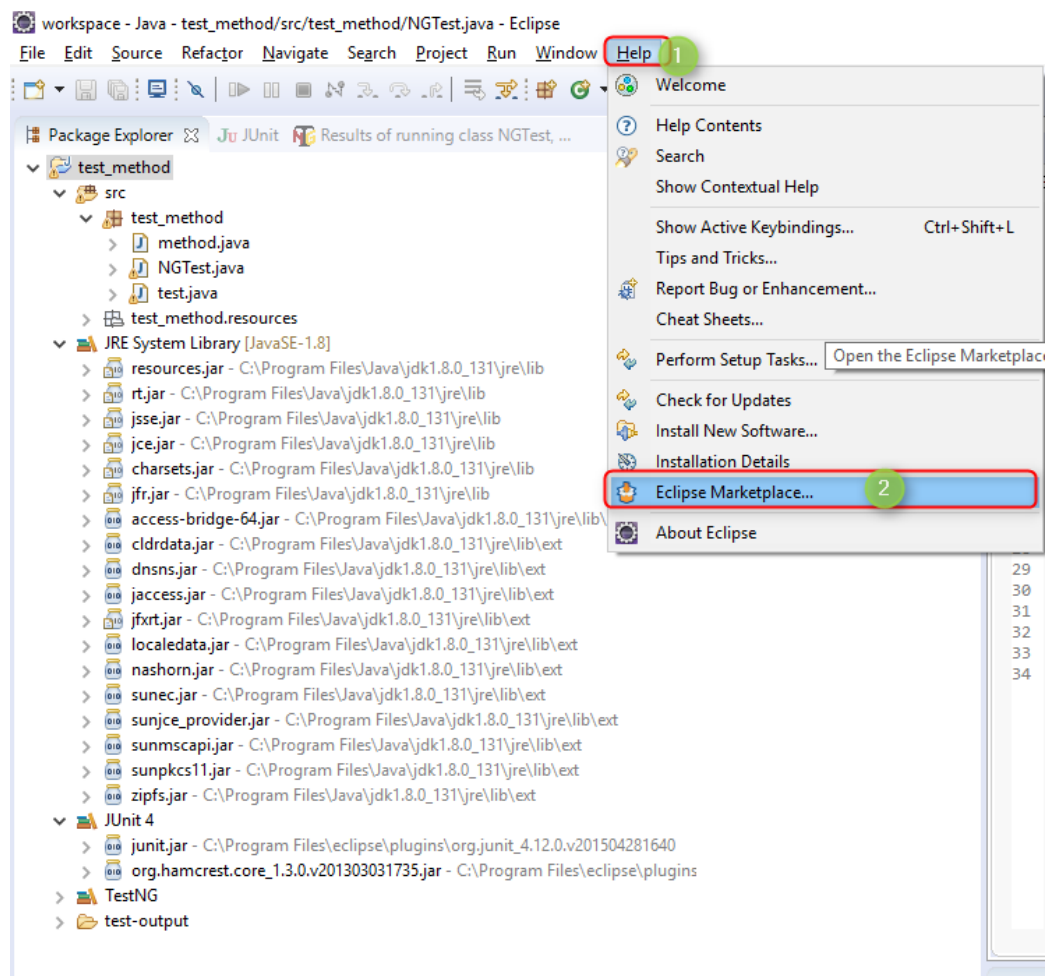
Для EclEmma необходимы Eclipse 3.5 (или выше) и Java 1.5 (или выше).

Новые версии Eclipse уже имеют встроенную EclEmma.



Если у вас старые версии Eclipse, то установить новый инструмент можно в несколько шагов:

