

МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(ФГБОУ ВО «ВГУ»)

Факультет прикладной математики, информатики и механики
Кафедра программного обеспечения
и администрирования информационных систем

Анализ существующих подходов к тестированию JVM приложений

Магистерская диссертация
Направление 02.03.03. Математическое обеспечение и администрирование
информационных систем
Профиль Информационные системы и базы данных

Зав. кафедрой _____ д. ф-м. н. проф. М. А. Артёмов _____.____2021 г.
Обучающийся _____ А. С. Пахомов
Руководитель _____ д. ф-м. н. проф. М. А. Артёмов

Воронеж 2021

Аннотация

Аннотация – краткое содержание работы, отражающее ее особенности. В тексте аннотации могут быть представлены: цель работы, метод исследования и полученные результаты, их область применения и внедрения. Изложение материала в аннотации должно быть кратким и точным. Рекомендуемый объем аннотации 500–1000 печатных знаков.

Содержание

Введение	4
Глава 1. Анализ существующих подходов к тестированию	5
1.1. Введение в тестирование программного обеспечения	5
1.1.1. Ручное тестирование	5
1.1.2. Автоматизированное тестирование	6
1.2. Подходы к написанию автоматизированных тестов	7
1.2.1. Тестирование на основе спецификации	10
1.2.2. Тестирование границ	12
1.2.3. Структурное тестирование	15
1.2.4. Тестирование на основе модели	23
1.2.5. Тестирование на основе контракта	25
1.2.6. Тестирование свойств	29
1.3. Продвинутое тестирование автоматизации тестирования	29
1.3.1. Статическое тестирование	30
1.3.2. Мутационное тестирование	31
1.3.3. Генерация псевдослучайных входных данных	33
1.3.4. Тестирование на основе анализа кода	35
1.4. Лексическая генерация случайных входных данных	36
1.4.1. Fuzzing: Breaking Things with Random Inputs	36
1.4.2. Code Coverage	36
1.4.3. Mutation-Based Fuzzing	36
1.4.4. Greybox Fuzzing	36
1.4.5. Search-Based Fuzzing	36
1.4.6. Mutation Analysis	36
1.5. Синтаксическая генерация случайных входных данных	36
1.5.1. Fuzzing with Grammars	36
1.5.2. Efficient Grammar Fuzzing	36
1.5.3. Grammar Coverage	37
1.5.4. Parsing Inputs	37
1.5.5. Probabilistic Grammar Fuzzing	37
1.5.6. Fuzzing with Generators	37
1.5.7. Greybox Fuzzing with Grammars	37
1.5.8. Reducing Failure-Inducing Inputs	37
1.6. Семантическая генерация случайных входных данных	37

1.6.1. Mining Input Grammarss	37
1.6.2. Tracking Information Flow	37
1.6.3. Concolic Fuzzing	37
1.6.4. Symbolic Fuzzing	37
1.6.5. Mining Function Specifications	37
1.7. Доменная генерация случайных входных данных	37
1.7.1. Testing Configurations	37
1.7.2. Fuzzing APIs	38
1.7.3. Carving Unit Tests	38
1.7.4. Testing Web Applications	38
1.7.5. Testing Graphical User Interfaces	38
Глава 2. Постановка задачи	39
Глава 3. Реализация	40
3.1. Средства реализации	40
3.2. Требования к программному и аппаратному обеспечению	40
3.3. Реализация	40
3.4. План тестирования	40
Заключение	41
Список литературы	42
Приложение А. Листинг кода	43

Введение

Введение содержит в сжатой форме положения, обоснованию которых посвящена магистерская диссертация: актуальность выбранной темы; степень её разработанности; цель и содержание поставленных задач; объект и предмет исследования; методы исследования; научная новизна (при наличии), практическая значимость. Обоснованию актуальности выбранной темы предшествует краткое описание проблемной ситуации.

Глава 1. Анализ существующих подходов к тестированию

Первая глава формируется на основе изучения имеющейся отечественной и зарубежной научной и специальной литературы по исследуемой теме (с обязательными ссылками на источники!), а также нормативных материалов. В ней содержится описание объекта и предмета исследования посредством различных теоретических концепций, принятых понятий и их классификации, а также степени проработанности проблемы в России и за ее пределами. Автор должен продемонстрировать глубину погружения в проблему, владение знаниями о текущем состоянии ее решения путем анализа максимально возможного количества источников. В редкой ситуации полной новизны, тем не менее, необходимо проанализировать состояние выбранной предметной области с последующими выводами об актуальности заявленных исследований. В первой главе могут рассматриваться существующие подходы к решению задач исследования, проводиться их сравнительный анализ с использованием системы критериев. Результаты анализа могут быть представлены в виде таблиц, графиков, диаграмм, схем для того, чтобы сделать выводы о сильных и слабых сторонах имеющихся решений и обосновать собственные предложения и подходы. Кроме того, может быть предложен собственный понятийный аппарат (при необходимости). Первая глава, по сути, служит теоретическим обоснованием исследований, проведенных автором. Последующие главы магистерской диссертации строятся по схеме: математическое, алгоритмическое, программное обеспечение.

1.1. Введение в тестирование программного обеспечения

Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом [1]. Существует множество техник и подходов к тестированию программного обеспечения.

1.1.1. Ручное тестирование

Ручное тестирование (англ. manual testing) — часть процесса тестирования на этапе контроля качества в процессе разработки ПО. Оно

производится тестировщиком без использования программных средств, для проверки программы или сайта путём моделирования действий пользователя [2].

Приемущества такого способа тестирования:

- Простота. От тестировщика не требуется знания специальных инструментов атоматизации.
- Тестируется именно то, что видит пользователь.

Основные проблемы ручного тестирования:

- Наличие человеческого труда. Тестирующих может допустить ошибку в процессе ручных действий.
- Выполнение ручных действий может занимать много времени.
- Такой вид тестирования не способен покрыть все сценарии использования ПО.
- Не исключается повторное внесение ошибки. Если пользователь системы нашел ошибку, тестировщик воспроизведет её только один раз. В последующих циклах разработки программного обеспечения ошибка может быть внесена повторно.

