Введение в программирование на Java

Лекция 11. Верификация ПО. Модульное тестирование.

Виталий Олегович Афанасьев

14 апреля 2025

Михаил Романович Шура-Бура (1918-2008):

Аксиома 1. *В каждой (даже отлаженной) программе есть хотя бы одна ошибка.*

Аксиома 2. *Если ошибки нет в программе, она есть в используемом алгоритме.*

Аксиома 3. *Если ошибки нет и в алгоритме, такая программа никому не нужна.*

Примеры ошибок в ПО

- Ariane-5 (1996)
- Mars Climate Orbiter (1999)
- Therac-25 (1985-1987)
- Intel Pentium (1994)
- UBS, облигации Capcom (2009)

Мифы о безопасном ПО: уроки знаменитых катастроф

Размеры современного ПО

- Windows NT 3.1 (1993) — 5 миллионов строк кода
- Windows 10 (2015) — 55 миллионов строк кода
- Все сервисы Google на момент 2015 года — 2 миллиарда строк кода

Визуализация кол-ва строк кода в разном ПО

Верификация и валидация

Верификация — проверка, что ПО соответствует поставленным требованиям.

Валидация — проверка, что ПО удовлетворяет нуждам и потребностям пользователей.

Характеристики качества ПО

Разные стандарты определяют качество ПО как набор некоторых атрибутов
(ISO/IEC 9126, ISO/IEC 25010, ГОСТ Р ИСО/МЭК 9126-93 и другие).

Основные из них:

- **Функциональность**
- **Надёжность**
- **Защищённость**
- **Переносимость**
- **Совместимость**
- **Производительность**
- **Удобство использования**
- **Удобство сопровождения**

Методы верификации ПО

- **Экспертиза**
- **Статический анализ**
- **Формальные методы**
- **Динамические методы**
  - **Тестирование**

Тестирование

Тестирование — метод верификации, при котором система или компонент системы проверяется на соответствие требованиям на конечном наборе ситуаций.

Например, какие ситуации необходимо проверить для калькулятора?

Тестирование

Тестирование — метод верификации, при котором система или компонент системы проверяется на соответствие требованиям на конечном наборе ситуаций.

Например, какие ситуации необходимо проверить для калькулятора?

- `1 + 2` $\longrightarrow$ 3
- `1 - 2` $\longrightarrow$ -1
- `2 * 3` $\longrightarrow$ 6
- `10 / 2` $\longrightarrow$ 5
- `10 / 0` $\longrightarrow$ Ошибка

Виды тестирования по масштабу проверяемых элементов

- **Модульное тестирование** — проверка отдельных компонентов (модулей) системы.
- **Интеграционное тестирование** — проверка взаимодействия нескольких компонентов системы.
- **Системное тестирование** — проверка работы системы в целом.
  - Альфа-тестирование
  - Бета-тестирование

Модульное тестирование

При модульном тестировании проверяются **единицы тестирования**.

Единица тестирования (Unit Under Test) — любой кусок кода, который мы хотим проверить отдельно.

Обычно, UUT — это набор операторов, метод или даже целый класс.

Модульное тестирование зачастую подразумевает выполнение UUT и сравнения результата с эталоном. Такое сравнение называет **assertion**.

JUnit

Библиотека JUnit представляет возможности модульного тестирования для Java, автоматизируя рутинные процессы:

- Запуск тестов
- Выдача результатов тестирования
- (Иногда) Генерация тестовых входных данных

От программиста при этом требуется разработать набор тестовых случаев (**test cases**), содержащих набор утверждений (**assertions**), сравнивающих поведение кода с эталонным.

