# “Современные системы программирования”
# Лабораторная работа №6
## Введение в тестирование и отладку кода

Антон Кабыш

## Введение «О тестировании кода»

Что бы убедится, что программа работает корректно её работу необходимо протестировать. В самом простом случае тестирование программы заключается в том, что Вы запустили программу “поглядели” на результат её работы.

Это поможет для простых задач, но в больших проектах работать не будет. Не будет работать по той причине, что не протестированный код будет вызывать другой непротестированный код. И если где-либо будет ошибка, то найти её будет очень непросто. Придется смотреть на код, изучать возможные пути исполнения, которые могли привести к ошибке, а потом думать где же в этих путях возможна ошибка. Если вы хорошо знаете свой код, то ошибка найдется скоро (если вам повезет). Если знаете код плохо, или он слишком сложен, то наступает паника и долгие часы отладки.

Более продвинутые программисты будут пользоваться отладчиком для поиска ошибки, вставлять диагностические `System.out.prinltn(...)` в тело кода, или изучать журнал логов работы программы. Но эти все подходы помогают вылечить симптомы (найти уже существующую ошибку), но не решают общую проблему — как предотвратить появление ошибок в программе, как уменьшить вероятность появления ошибки в новом коде.

Умные же разработчики делают так, что бы вероятность попадания ошибки в код, и время её поиска в программе было минимальным. Для этого есть много методологий программирования, способов организации кода, а так же множество подходов к тестированию кода. Одной из таких методологий, получившей в настоящее время повсеместное признание, является модульное тестирование. Основной принцип модульного тестирования заключается в том, что программа разбивается на модули,

каждый из которых тестируется по отдельности. Программа состоящая из протестированных модулей сама по себе скорее всего будет корректна.

В языке программирования модулями являются ровно две вещи:

- Процедуры/функции/методы.
- Классы и создаваемые на их основе объекты.

Проблема уменьшения вероятности попадания ошибки в код решается тем способом, что тесты пишутся до начала написания кода. Проблема времени поиска решается следующим образом — если в программе была допущена ошибка, то упадет один из ранее написанных тестов, что служит индикатором места где в программе затаилась ошибка.

# 1 Вводное задание (0.5)

Прежде чем начать выполнять основные задания, выполните пробное тестирование очень простого статического метода `Sum.accum`, выполняющего сумму переданных значений и разберитесь с библиотекой JUnit.

- Создаете новый класс и копируете код класса `Sum`.
- Создаете тестовый класс `SumTest`. Как создавать тесты см. раздел «Подключение и настройка JUnit».
- Пишите тест к методу `Sum.accum` и проверяете его исполнение. Тест должен проверять работоспособность функции `accum`.
- Очевидно, что если передать слишком большие значения в `Sum.accum`, то случится переполнение. Модифицируйте функцию `Sum.accum`, чтобы она возвращала значение типа `long` и напишите новый тест, проверяющий корректность работы функции с переполнением. Первый тест должен работать корректно.
- Если что-то не получается, то спрашиваете коллег или преподавателя. Если разобрались — то переходите к основному заданию.

```
public class Sum {

    public static int accum(int... values) {
        int result = 0;
        for (int i = 0; i < values.length; i++) {
            result += values[i];
        }
        return result;
    }
}
```

# 2 Основные сведения про JUnit

## Подключение и настройка JUnit

Для быстрого создания тестов нужно установить курсор в строке объявления класса, нажать Alt + Enter и в контекстном меню выбрать `Create Test`.

После этого следует указать где будет располагаться тест и выбрать библиотеку тестирования (JUnit 4). Для того, чтобы классы библиотеки были подключены к проекту необходимо нажать кнопку `Fix`), тестируемые методы и дополнительные опции.