1.1.2. Автоматизированное тестирование

Автоматизированное тестирование программного обеспечения — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно использует программные средства для выполнения тестов и проверки результатов выполнения [3].

Подходы к автоматизации тестирования:

- Тестирование пользовательского интерфейса. С помощью специальных тестовых библиотек производится имитация действий пользователя.
- Тестирование на уровне кода (модульное тестирование).

Приемущества атоматизированного тестирования:

- сокращение времени тестирования;
- уменьшение вероятности допустить ошибку по сравнению с ручным тестированием;
- исключение появления ошибки в последующей разработке программного обеспечения.

Недостатки атоматизированного тестирования:

- Трудоемкость. Поддержка и обновление тестов являются трудоемким процессом.
- Необходимость знания инструментария.
- Автоматическое тестирование не может полностью заменить ручное. На практике используется комбинация ручного и автоматизированного тестирования.

Существует множество инструментов для написания и запуска тестов на языке Java: JUnit, Spock Framework, TestNG, UniTESK, JBehave, Serenity, Selenide, Gauge, Geb.

JUnit

JUnit — самый распространенный инструмент для написания и запуска тестов на языке Java. Последняя версия 5.7.1 [4].

Сценарий использования JUnit 5:

1. Определить тестируемый класс или модуль. Листинг 1.1.
2. Создать новый класс, для написания тестов. По соглашению, имя класса должно совпадать с именем тестируемого класса и заканчиваться постфиксом *Test*. Листинг 1.2.
3. Для каждого тестового сценария необходимо написать метод и пометить его аннотацией *@org.junit.jupiter.api.Test*.
4. В каждом сценарии нужно написать соответствующий код, который заканчивается выражением из пакета *org.junit.jupiter.api.Assertions.**.
5. Запустить тест в среде разработки (IDE) или с помощью системы сборки (Gradle, Maven).

При разработке тестовых сценариев главным изменяющимся компонентом являются входные данные. Структура теста меняется реже. Что бы избежать дублирования кода, JUnit 5 предоставляет возможность писать *параметризованные тесты*. Чтобы заменить обычный тест параметризованным, достаточно поменять аннотацию *@Test* на *@ParametrizedTest* и указать источник входных данных. Пример параметризованного теста представлен в листинге 1.3.

1.2. Подходы к написанию автоматизированных тестов

Процесс тестирования программного обеспечения можно разделить на две фазы: разработка тестовых сценариев и запуск тестовых сценариев.

Листинг 1.1 Тестируемый класс *RomanNumeral*

```
public class RomanNumeral {  
    private static Map<Character, Integer> map;  
  
    static {  
        map = new HashMap<>();  
        map.put('I', 1);  
        map.put('V', 5);  
        map.put('X', 10);  
        map.put('L', 50);  
        map.put('C', 100);  
        map.put('D', 500);  
        map.put('M', 1000);  
    }  
  
    public int convert(String s) {  
        int convertedNumber = 0;  
  
        for (int i = 0; i < s.length(); i++) {  
            int currentNumber = map.get(s.charAt(i));  
            int next = i + 1 < s.length() ? map.get(s.charAt(i + 1))  
            : 0;  
  
            if (currentNumber >= next) {  
                convertedNumber += currentNumber;  
            } else {  
                convertedNumber -= currentNumber;  
            }  
        }  
  
        return convertedNumber;  
    }  
}
```

Разработка тестовых сценариев подразумевает анализ, дизайн и написание кода. Смысл этой фазы состоит в том, что бы разработать такое множество тестовых сценариев, которое бы удовлетворяло стандартам качества разрабатываемого программного обеспечения. Понятие «автоматизированное тестирование» не включает в себя автоматизацию этой фазы.

Листинг 1.2 Тестирующий класс *RomanNumeralTest*

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

public class RomanNumeralTest {

    @Test
    void convertSingleDigit() {
        RomanNumeral roman = new RomanNumeral();
        int result = roman.convert("C");

        assertEquals(100, result);
    }

    @Test
    void convertNumberWithDifferentDigits() {
        RomanNumeral roman = new RomanNumeral();
        int result = roman.convert("CCXVI");

        assertEquals(216, result);
    }

    @Test
    void convertNumberWithSubtractiveNotation() {
        RomanNumeral roman = new RomanNumeral();
        int result = roman.convert("XL");

        assertEquals(40, result);
    }
}
```

Вторая фаза подразумевает инсталляцию программного обеспечения и выполнение тестовых сценариев, разработанных на первой фазе. Запуск тестовых сценариев чаще всего автоматизируется.

Разработка тестовых сценариев — комплексная задача. Сложность ее состоит в нахождении достаточного набора тестовых сценариев, который удовлетворяет стандартам качества. Одновременно с этим, набор тестов должен быть конечен и выполняться достаточно быстро. Далее будут рассмотрены техники анализа, дизайна и написания тестовых сценариев.

Листинг 1.3 Пример параметризованного теста

```

@ParameterizedTest(name = "small={0}, big={1}, total={2},
    result={3}")
@CsvSource({
    "1,1,5,0", "1,1,6,1", "1,1,7,-1", "1,1,8,-1",
    "4,2,3,3", "3,2,3,3", "2,2,3,-1", "1,2,3,-1"
})
void boundaries(int small, int big, int total, int
    expectedResult) {
    int result = new ChocolateBars().calculate(small, big, total
    );
    Assertions.assertEquals(expectedResult, result);
}

```

1.2.1. Тестирование на основе спецификации

Спецификация — набор требований к программному обеспечению. Может быть представлена как текстовый файл или UML диаграмма.

Тестирование на основе спецификации — подход к разработке минимального набора тестов, которые будут удовлетворять спецификации. Такой подход позволяет абстрагироваться от конкретной реализации системы (тестирование «черного ящика»).

Основу этого метода составляет группировка множества входных данных.

Группировка множества входных данных

Пример спецификации. Определение високосного года. На вход программе поступает год в виде числа, программа должна возвращать `true` если год является високосным и `false` в противном случае. Год високосный, если:

- год кратен 4;
- год не кратен 100;
- исключение: если год кратен 400, то он високосный.

Реализация спецификации представлена в листинге 1.4.

Для подбора оптимальных входных данных, нужно разбить программу на классы (группы). Другими словами, нужно разбить множество входных данных следующим образом:

Листинг 1.4 Определение високосного года