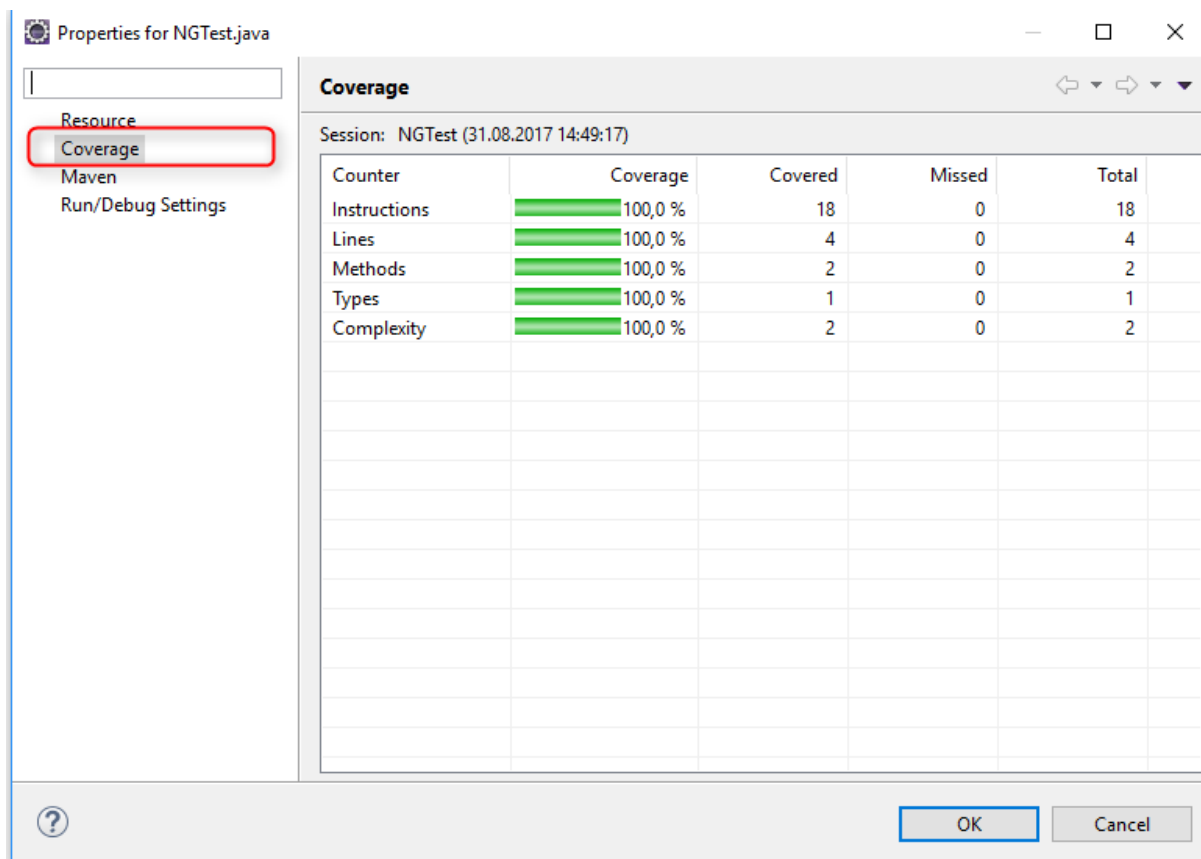
1. В меню Eclipse нажмите **Help** → **Eclipse Marketplace**.



2. Далее в окошке в поле Find укажите EcIEmma, найдите нужный плагин и установите его, выбрав Install.

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 скриншоте ниже.