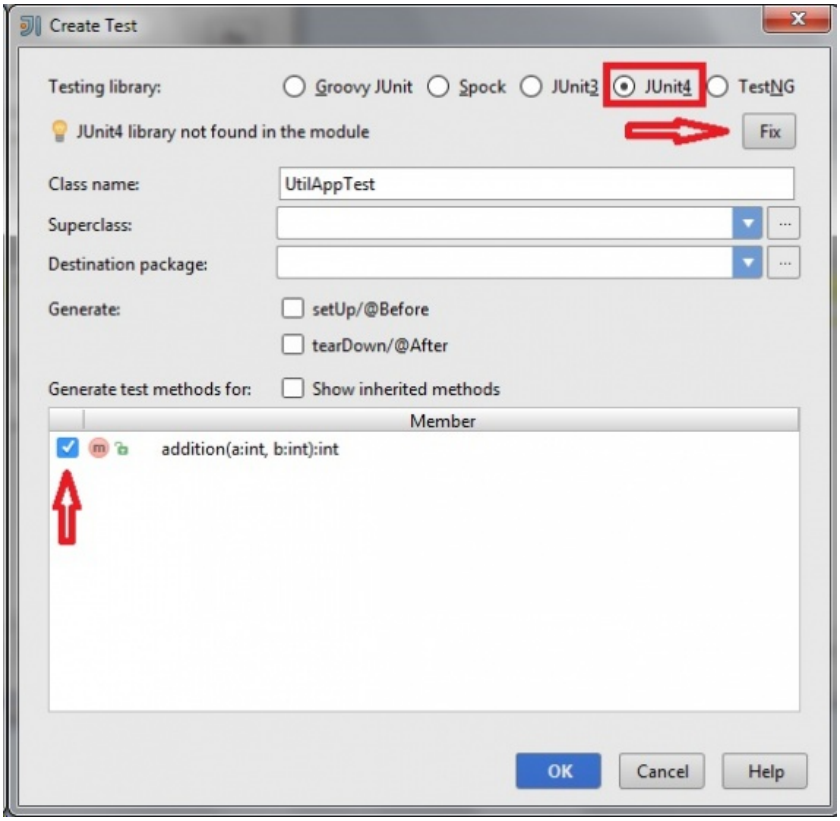


Рис. 1: Создание теста и подключение JUnit

Далее остается только наполнить тело метода. Внутри тестового метода пишутся «утверждения» — специальные методы предоставляемые JUnit. Упрощенно их работа выглядит следующим образом: в метод .assert* передается ожидаемый результат и результат вызова тестируемого метода, для удобства первым параметром можно добавить поясняющее сообщение. Если во время выполнения теста параметры не совпадут, Вы будете проинформированы об этом. Запускать тестовый класс на выполнение можно как и обычный класс.

Минимальный JUnit-тест имеет вид:

```java
import org.junit.*; // Импорт всех основных классов и аннотаций JUnit

import static org.junit.Assert.*; // Импорт утверждений

// Тестовый сценарий
public class JUnitTestCase {

    // Отдельный тест
    @Test
    public void TestName() {
        //  Проверка утверждений
    }

}
```

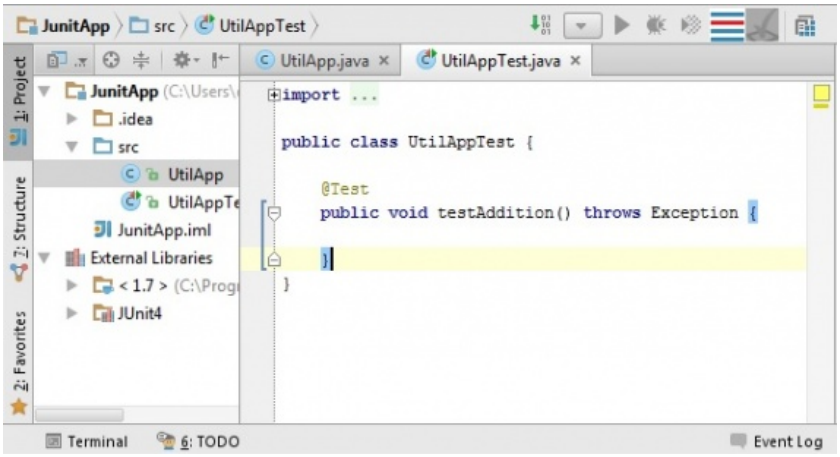


Рис. 2:

## Терминология JUnit

Модель тестирования JUnit

- **Тест в JUnit** — это `public` и *не* `static` метод, помеченный аннотацией `@Test` (`@org.junit.Test`), содержащий одно или несколько утверждений и/или предложений о том, как должен вести себя тестируемый модуль.
- **Утверждение** (*Assertion*) — логические предположения, которые должны быть истинными в тесте. Если хотя бы одно из утверждений нарушается, тест считается проваленным.
- **Тестовый класс, набор тестов** (Test Case) — класс, содержащий один и более тестовых методов (помеченных аннотацией `@Test`). Тестовые классы используются для группировки тестов, тестирующих поведение одного модуля. Обычно один класс/метод имеет соответствующий ему тестовый класс.