```
public class LeapYear {  
  
    public boolean isLeapYear(int year) {  
        if (year % 400 == 0)  
            return true;  
        if (year % 100 == 0)  
            return false;  
  
        return year % 4 == 0;  
    }  
}
```

1. каждый класс уникален, т. е. не существует двух классов, которые приводят к одному и тому же поведению программы;
2. поведение программы может быть однозначно интерпритированно как корректное или некорректное.

Учитывая требования к классам и спецификацию, можно получить следующий набор классов:

- год кратен 4, но не кратен 100 — високосный, `true`;
- год кратен 4, кратен 100, кратен 400 — високосный, `true`;
- год не кратен 4 — не високосный, `false`;
- год кратен 4, кратен 100, но не кратен 400 — не високосный, `false`.

Каждый класс может быть выражен в бесконечном множестве входных данных. Однако, каждый конкретный набор входных данных из одного и того же класса приводит к одному и тому же поведению программы. Таким образом, классы образуют *классы эквивалентности*. Достаточно выбрать один набор входных данных из каждого класса:

- 2016, год кратен 4, но не кратен 100;
- 2000, год кратен 4, кратен 100, кратен 400;
- 39, год не кратен 4;
- 1900, год кратен 4, кратен 100, но не кратен 400.

Пример тестирующего кода представлен в листинге 1.5.

Листинг 1.5 Тестирующий класс *LeapYearTest*

```

public class LeapYearTest {

    private final LeapYear leapYear = new LeapYear();

    @Test
    public void divisibleBy4_notDivisibleBy100() {
        boolean leap = leapYear.isLeapYear(2016);
        assertTrue(leap);
    }

    @Test
    public void divisibleBy4_100_400() {
        boolean leap = leapYear.isLeapYear(2000);
        assertTrue(leap);
    }

    @Test
    public void notDivisibleBy4() {
        boolean leap = leapYear.isLeapYear(39);
        assertFalse(leap);
    }

    @Test
    public void divisibleBy4_and_100_not_400() {
        boolean leap = leapYear.isLeapYear(1900);
        assertFalse(leap);
    }
}

```

1.2.2. Тестирование границ

Ошибки, основанные на граничных условиях, очень распространены. Например, разработчики часто ошибаются в операторах «больше» (>) или «больше или равно» (>=). Техника тестирования границ позволяет избежать подобных ошибок.

Границы между классами

В предыдущем разделе описан подход к написанию тестов с помощью классов эквивалентности. Эти классы имеют границы. Другими словами,

если применять маленькие изменения к входным данным (например, +1) рано или поздно набор входных данных перейдет в другой класс. Конкретная точка в которой входные данные переходят из одного класса в другой, называется *граничным значением*. Суть тестирования границ — тестирование корректности программы на граничных значениях.

Более формально, граничные значения — это два набора ближайших к друг другу входных данных $[p_1, p_2]$, где p_1 относится к группе A , а p_2 относится к группе B .

На практике тестирование границ комбинируется с тестированием, основанным на спецификации. Такая комбинация называется *доменным тестированием*.

Пример тестирования границ

Постановка задачи. Подсчет количества очков игрока. Даны очки игрока и количество оставшихся жизней, программа должна:

- Если количество очков игрока меньше 50, то всегда добавлять 50 очков к текущему значению.
- Если количество очков игрока больше или равно 50, то:
 - Если количество оставшихся жизней больше, чем 3, то умножить очки игрока на 3.
 - Иначе добавить 30 очков к текущему значению.

Реализация поставленной задачи представлена в листинге 1.6.

Листинг 1.6 Подсчет количества очков игрока

```
public class PlayerPoints {

    public int totalPoints(int currentPoints, int remainingLives
    ) {
        if(currentPoints < 50)
            return currentPoints+50;

        return remainingLives < 3 ? currentPoints+30 :
        currentPoints*3;
    }
}
```

Разбитие входных данных на классы выглядит следующим образом:

1. Количество очков < 50 .
2. Количество очков ≥ 50 и оставшихся жизней < 3 .
3. Количество очков ≥ 50 и оставшихся жизней ≥ 3 .

Тестирующий код представлен в листинге 1.7.

Листинг 1.7 Тестирующий код

```
public class PlayerPointsTest {

    private final PlayerPoints pp = new PlayerPoints();

    @Test
    void lessPoints() {
        assertEquals(30+50, pp.totalPoints(30, 5));
    }

    @Test
    void manyPointsButLittleLives() {
        assertEquals(300+30, pp.totalPoints(300, 1));
    }