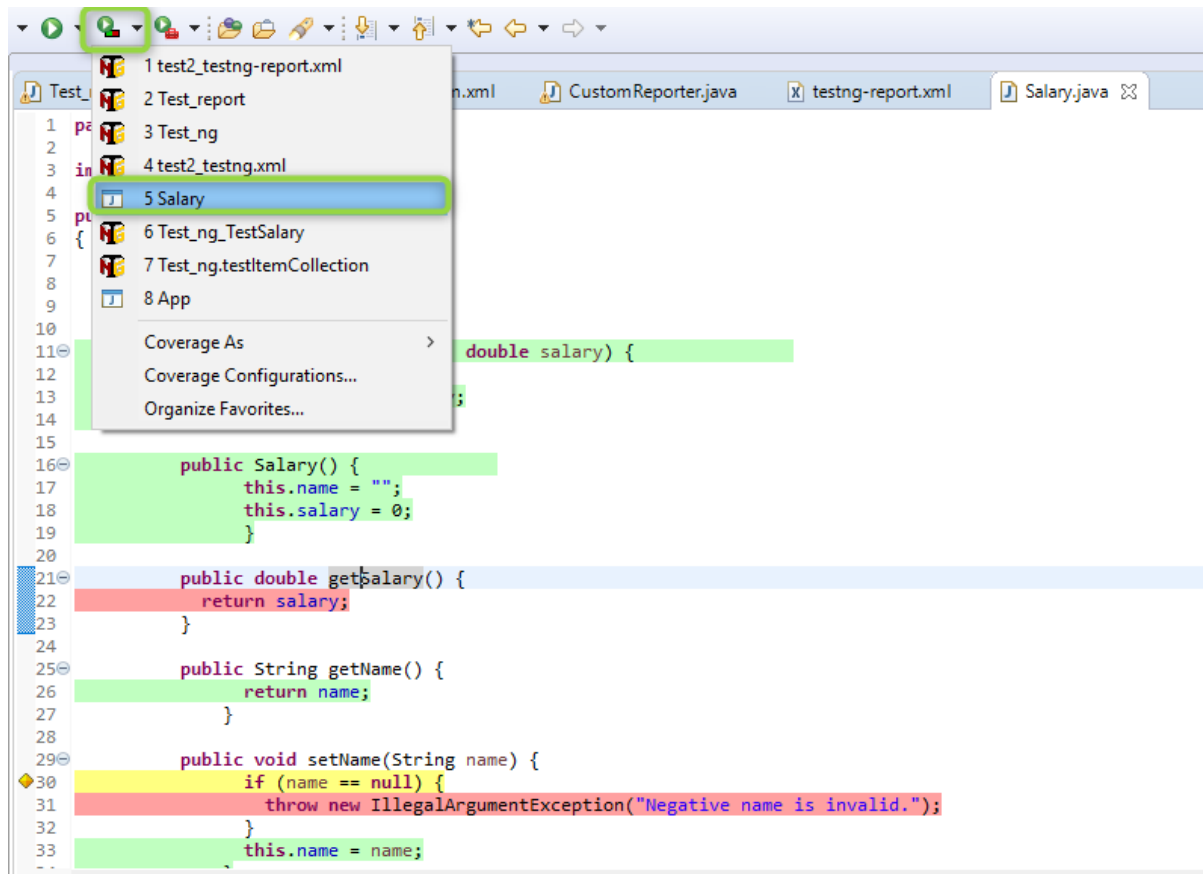
Набор покрытия создается каждый раз, когда выполняется тест или приложение с измерением покрытия кода.

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.

Измерим покрытие кода нашей простой программы по расчету бонусов:

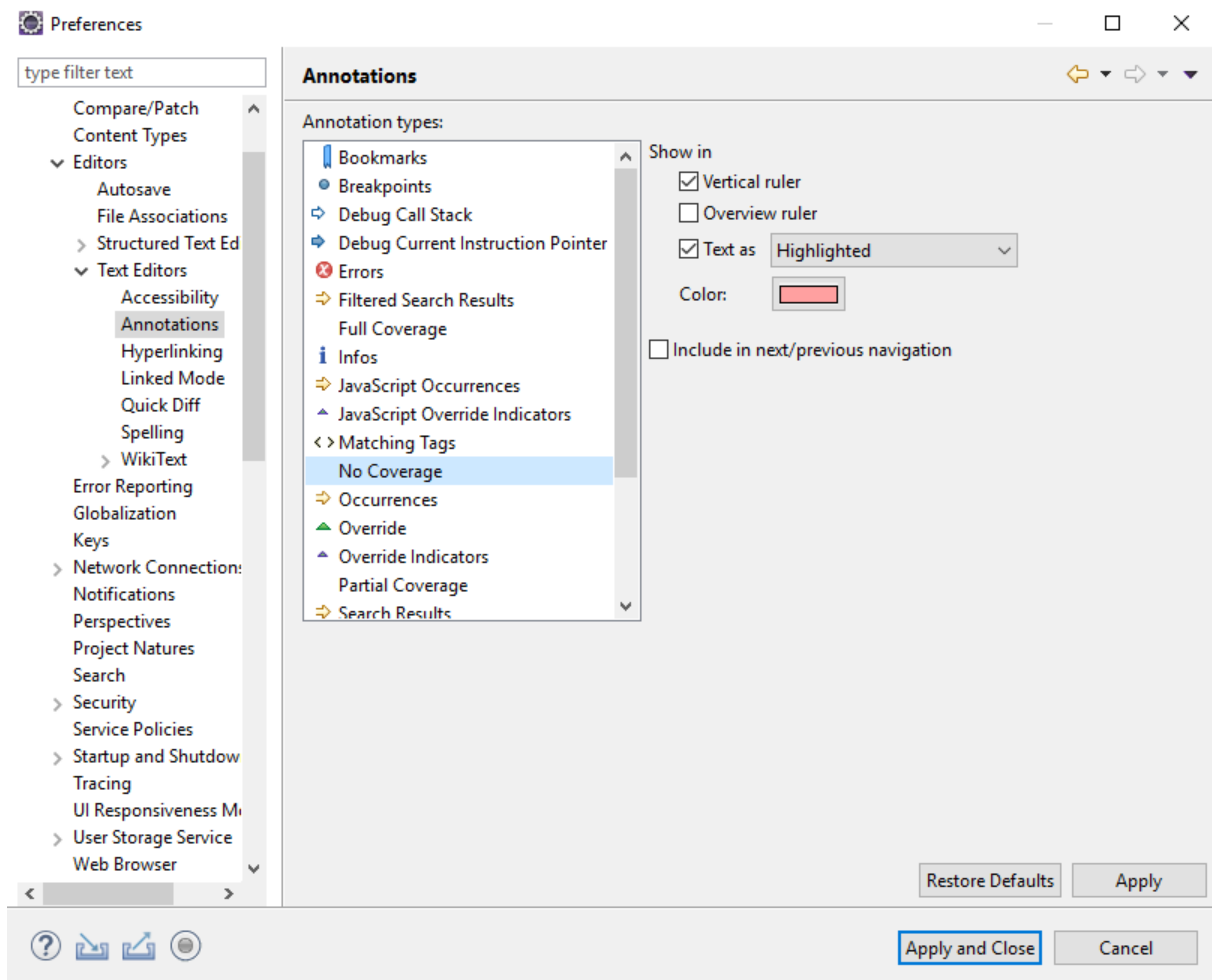


Кроме того, цветные бриллианты показаны слева для строк, содержащих ветви условий. Цвета бриллиантов имеют идентичную семантику с цветами линий:

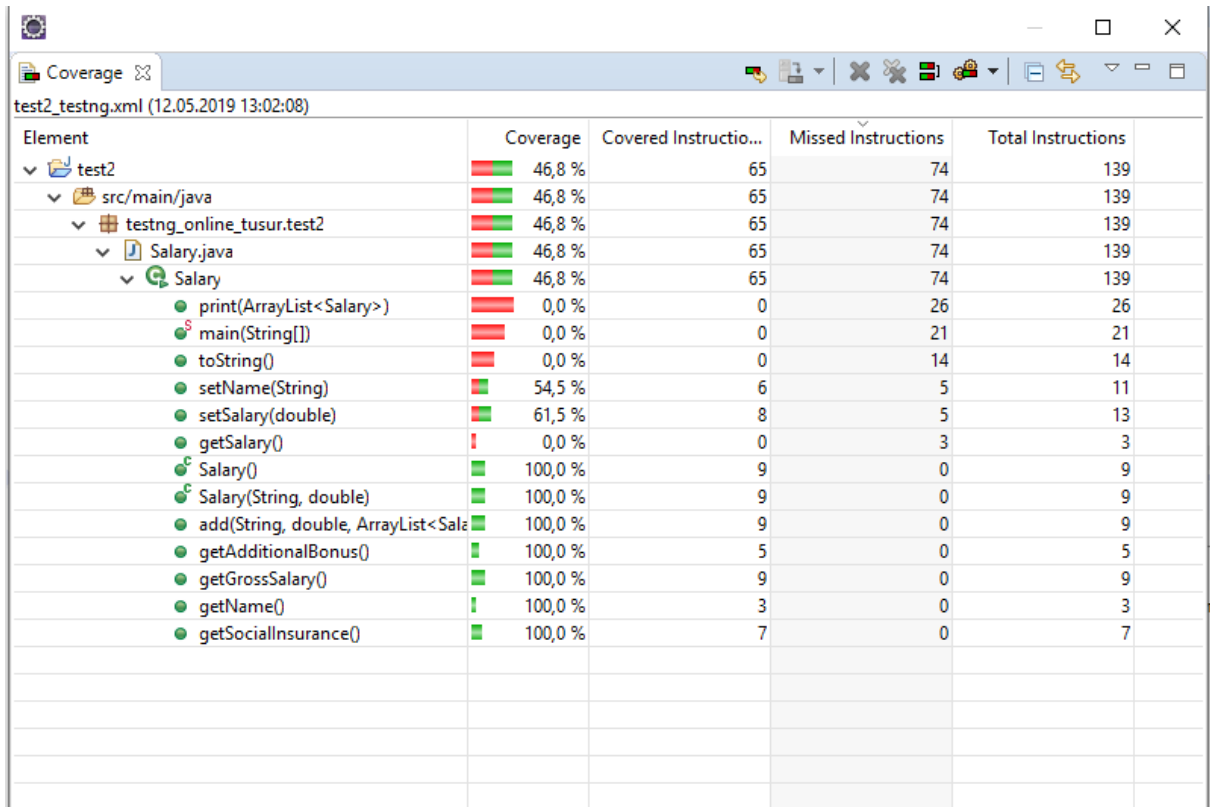
- зеленый для полностью покрытых ветвей;
- желтый для частично покрытых ветвей;
- красный, если не было выполнено ни одной ветки в конкретной строке.