JUnit: Пример 1

Для примера, протестируем метод abs из класса Math.

```
1  public class MathAbsTest {
2      @Test
3      public void testAbs_positive() {
4          assertEquals(10, Math.abs(10));
5      }
6
7      @Test
8      public void testAbs_negative() {
9          assertEquals(5, Math.abs(-5));
10     }
11
12     @Test
13     public void testAbs_zero() {
14         assertEquals(0, Math.abs(0));
15     }
16 }
```

JUnit: Пример 2

В качестве следующего примера, протестируем целый набор операций класса `IntStack`.

```
1  public class IntStackTest {
2      @Test
3      public void testPushPop_poppedInReverseOrder() {
4          IntStack stack = new IntStack();
5
6          stack.push(1);
7          stack.push(2);
8          int top = stack.pop();
9          int bottom = stack.pop();
10
11         assertEquals(2, top);
12         assertEquals(1, bottom);
13     }
14 }
```

Общие соглашение о Unit-тестах

Unit-тесты должны быть:

- Быстрыми.
- Независимыми.
- Повторяемыми/воспроизводимыми.
- Целенаправленными.
- Читаемыми.

Преимущества модульного тестирования

Помимо нахождения ошибок, модульное тестирование полезно и по ряду других причин:

- Отслеживание регрессий.
- Документация кода.

Проблема полноты тестирования (1)

Вопрос: сколько тестов нужно написать, чтобы **полностью** протестировать метод ниже?

```
1 public class Math {
2     public static byte abs(byte value) {
3         ...
4     }
5 }
```

Проблема полноты тестирования (2)

А сколько тестов нужно написать, чтобы **полностью** протестировать данный метод?

```
1 public class Math {
2     public static short sum(byte left, byte right) {
3         ...
4     }
5 }
```

Проблема полноты тестирования (3)

А сколько тестов нужно написать, чтобы **полностью** протестировать все операции в этом классе?

```
1  public class IntStack {
2      private final int[] values;
3      private final int size;
4
5      public void push(int value) {
6          ...
7      }
8      public int pop() {
9          ...
10     }
11 }
```

Проблема полноты тестирования (4)

Эдсгер Дейкстра:

> *Тестирование программы может вполне эффективно служить для демонстрации* ***наличия в ней ошибок****, но, к сожалению,* ***непригодно для доказательства их отсутствия****.*

Проблема полноты тестирования (5)

Полное тестирование даже для маленьких программ не осуществимо.

Составлять тестовые сценарии необходимо на основе конкретного ПО и того, какие риски и ошибки более важны.

Для составления тестового набора используют разные техники:

- Тестирование граничных значений.
- Сценарное тестирование.
- Вероятностное тестирование.
- . . . и другие.

Способы оценки полноты тестирования

Для оценки того, насколько полны тесты, используют разные эвристики:

- Покрытие строк.
- Покрытие инструкций.
- Покрытие ветвей.
- Покрытие условий.
- Покрытие путей.
- . . . и другие.

Важно: никакой из способов оценки не покажет, всё ли протестировано в системе.

Покрытие ветвей VS Покрытие условий

```
1 if (user.isAuthenticated() || user.name().contains("admin")) {
2     showSecretData();
3 }
```

При покрытии условий считаются все комбинации операндов в логическом выражении. При двух операндах таких комбинаций 4, при трёх — восемь, и т.д.

При покрытии ветвей оператор ветвления считается покрытым, если есть хотя бы два теста — один, при котором ветвь выполняется, и один, при котором не выполняется.

Тестирование приватных методов

Библиотеки для тестирования обычно позволяют тестировать только публичные методы. Но что делать с методами, которые не видны публично?

Точки зрения:

1. Приватные методы не являются единицами тестирования. Тестироваться должны только публичные члены модулей без нарушения инкапсуляции. Приватные методы тестируются опосредованно.
2. Приватный доступ — лишь инструмент для разработки, но тестирование может такой доступ игнорировать (например, используя вспомогательные test-only методы или Reflection API).

Виды тестирования по уровню доступа к исходной системе

- Тестирование "чёрного ящика" (функциональное тестирование) — тестирование без знания исходного кода.
- Тестирование "белого ящика" (структурное тестирование) — тестирование со знанием исходного кода.
- Тестирование "серого ящика".