    @Test
    void manyPointsAndManyLives() {
        assertEquals(500*3, pp.totalPoints(500, 10));
    }
}
```

Определение граничных значений:

1. **Граничное значение 1:** Когда количество очков строго меньше чем 50, набор входных данных относится к группе 1. Если количество очков больше или равно 50, набор входных данных относится к группам 2 и 3. Таким образом, граничные значения равны 49 и 50.
2. **Граничное значение 2:** Когда количество очков больше или равно 50 и количество оставшихся жизней меньше чем 3, тогда набор данных относится к группе 2, иначе он относится к группе 3.

Получившиеся границы представлены на рис. 1.1.

Тестирующий код представлен в листинге 1.8.

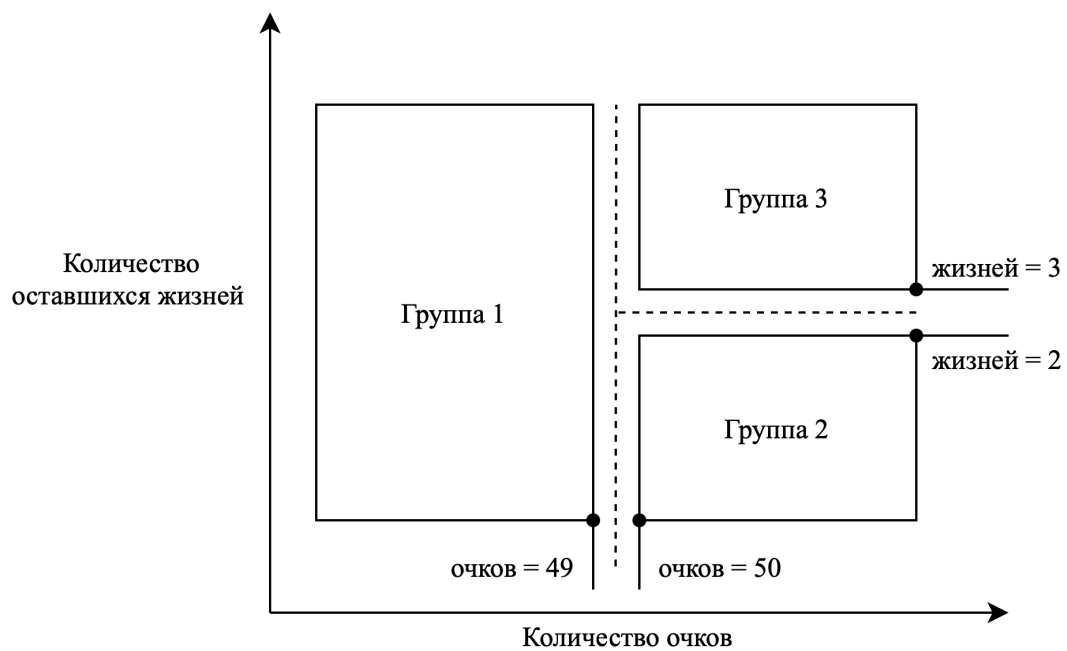


Рис. 1.1. Границы групп

Листинг 1.8 Тестирование границ

```
@Test
void betweenLessAndManyPoints() {
    assertEquals(49+50, pp.totalPoints(49, 5));
    assertEquals(50*3, pp.totalPoints(50, 5));
}

@Test
void betweenLessAndManyLives() {
    assertEquals(500*3, pp.totalPoints(500, 3));
    assertEquals(500+30, pp.totalPoints(500, 2));
}
```

1.2.3. Структурное тестирование

В предыдущих разделах были рассмотрены подходы к разработке тестовых сценариев, которые основаны исключительно на спецификации программы. Далее будут рассмотрены подходы, которые учитывают исходный код программы. Техники, которые используют исходный код программы для разработки тестов, называются техниками **структурного тестирования**.

Основу структурного тестирования составляет *критерий покрытия*. Критерий покрытия тесно связан с понятием *покрытия тестами*. Под

покрытием тестами подразумевается процент всего исходного кода программы, который выполняется в ходе работы тестов.

Преимущества структурного тестирования:

- Позволяет разработать тесты исходя только из исходного кода программы.
- Четко определяет критерий полноты тестирования. Это может быть 90 % (в крайних случаях 100 %).

Далее будут рассмотрены следующие критерии покрытия:

- покрытие строк;
- покрытие блоков;
- покрытие ветвлений;
- покрытие условий;
- покрытие путей исполнения;
- MC/DC покрытие.

Критерий покрытия строк

Покрытие строк (англ. line coverage) — критерий покрытия, основанный на подсчете исполненных в ходе выполнения тестов строк кода. Рассчитывается этот критерий как процентное соотношение исполненных в ходе выполнения тестов строк кода к общему числу строк кода.

$$\text{покрытие строк} = \frac{\text{количество исполненных строк}}{\text{общее количество строк}} \times 100\%$$

Для демонстрации подсчета этого критерия рассмотрим следующий пример. Программа принимает на вход два числа — количество очков первого игрока и количество очков второго игрока. Программа должна возвращать количество очков победителя. Победителем становится игрок, набравший максимально близкое к 21 количество очков. Если игрок набрал больше очков чем 21, он проигрывает. Если оба игрока проиграли, программа должна вернуть 0. Реализация представлена в листинге 1.9. Тестирующий код представлен в листинге 1.10.

Первый тест выполняет строки 1-7 и 10. Покрытие этого теста составляет 90 %.

$$\frac{9}{10} \times 100\% = 90\%$$

Листинг 1.9 Реализация программы Black Jack

```

public class BlackJack {
    public int play(int left, int right) {
        1. int ln = left;
        2. int rn = right;
        3. if (ln > 21)
        4.     ln = 0;
        5. if (rn > 21)
        6.     rn = 0;
        7. if (ln > rn)
        8.     return ln;
        9. else
        10.    return rn;
    }
}

```

Листинг 1.10 Тестирующий код

```

public class BlackJackTest {
    @Test
    void bothPlayersGoTooHigh() {
        int result = new BlackJack().play(30, 30);
        assertThat(result).isEqualTo(0);
    }

    @Test
    void leftPlayerWins() {
        int result = new BlackJack().play(10, 9);
        assertThat(result).isEqualTo(10);
    }
}

```

Не исполненной остается только строка 8. Второй тест её исполняет. Таким образом, покрытие теста BlackJackTest составляет 100 %

Главной проблемой критерия покрытия строк является то, что этот критерий не всегда отражает реальное покрытие всех возможных сценариев выполнения программы. В листинге 1.11 представлена другая реализация поставленной ранее задачи.

Листинг 1.11 Компактная реализация программы Black Jack