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.



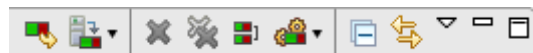
Как видим, большая часть кода у нас не покрыта тестами.



test2_testng.xml (12.05.2019 13:02:08)

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<Sale	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

Рассмотрим панель инструментов и раскрывающееся меню на Coverage:

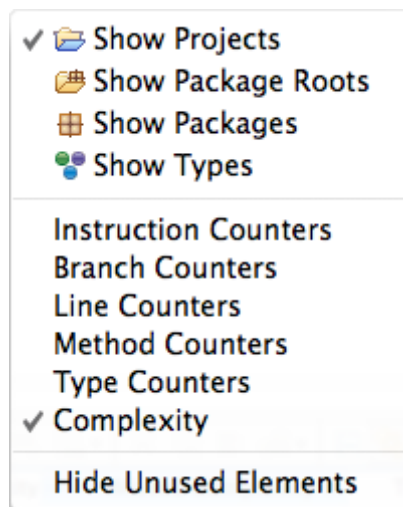


Панель инструментов представления покрытия предлагает следующие действия:

- 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 expectedAdditionalBonus = salary * 0.1;
    double expectedSocialInsurance = salary * 0.25;
    double expectedGross = salary + expectedSocialInsurance + expectedAdditionalBonus;

    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 так же можно тестировать входящий поток `Scanner`:

```
@Test
void testMain(String[] args) {
    //ожидаемое
```

```

String expectedValue ="Введите число:\r\nВведите степень числа:\r\nСте-
пень 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.

All suites

Suite

Info

- C:\Users\User\workspace\test2\testng.xml
- 1 test
- 0 groups
- Times
- Reporter output
- Ignored methods
- Chronological view

Results

- 10 methods, 2 failed, 8 passed
- Failed methods (hide)
 - ✖ testsName(null, null)
 - ✖ testsSs
- Passed methods (hide)
 - ✔ testEmptyCollection
 - ✔ testMain
 - ✔ testOneItemCollection
 - ✔ testPrint
 - ✔ testSalaryWithValidSalary
 - ✔ testToString
 - ✔ testsName(Иванов, Иванов)
 - ✔ testsS

TestNg.java testng.xml test2/pom.xml CustomReporter.java testng-report.xml Salary.java TestNG reports