- **Тестовый набор** (*Test Suite*) — группа тестов. Тестовый набор используется для группировки связанных тестов в единый набор, запускаемый одним сеансом. Например, нужно запустить все тесты, покрывающие функциональность некоторого пакета.

## Основные утверждения в JUnit

### Утверждения истинности

```
// Проверяет, что результат логического выражения вернул true
assertTrue(<boolean expression>);
```

### Утверждения ложности

```
// Проверяет, что результат логического выражения вернул false
assertFalse(<boolean expression>);
```

### Утверждение эквивалентности

```
assertEquals(expected, actual);
```

### Утверждение неэквивалентности

```
assertNotEquals(expected, actual);
```

### Утверждение эквивалентности массивов

```
assertArrayEquals(expecteds[], actuals[]);
```

Где:

- `expected` — ожидаемое значение тестируемого значения.
- `actual` — действительное значение тестируемого значения.

Важное замечание: сравниваемыми аргументами в `assertEquals` могут быть и массивы. Их сравнение выполняется поэлементно.

### Null утверждения

```
// Проверяет, что объект равен null
assertNull(Object);

// Проверяет, что объект не равен null
assertNotNull(Object);
```

### Утверждения идентичности

```
// Проверяет, что оба объекта идентичны друг другу
assertSame(expected, actual);

// Проверяет, что оба объекта не идентичны друг другу
assertNotSame(expected, actual);
```

## Объявление тестов

### Тестирование метода

Чтобы превратить метод в тест, достаточно добавить аннотацию `@Test` из пакета `org.junit`:

```
@Test
public void someTest() {
    ...
}
```

### Тестирование исключений

Если надо проконтролировать появление ожидаемых исключений:

```
@Test(expected = SomeException.class)
public void someTest() {
    ...
}
```

**Тестирование по времени выполнения** Если необходимо проконтролировать время выполнения теста (время указывается в миллисекундах):

```
@Test(timeout = 100)
public void someTest() {
    ...
}
```

### Игнорирование тестов

Чтобы пропустить тест:

```
@Ignore
@Test
public void someTest() {
    ...
}
```

## Возможные исходы теста

Тест завершается одним из следующих **исходов** (*outcome*):

- **Успешно** (*Success*) — все предположения выполнились, все утверждения пройдены успешно, неожиданных исключений не выброшено.
- **Провал** (*Failure*) — одно из утверждений внутри теста не выполнилось.
- **Завершился с ошибкой** (*Error*) — выполнение теста завершилось преждевременной по причине некоторой ошибки или исключения. Итоговый успех/провал теста не известен.
- **Пропущен, проигнорирован** (*Ignore*) — тест был проигнорирован, или не выполнилось одно из предположений, необходимых для выполнения теста.

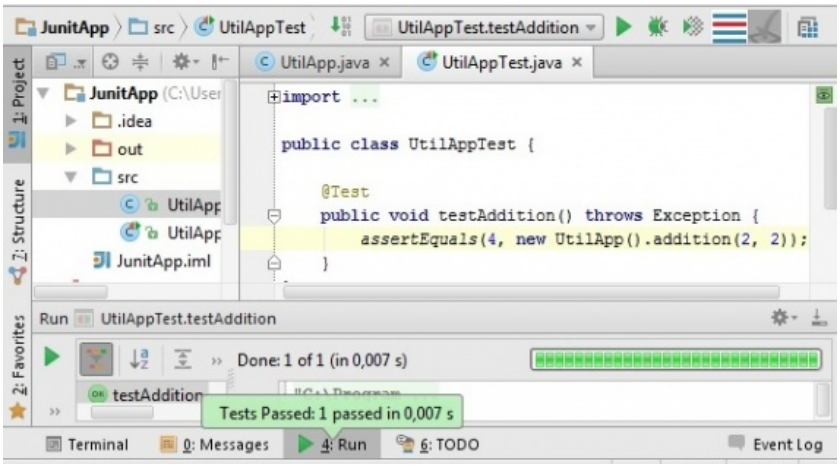


Рис. 3: Пример выполнения теста

## Фикстуры

Сценарии тестов могут нуждаться в некоторых действиях, предшествующих тесту(наполнение БД) или выполняющихся после него(очистка БД). Для этого существуют так называемые фикстуры.