```

public int play(int left, int right) {
    1. int ln = left;
    2. int rn = right;
    3. if (ln > 21) ln = 0;
    4. if (rn > 21) rn = 0;
    5. if (ln > rn) return ln;
    6. else return rn;
}

```

В таком случае покрытие теста `BlackJackTest.leftPlayerWins` составляет 83 %. Однако тот же самый тест показывает покрытие в 60 % в первой реализации `BlackJack`. Покрытие строк зависит не только от тестового сценария и набора входных данных, но и от конкретного стиля написания кода.

Критерий покрытия блоков

Граф потока управления — представление всех путей исполнения кода. Он состоит из *базовых блоков, управляющих блоков и рёбер*. Пример графа управления для программы `BlackJack` из листинга 1.9 представлен на рис. 1.2.

Приемущество графа выполнения состоит в том, что он практически не зависит от языка программирования, на котором написанна программа.

Покрытие блоков — критерий покрытия, основанный на подсчете соотношения задействованных в ходе выполнения тестов блоков к общему числу блоков в графе потока управления программы.

$$\text{покрытие блоков} = \frac{\text{количество задействованных блоков}}{\text{общее количество блоков}} \times 100\%$$

Критерий покрытия блоков более устойчив к изменениям форматирования исходного кода, чем критерий покрытия строк.

Критерий покрытия ветвлений

Покрытие ветвлений — критерий покрытия, основанный на подсчете соотношения задействованных в ходе выполнения программы блоков и ребер

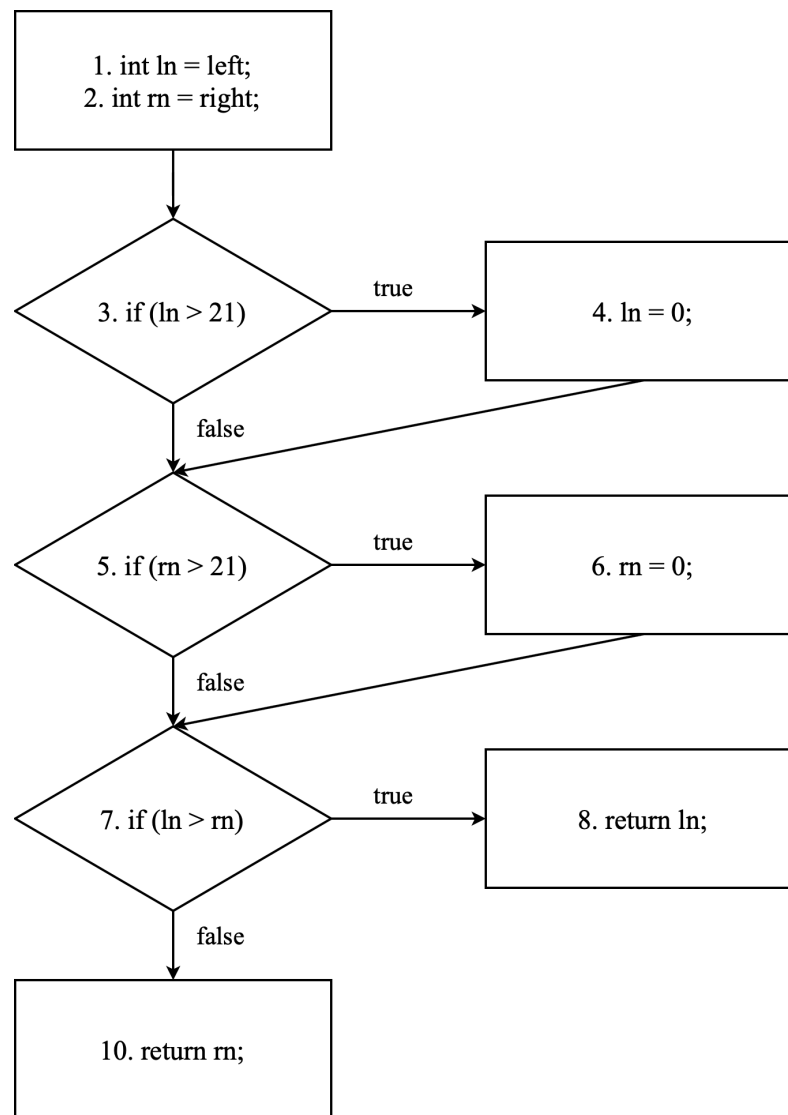


Рис. 1.2. Граф потока управления для программы BlackJack

к общему числу блоков и ребер в графе управления программы. По сути, это расширенный критерий порывтия блоков.

Комбинация из ребра и блока составляет *решение* или *ветвь*. На основе решиний или ветвей и строится критерий покрытия ветвлений.

$$\text{покрытие ветвлений} = \frac{\text{количество задействованных ветвей}}{\text{общее количество ветвей}} \times 100\%$$

На практике критерий покрытия ветвлений оказывается предпочтительнее. Он отражает более четкую картину. Например, на графе потока управления в один блок могут приводить несколько ребер. Это означает, что при использовании покрытия блоков, достаточно выполнить блок всего один раз, чтобы достичь 100 % покрытия. В то время как одно из ребер не будет выполнено. Покрытие ветвлений решает эту проблему.

Критерий покрытия условий

Покрытие ветвлений предоставляет два состояния для каждого управляющего блока (условие выполнено или нет). В тех случаях, когда условие внутри управляющего блока состоит из более чем одного логического оператора, покрытия ветвлений недостаточно.

Например, $a > 10 \ \&\& \ b < 20 \ \&\& \ c < 10$. Чтобы достичь 100 % покрытия по критерию ветвлений, достаточно подобрать два набора входных параметров: $(a=20, b=10, c=5) - \text{true}$ и $(a=5, b=10, c=5) - \text{false}$. Однако, эти наборы вводных данных не покрывают все логические комбинации. Например: $(a=20, b=30, c=5) - \text{false}$.

Базовый критерий покрытия условий решает эту проблему. Его суть состоит в трансформации исходного графа выполнения в граф выполнения, который не содержит составных логических условий.

$$\text{покрытие условий} = \frac{\text{количество задействованных условий}}{\text{общее количество условий}} \times 100\%$$

Одного покрытия условий бывает недостаточно. 100% покрытие условий не гарантирует 100 % покрытия всех возможных путей выполнения программы. На практике используется C/DC покрытие (Conditions/Decisions coverage). Оно включает в себя покрытие условий и покрытие ветвлений.

$$\text{C/DC} = \frac{P_1 + P_2}{M_1 + M_2} \times 100\%$$

Где P_1 — количество задействованных условий, P_2 — количество задействованных ветвей, M_1 — общее количество условий, M_2 — общее количество ветвей.

Критерий покрытия путей исполнения

C/DC критерий приводит к большому количеству возможных тестовых сценариев. Это происходит потому, что для разработки тестовых сценариев используются все возможные исходы всех логических условий, а так же всех блоков выполнения.

Критерий покрытия путей исполнения сосредоточен на подсчете уникальных путей, вместо подсчета результатов блоков выполнения и условных блоков. На графе выполнения программы путём является уникальный маршрут из блока А в блок В.

$$\text{покрытие путей исполнения} = \frac{\text{количество задействованных путей}}{\text{общее количество путей}} \times 100\%$$

MC/DC покрытие

MC/DC покрытие (Modified C/DC) очень похоже на покрытие путей исполнения. Отличие состоит в том, что MC/DC учитывает не все возможные пути выполнения, а только «важные». В результате общее количество тестов сокращается.

Ключевая идея за MD/CD: выполнить каждое условие таким образом, чтобы оно изменило результат программы, независимо от других условий. Рассмотрим пример условного оператора в листинге 1.12.

Листинг 1.12 Пример условного оператора