```

13         this.salary = salary;
14     }
15
16     public Salary() {
17         this.name = "";
18         this.salary = 0;
19     }
20
21     public double getSalary() {
22         return salary;
23     }
24
25     public String getName() {
26         return name;
27     }
28
29     public void setName(String name) {
30         if (name == null) {
31             throw new IllegalArgumentException("Negative name is invalid.");
32         }
33         this.name = name;
34     }
35
36     public void setSalary(double salary) {
37         if (salary < 0) {
38             throw new IllegalArgumentException("Negative salary is invalid.");
39         }

```

Results of running suite
Console
Coverage

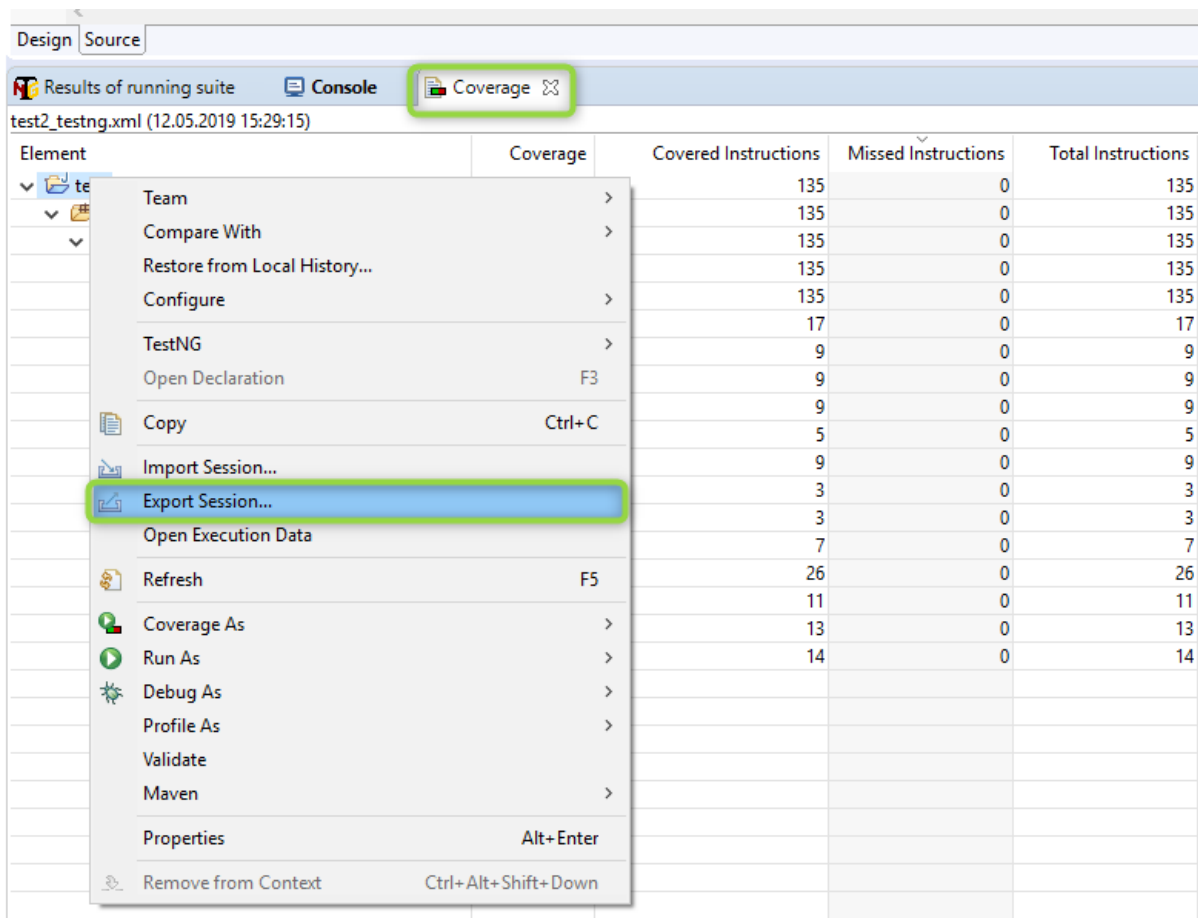
test2 testng.xml (12.05.2019 15:04:49)

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

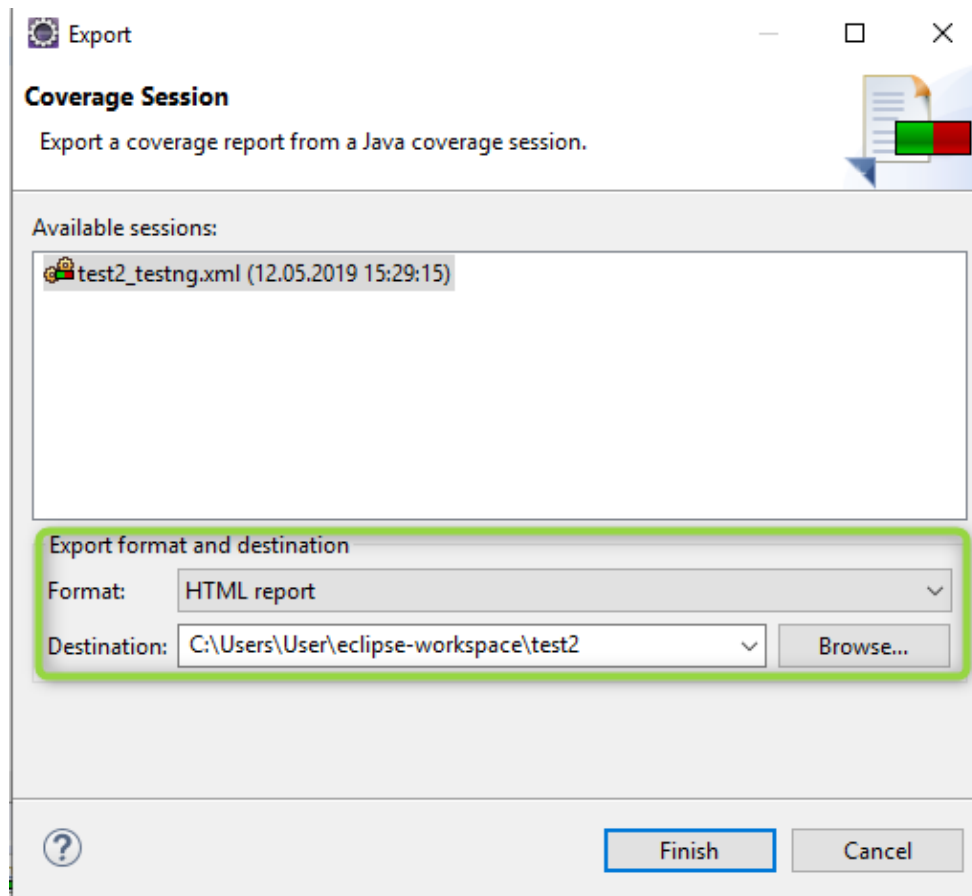
Мастер экспорта позволяет экспортировать сессии покрытия в одном из следующих форматов:

- 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.



Выбираем формат экспорта:



Обновив проект, увидим новую папку jacoco-resources и отчет:

Results of running suite

file:///C:/Users/User/eclipse-workspace/test2/src/main/java/testng_online_tusur.test2/Salary.html

test2_testng.xml (12.05.2019 15:29:15) > test2 > src/main/java > testng_online_tusur.test2 > Salary

Salary

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods
print(ArrayList)		100%		100%	0 2	0 2	0 1
main(String[])		100%		n/a	0 1	0 5	0 1