### Выполнение кода до и после каждого теста

Чтобы определить код, который должен выполнятся перед каждым тестом из сценария:

```
@Before
public void setUpBeforTest() {
    ...
}
```

Код, выполняемый после исполнения каждого теста из сценария:

```
@After
public void setUpBeforTest() {
    ...
}
```

## Группа тестов (Test Suite)

**Тестовый набор** (*Test Suite*) — группа тестов, запускаемая в одном сеансе. Например, нужно запустить все тесты, покрывающие функциональность некоторого пакета.

- Что бы создать тестовый набор в JUnit выберите некоторый класс, или создайте новый пустой класс.
- Пометьте класс аннотацией `@RunWith(Suite.class)`, которая указывает библиотеке JUnit, что данный класс стоит обрабатывать как тестовый набор.
- Пометьте класс аннотацией `@SuiteClasses({ TestClass1.class, ... })` в которой в фигурных скобках, через запятую перечислите все классы, входящие в данный тестовый набор.

Пусть, есть тестовые классы `WcFileTest`, `WcDirTest`, `WcUnicodeTest`, `WcUnicodeTest`. Тогда, объединение их в тестовый набор будет иметь вид:

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;


@RunWith(Suite.class) // Запустить класс как тестовый набор
@SuiteClasses({       // Список тестовых классов в наборе для запуска
    WcFileTest.class,
    WcDirTest.class,
    WcUnicodeTest.class,
    WcUnicodeTest.class
})
public class RunAll {
    // the class remains empty,
    // used only as a holder for the above annotations
}
```

# 3 Задание 1 — Тестирование процедур

**Подготовка к выполнению:**

- Создаете новый проект в Eclipse.
- Создаете класс `StringUtils` в котором будут находится ваши функции по варианту.
- Разбираетесь как писать тесты под JUnit.

## 3.1 Выполнение задания

**Часть 1 — Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.**

В первом задании Вам дана спецификация метода — описание того, как данный метод должен работать. Ваша задача состоит в том, что бы *написать тесты для метода согласно спецификации* (до написания кода метода), а затем корректно реализовать метод таким образом, что бы он соответствовал описанной спецификации и проходил все тесты.

Последовательность выполнения работы следующая:

- Создаете заглушку вашей функции по варианту в классе `StringUtils` (пустую реализацию метода, возвращающую бессмысленное значение).
- Создаете тестовый класс `StringUtilsTest` в котором будут располагаться тесты для вашей функции. При создании набора тестов указывайте используемую версию библиотеки как **JUnit 4.x**.
- Пишете тесты согласно спецификации указанной в задании **до** того как начать реализацию функции. Написав тесты запустите и убедитесь, что они все проваливаются.
- Добавляете несколько новых *проверочных тестов*, которых нет в спецификации. Помечаете их аннотацией `@Ignore`.
- Пишите реализацию вашей функции таким образом, что бы она начала соответствовать поставленным тестам.
- В итоге, вы получаете реализацию функции, которая проходит все базовые тесты.
- Убираете аннотацию `@Ignore` с дополнительных тестов и проверяете реализацию функции на **всех** тестах.

### Часть 2 — Самостоятельно разработать спецификацию метода

Во втором задании вам необходимо самостоятельно разработать спецификацию функции. Реализовать данную спецификацию в виде тестов, а затем корректно реализовать функцию.

Последовательность выполнения работы следующая:

- Создаете заглушку вашей функции по варианту в классе `StringUtils` (пустую реализацию метода, возвращающую бессмысленное значение).
- Для вашей функции создаете новый (второй) тестовый класс.
- Самостоятельно разрабатываете спецификацию функции и реализуете её в виде тестов.
- Написав тесты запустите и убедитесь, что они все проваливаются.
- Добавляете несколько новых проверочных тестов, которых нет в спецификации. Помечаете их аннотацией `@Ignore`.
- Пишите реализацию вашей функции таким образом, что бы она начала соответствовать поставленным тестам.
- В итоге, вы получаете реализацию функции, которая проходит все базовые тесты.
- Убираете аннотацию `@Ignore` с дополнительных тестов и проверяете реализацию функции на **всех** тестах.

### Часть 3 — Реализация тестового набора (одинаково для всех вариантов)

В двух предыдущих заданиях Вы написали два тестовых класса. В данном задании необходимо реализовать **тестовый набор** (Test Suite), запускающий оба тестовых сценария, полученных на предыдущих заданиях.

### Советы

- Возможно, во время реализации обоих функций вам понадобятся еще и вспомогательные функции.

## Вариант 1

### Часть 1. Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Реализуйте и протестируйте метод `String repeat(String patthern, int repeat)`, который строит строку из указанного паттерна повторённого заданное количество раз.

Спецификация метода:

```
repeat(”e”, 0)   = ””
repeat(”e”, 3)   = ”eee”
repeat(”ABC”, 2) = ”ABCABC”
repeat(’e’, -2)  = IllegalArgumentException
```

### Часть 2. Самостоятельно разработайте спецификацию метода, реализуйте её в тестах, а затем реализуйте сам метод.

Разработайте метод `String repeat(String str, String separator, int repeat)`, который строит строку из указанного паттерна повторённого заданное количество раз, вставляя строку-разделитель при каждом повторении.

## Вариант 2

### Часть 1. Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Напишите метод `String keep(String str, String pattern)` который оставляет в первой строки все символы, которые есть так же во второй.

```
keep(null, null)        = NullPointerException
keep(null, *)        = null
keep(””, *)          = ””
keep(*, null)        = ””
keep(*, ””)          = ””
keep(”hello”, ”hl”)  = ”hll”
keep(”hello”, ”le”)  = ”ell”
```

### Часть 2. Самостоятельно разработайте спецификацию метода, реализуйте её в тестах, а затем реализуйте сам метод.

Реализуйте функцию `String loose(String str, String remove)`, удаляющую из первой строки все символы, которые есть так же во второй.

## Вариант 3

### Часть 1. Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Реализуйте и протестируйте метод `int indexOfDifference(String str1, String str2)` который сравнивает две строки и возвращает индекс той позиции, в которой они различаются. Например, `indexOfDifference("i am a machine", "i am a robot")` должно вернуть 7.

```
StringUtils.indexOfDifference(null, null) = NullPointerException
StringUtils.indexOfDifference("", "") = -1
StringUtils.indexOfDifference("", "abc") = 0
StringUtils.indexOfDifference("abc", "") = 0
StringUtils.indexOfDifference("abc", "abc") = -1
StringUtils.indexOfDifference("ab", "abxyz") = 2
StringUtils.indexOfDifference("abcde", "abxyz") = 2
StringUtils.indexOfDifference("abcde", "xyz") = 0
```

### Часть 2. Самостоятельно разработайте спецификацию метода, реализуйте её в тестах, а затем реализуйте сам метод.

Напишите метод `String difference(String str1, String str2)` сравнивающий две строки и возвращающий разницу между ними. Другими словами, метод должен остаток от строки, где они различаются.

## 3.2 Вариант 4

### Часть 1. Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Напишите метод `String substringBetween(String str, String open, String close)` выделяющий подстроку относительно открывающей и закрывающей строки.

```
substringBetween(null, null, null)    = NullPointerException
substringBetween(null, *, *)          = null
substringBetween(*, null, *)          = null
substringBetween(*, *, null)          = null
substringBetween("", "", "")          = ""
substringBetween("", "", "]")         = null
substringBetween("", "[", "]")        = null
substringBetween("yabcz", "", "")     = ""
substringBetween("yabcz", "y", "z")   = "abc"
substringBetween("yabczyabcz", "y", "z")   = "abc"
substringBetween("wx[b]yz", "[", "]") = "b"
```

### Часть 2. Самостоятельно разработайте спецификацию метода, реализуйте её в тестах, а затем реализуйте сам метод.

Реализуйте функцию `String overlay(String str, String overlay, int start, int end)`, которая перекрывает содержимое первой строки содержимым второй строки.

- `str` — первая строка, оригинал.
- `overlay` — вторая строка, содержимое которой накладывается на первую.
- `start` — позиция в оригинальной строке, с которой начинается перекрытие
- `end` — позиция в оригинальной строке, перед которой закончить перекрытие.

## Вариант 5

### Часть 1. Написать тесты к методу, а затем реализовать сам метод по заданной спецификации.

Напишите метод `int levenshteinDistance(String s, String t)` рассчитывающий расстояние Ливенштейна для двух строк. Расстояние Ливенштейна между двумя строками — это то количество посимвольных трансформаций необходимое, что бы превратить одну строку в другую.