```
if (!Character.isLetter(str.charAt(i))
& (last == 's' | last == 'r')) {
    words++;
}
```

Его можно интерпритировать как $(A \ \& \ (B \ | \ C))$ следующим образом:

- $A = !\text{Character.isLetter}(\text{str.charAt}(i));$
- $B = \text{last} == 's';$
- $C = \text{last} == 'r'.$

Чтобы достичь 100 % покрытия, используя критерий покрытия путей, нужно написать 8 тестовых сценариев (2^8). MC/DC критерий позволяет сократить число тестовых сценариев до 4. Проблема состоит в том, что бы выбрать из 8 возможных тестовых сценариев необходимые 4. Суть подбора заключается в следующем: каждый следующий тестовый сценарий должен изменять результат работы программы (по сравнению с предыдущим тестовым сценарием), при этом набор входных параметров должен отличаться не более, чем на один параметр. Алгоритм подбора тестовых данных для программы из листинга 1.11:

- $(A = \text{true}, B = \text{false}, C = \text{true}) = \text{true}.$ Изменяем параметр C.
- $(A = \text{true}, B = \text{false}, C = \text{false}) = \text{false}.$ Изменяем параметр B.

- $(A = \text{true}, B = \text{true}, C = \text{false}) = \text{true}$. Изменяем параметр A.
- $(A = \text{false}, B = \text{true}, C = \text{false}) = \text{false}$.

Приемуществом MC/DC критерия является меньшее количество наборов тестовых данных ($N + 1$, N - количество входных параметров) по сравнению с критерием покрытия путей (2^N , N - количество входных параметров).

Выбор критерия покрытия

Приведенные критерии покрытия имеют разные свойства. Некоторые из них просты и интуитивны. Другие покрывают максимально возможное количество путей исполнения программы, но обладают сложностью в реализации. На рис 1.3 показано отношение критериев с точки зрения полноты покрытия.

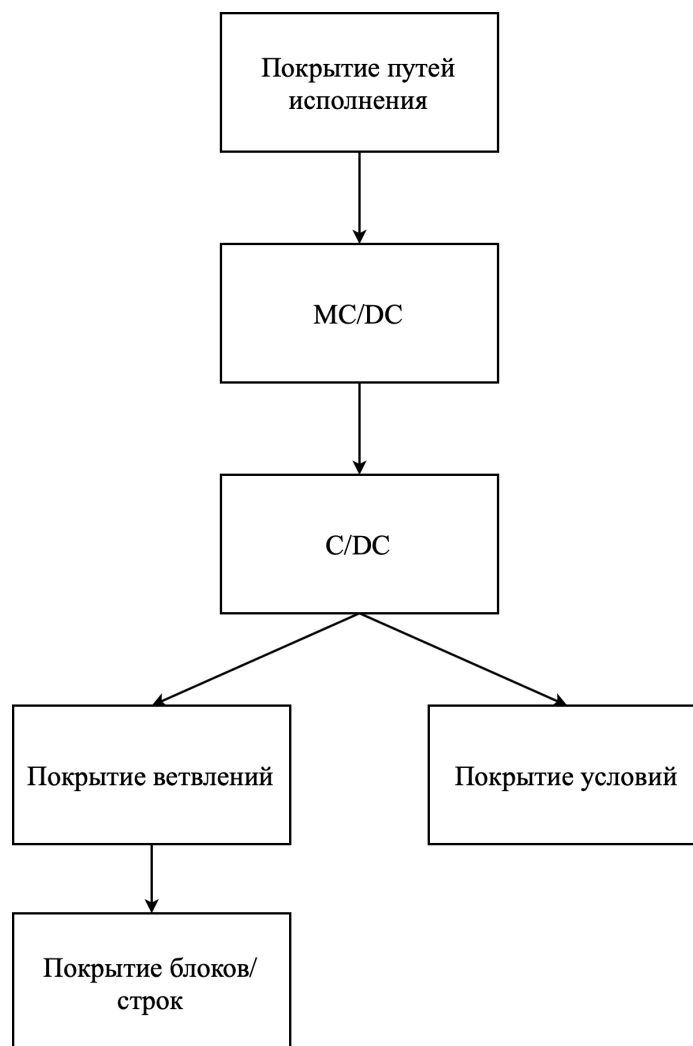


Рис. 1.3. Соотношение критериев покрытия

Например, 100 % покрытие ветвлений автоматически означает 100 % покрытия строк кода. Но 100 % покрытия MC/DC не означает 100 % покрытия путей исполнения.

На практике критерий покрытия строк кода является самым быстрым и простым. Если требуется более детальное покрытие, выбирается MC/DC.

В больших программах сложно добиться 100 % покрытия. Часто это не выгодно с точки зрения затраченных человеческих ресурсов. Рекомендованный порог покрытия 90 % [5].

Важно отметить, что структурное тестирование, не зависимо от выбранного критерия покрытия, не способно полностью заменить тестирование на основе спецификации. Структурное тестирование хорошо дополняет тестирование на основе спецификации.

1.2.4. Тестирование на основе модели

Модель — способ формализации работы тестируемой системы. Модель используется для анализа и тестирования программного обеспечения. Далее будут рассмотрены два вида моделей: *таблицы решений* и *конечные автоматы*.

Таблицы решений

Таблица решений — таблица, состоящая из *условий* и *действий*, которые выполняет программа при указанных условиях.

Таблицы решений используются для моделирования того, как комбинация входных параметров приводит к определённому результату выполнения программы.

Пример таблицы решений представлен в табл. 1.1.

Таблица 1.1. Пример таблицы решений

		Варианты			
<i>Условия</i>	<Условие1>	T	T	F	F
	<Условие2>	T	F	T	F
<i>Действие</i>	<Действие>	значение1	значение2	значение3	значение4