toString()	26	100%		n/a	0 1	0 2	0 1
setSalary(double)		100%		100%	0 2	0 4	0 1
setName(String)		100%		100%	0 2	0 4	0 1
Salary(String_double)		100%		n/a	0 1	0 4	0 1
Salary()		100%		n/a	0 1	0 4	0 1
getGrossSalary()		100%		n/a	0 1	0 1	0 1
add(String_double_ArrayList)		100%		n/a	0 1	0 2	0 1
getSocialInsurance()		100%		n/a	0 1	0 1	0 1
getAdditionalBonus()		100%		n/a	0 1	0 1	0 1
getSalary()		100%		n/a	0 1	0 1	0 1
getName()		100%		n/a	0 1	0 1	0 1
Total	0 of 135	100%	0 of 6	100%	0 16	0 32	0 13

test2_testng.xml (12.05.2019 15:29:15)

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

1. Установите **Eclipse** (в пособии используется Java EE IDE for Web Developers, Version: 2018-09 (4.9.0)). Скачать ПО можно по ссылке: <https://www.eclipse.org/downloads/packages/release/2018-09/r/eclipse-ide-java-ee-developers> .

2. Создайте **Java-проект**.

3. Добавьте в проект **JUnit5**.

4. Соберите проект.

5. Добавьте в файл **App.java** код программы из лабораторной работы № 1 (согласно варианту), которую будем тестировать.

6. Добавьте в проект в **AppTest.java** простейший класс с тестовым методом:

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
public class AppTest {
private Test app = new Test();

@Test()
public void test1() {
int n=1;
int x=10;
```

```

int expected = 10; //ожидаемое значение
double actualValue = app.test(x,n); //фактическое, полученное методом
test()

assertEquals(expected, actualValue); //сравниваем ожидаемое с фактиче-
ским
}

}