```
levenshteinDistance(null, null)          = NullPointerException
levenshteinDistance(null, *)             = -1
levenshteinDistance(*, null)             = -1
levenshteinDistance("","")               = 0
levenshteinDistance("","a")              = 1
levenshteinDistance("aaapppp", "")       = 7
levenshteinDistance("frog", "fog")       = 1
levenshteinDistance("fly", "ant")        = 3
levenshteinDistance("elephant", "hippo") = 7
levenshteinDistance("hippo", "elephant") = 7
levenshteinDistance("hippo", "zzzzzzzz") = 8
levenshteinDistance("hello", "hallo")    = 1
```

### Часть 2. Самостоятельно разработайте спецификацию метода, реализуйте её в тестах, а затем реализуйте сам метод.

Реализуйте метод `int hamingDistance(String str1, String str2)`, определяющий расстояние Хэминга для двух строк. Дистанция Хэминга — это число позиций, в которых соответствующие символы двух слов одинаковой длины различны.

# 4 Задание 2 — Поиск ошибок, отладка и тестирование классов

## 4.1 Введение — Структуры данных

**Стек** (*Stack*) – структура данных реализующая принцип "последним пришел, первым вышел" (LIFO = *"Last In, First Out"*). Поддерживает следующие операции:

- `push` — вставляет элемент в начало стека.
- `pop` — удаляет с вершины стека последний добавленный элемент.

**Очередь** (*Queue*) — структура данных реализующая принцип "первым пришел, первым вышел" (FIFO = "*First In, First Out*"). Имеет следующие операции:

- `enqueue` = вставляет элемент в начало очереди.
- `deque` = извлекает элемент из конца очереди.

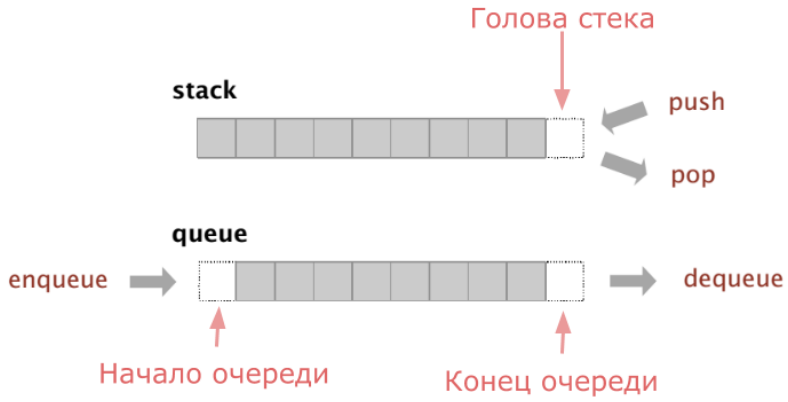


Рис. 4: Стек и очередь одинаковы при добавлении, но отличаются, при удалении элементов.

## 4.2 Выполнение задания

### 4.2.1 Часть 1 — Импорт проекта

Импортируйте один из проектов по варианту:

- **Stack** — проект содержит реализацию стека на основе связного списка: `Stack.java`.
- **Queue** — содержит реализацию очереди на основе связного списка: `Queue.java`

Разберитесь как реализована ваша структура данных.

Каждый проект содержит:

- Клиент для работы со структурой данных и правильности ввода данных реализации (см. метод main()).
- `TODO`-декларации, указывающие на нереализованные методы и функциональность.
- `FIXME`-декларации, указывающую на необходимые исправления.
- Ошибки компиляции (Синтаксические)
- Баги в коде (!).
- Метод `check()` для проверки целостности работы класса.

### 4.2.2 Часть 2 — Поиск ошибок

- Исправить синтаксические ошибки в коде.
- Разобраться в том, как работает код, подумать о том, как он должен работать и найти допущенные баги.

### 4.2.3 Часть 3 — Внутренняя корректность

- Разобраться что такое утверждения (assertions) в коде и как они включаются в Java.
- Заставить ваш класс работать вместе с включенным методом `check`.
- Выполнить клиент (метод main() класса) передавая данные в структуру используя включенные проверки (assertions).

### 4.2.4 Часть 4 — Реализация функциональности

- Реализовать пропущенную функции в классе.
- См. документацию перед методом относительно того, что он должен делать и какие исключения выбрасывать.
- Добавить и реализовать функцию очистки состояния структуры данных.

### 4.2.5 Часть 5 — Написание тестов

- Все фукции вашего класса должны быть покрыты тестами.
- Использовать фикстуры для инициализации начального состояния объекта.
- Итого, должно быть несколько тестовых классов, в каждом из которых целевая структура данных создается в фикстуре в некотором инициализированном состоянии (пустая, заполненная и тд), а после очищается.
- Написать тестовый набор, запускающий все тесты.