Таблица содержит все комбинации условий (2^N). На практике, условия, не влияющие на результат, помечаются как dc (don't care) и не участвуют в комбинировании входных данных.

Существует несколько стратегий написания тестовых сценариев, основанных на таблице решений:

- Все варианты, представленные в таблице. Каждая колонка в таблице — отдельный набор тестовых сценариев.
- Все возможные варианты. Каждый набор входных данных — уникальная комбинация. Количество таких комбинаций 2^N . На практике это почти недостижимый способ разработки тестовых сценариев.
- Учет уникальных действий. Каждый тестовый сценарий приводит к уникальному действию.
- Каждое условие является истинным и ложным. Каждое условие в таблице решений должно принять оба своих состояния: `true` и `false`. Чаще всего это приводит к двум наборам входных данных: все условия истинны и все условия ложны.
- MC/DC. Применение техники MC/DC к таблице решений.

Для написания сценариев тестирования, основанных на таблице решений, часто используются параметризованные тесты JUnit 5.

Конечные автоматы

Модель конечных автоматов описывает *состояния* системы и *переходы* между ними. Состояние отражает конкретное место в коде в момент выполнения. Переход — действие, которое переводит систему из одного состояния в другое. Помимо состояний и переходов, конечный автомат имеет *начальное состояние* и *события*. Начальное состояние — состояние, с которого начинается работа системы. События — описание того, что происходит в процессе перехода.

Конечные автоматы представляются в виде UML диаграммы. Пример конечного автомата представлен на рис. 1.4

Для разработки тестовых сценариев на основе конечного автомата, необходимо преобразовать конечный автомат в *дерево переходов*. Пример дерева переходов представлен на рис. 1.5

Из дерева переходов можно разработать набор тестовых сценариев. Техника подбора тестовых сценариев проста: нужно посетить все листья дерева переходов. Такой подход близок к структурному тестированию с критерием покрытия путей исполнения.

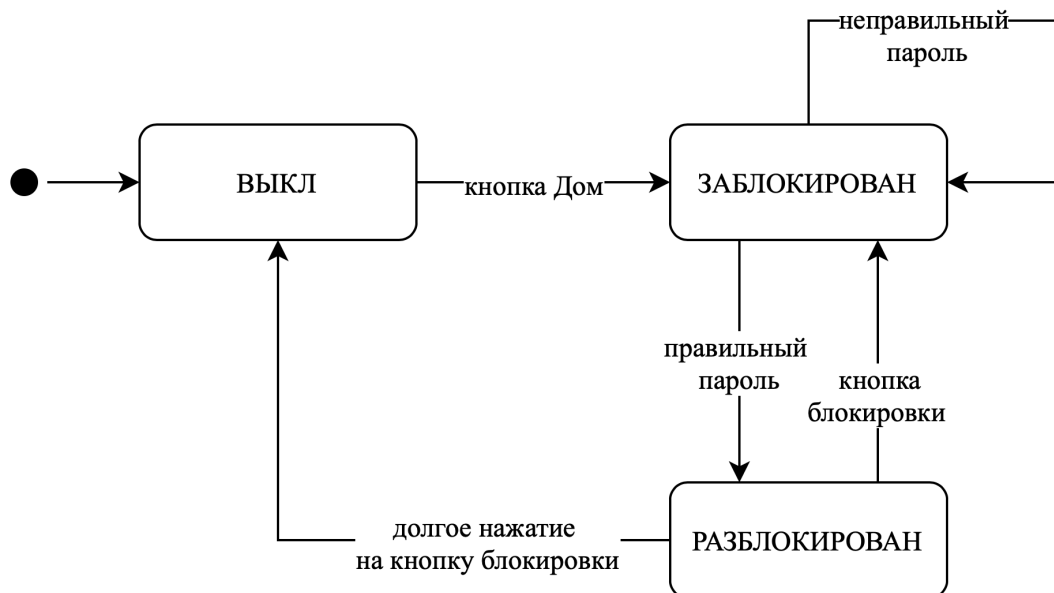


Рис. 1.4. Конечный автомат главного экрана телефона

Описанный подход позволяет формализовать разработку тестов. Однако, для разработки понятных и поддерживаемых тестов на основе конечного автомата часто требуются дополнительные усилия. Например, введение нового уровня абстракции в кодовую базу тестов (интроспекция состояний и переход из одного состояния в другое).

1.2.5. Тестирование на основе контракта

Самотестируемые программы

Программа, которая тестирует свое поведение в ходе исполнения, называется самотестируемой. До настоящего момента были рассмотрены подходы к тестированию программного обеспечения, которые подразумевают написание отдельного кода с тестами. Самотестируемые программы исключают написание специального кода для тестов. Они содержат тесты внутри себя. Далее будут рассмотрены способы реализации самотестируемых программ.

Утверждения

Самый простой способ проверить состояние программы на соответствие какому-либо условию — утверждения (assertions). Пример программы с утверждениями представлен в листинге 1.13.

Утверждения играют роль *оракулов*. Они информируют программиста в случае, если программа работает некорректно.

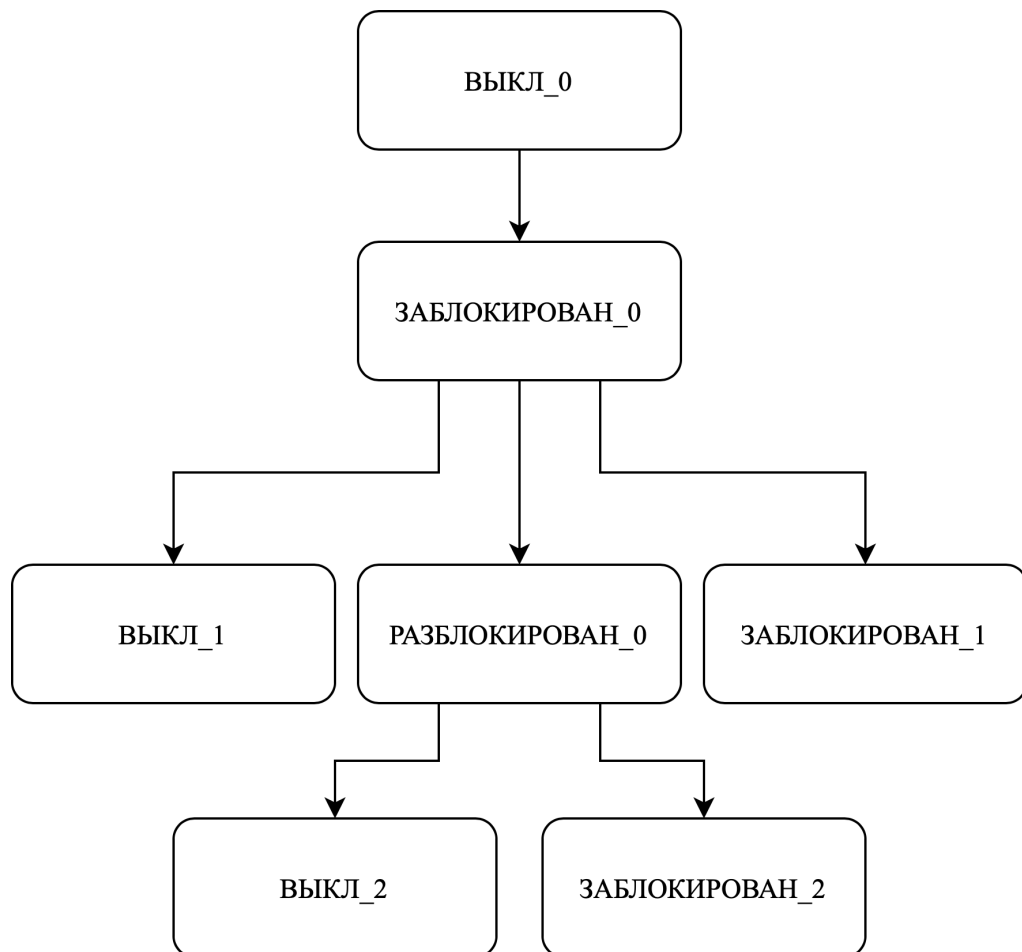


Рис. 1.5. Дерево переходов главного экрана телефона

Листинг 1.13 Пример утверждения