```

Запустите проект и убедитесь, что созданный тестовый метод запускается.

7. Разработанные ранее тесты (см. лабораторную работу № 1) 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 (используйте мастер экспорта)*.

Комментарии в коде программы приветствуются!

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.

2 МЕТОДИЧЕСКИЕ УКАЗАНИЯ ДЛЯ ОРГАНИЗАЦИИ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

2.1 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 изучение тем теоретической части курса, подготовку к лабораторным и контрольной работам, оформление отчетов по лабораторным работам.

2.2 Проработка лекционного материала и подготовка к контрольной работе

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

Проработка лекционного материала включает:

- чтение студентами рекомендованной литературы и усвоение теоретического материала курса;
- знакомство с интернет-источниками;
- подготовку к контрольной работе;
- 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.

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 знаний по дисциплине рекомендуется использовать интернет-ресурсы; проводить поиски в различных системах и использовать материалы сайтов, рекомендованных преподавателем.

Задачи, стоящие перед студентом при подготовке к выполнению контрольной работы:

- закрепление полученных ранее теоретических знаний;
- выработка навыков самостоятельной работы.

Вопросы для подготовки к контрольной работе

1. Каковы цели тестирования?
2. Назовите 7 принципов тестирования и опишите их суть.
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 они пишутся?
9. Что такое чек-лист, для чего он 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? Приведите примеры (виды нефункционального тестирования).
15. Что такое регрессионное тестирование?
16. Укажите причины возникновения повторных ошибок.
17. Укажите типичные ошибки при проведении регрессионного тестирования.
18. Перечислите виды регрессионного тестирования.
19. Назовите 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. Приведите порядок действий при проведении автоматизации.
25. Какие тесты – лучшие претенденты на автоматизацию?

2.3 Подготовка к лабораторным работам

Проведение лабораторных работ включает в себя следующие этапы:

- постановку темы и определение задач лабораторной работы;
- определение порядка лабораторной работы или отдельных ее этапов;
- непосредственное выполнение лабораторной работы студентами;
- подведение итогов лабораторной работы и формулирование основных выводов;
- оформление отчета по лабораторной работе.

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

РЕКОМЕНДУЕМЫЕ ИСТОЧНИКИ

1. IEEE Standard 610–90 (Standard Glossary of Software Engineering Terminology) [Электронный ресурс]. – URL: <https://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf> (дата обращения: 03.04.2019).
2. Бек, К. Экстремальное программирование. Разработка через тестирование / Кент Бек. – СПб. : Питер, 2017. – 224 с.
3. Криспин, Л. Гибкое тестирование. Практическое руководство для тестирующих ПО и гибких команд / Лиза Криспин, Джанет Грегори. – М. : ООО «ИД «Вильямс», 2010. – 464 с.
4. Гудлиф, П. Ремесло программиста. Практика написания хорошего кода / П. Гудлиф. – СПб. : СимволПлюс, 2009. – 704 с.
5. Майерс, Г. Искусство тестирования программ : пер. с англ. / Гленфорд Дж. Майерс ; ред. пер. Б. А. Позин. – М. : Финансы и статистика, 1982. – 176 с.
6. Савин, Роман. Тестирование Дот Ком, или Пособие по жестокому обращению с багами в интернет-стартапах [Электронный ресурс] / Р. Савин. – URL: adm-lib.ru/testirovanie/testirovanie-dot-com.html (дата обращения: 15.05.2019).
7. Уиттакер, Дж. Как тестируют в Google / Дж. Уиттакер, Дж. Арбон, Дж. Каролло. – СПб. : Питер, 2014. – 320 с.
8. Калбертсон, Р. Быстрое тестирование / Р. Калбертсон, К. Браун, Г. Кобб. – М. : Вильямс, 2002. – 384 с.
9. Бейзер, Б. Тестирование черного ящика. Технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. – СПб. : Питер, 2004. – 320 с.
10. Портал Software-Testing.Ru [Электронный ресурс]. – URL: <http://software-testing.ru/> (дата обращения: 15.05.2019).
11. Сайт IntelliJ IDEA [Электронный ресурс]. – URL: <https://www.jetbrains.com/idea/> (дата обращения: 15.05.2019).

12. Сайт фонда Eclipse [Электронный ресурс]. – URL: <https://www.eclipse.org/> (дата обращения: 15.05.2019).

13. Сайт Apache Maven [Электронный ресурс]. – URL: <http://maven.apache.org> (дата обращения: 15.05.2019).

14. Сайт JUnit [Электронный ресурс]. – URL: <http://junit.org> (дата обращения: 15.05.2019).

15. Руководство пользователя JUnit 5 [Электронный ресурс] // Сайт JUnit. – URL: <http://junit.org/junit5/docs/current/user-guide/> (дата обращения: 15.05.2019).

16. Сайт Java Code Coverage for Eclipse [Электронный ресурс]. – URL: <https://www.jacoco.org/> (дата обращения: 15.05.2019).

ПРИЛОЖЕНИЕ А
Форма титульного листа отчета

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное
учреждение высшего образования
«ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ» (ТУСУР)

Кафедра автоматизации обработки информации (АОИ)

НАЗВАНИЕ ЛАБОРАТОРНОЙ РАБОТЫ

Отчет по лабораторной работе № ____ по дисциплине
«Тестирование программного обеспечения»

Студент гр. ____
____ И. О. Фамилия
« ____ » _____ 20__ г.

Руководитель
доцент каф. АОИ,
канд. техн. наук
____ Ю. В. Морозова
« ____ » _____ 20__ г.

Томск 20__