```

public class MyStack {
    public Element pop() {
        assert count() > 0 : "The stack does not have any elements
        to pop";

        // ... actual method body ...

        assert count() == oldCount - 1 : "Size of stack did not
        decrease by one";
    }
}

```

Стоит отметить, что утверждения не заменяют модульное тестирование.

Предусловия и постусловия

Понятия предусловия и постусловия определяются в логике Хоара, а именно **тройками Хоара** [6]. Тройка Хоара выглядит следующим образом:

$$\{P\}C\{Q\}$$

где P и Q являются утверждениями, а C — командой (выражение, метод). P — предусловие, Q — постусловие. Можно утверждать, что если выполнено предусловие P , то команда C будет исполнена. Если предусловие P не выполнено, то команда C не может быть выполнена. Если команда C выполнена, то гарантируется соблюдение постусловия Q .

Пример программы с предусловиями и постусловиями представлен в листинге 1.14.

Листинг 1.14 Программа с постусловиями и предусловиями

```
public class FavoriteBooks {
    // ...

    public void merge(List<Book> books) {
        assert books != null : "The list of books is null";
        assert favorites != null : "The favorites list is null";

        List<Book> newBooks = books.removeAll(favorites);

        if (!newBooks.isEmpty()) {
            favorites.addAll(newBooks);
            pushNotification.booksAdded(newBooks);
        }

        assert favorites.containsAll(books) : "Not all books were
        added to favorites";
    }
}
```

Важное замечание: если не было выполнено предусловие, то метод не может гарантировать соблюдение постусловия.

Инварианты

Инвариант программы — условие, которое выполняется на протяжении всего времени жизни программы. Отличие инварианта от

предусловия или постусловия в том, что предусловие выполняется только до выполнения метода, а постусловие только после. Инвариант выполняется всегда.

Для проверки инварианта, как правило, создается проверяющий метод. Пример проверяющего метода представлен в листинге 1.15. Он проверяет, что дочерние элементы двоичного дерева всегда содержат указатель на родительский элемент.

Листинг 1.15 Пример проверяющего метода

```
public void checkRep(BinaryTree tree) {
    BinaryTree left = tree.getLeft();
    BinaryTree right = tree.getRight();

    assert (left == null || left.getParent() == tree) &&
        (right == null || right.getParent() == tree) :
        "A child does not point to the correct parent";

    if (left != null) {
        checkRep(left);
    }
    if (right != null) {
        checkRep(right);
    }
}
```

Инварианты могут быть не только на уровне метода. В объектно-ориентированных языках часто встречаются **инварианты классов**. Инвариант класса — условие, которое выполняется на протяжении всей жизни экземпляра класса. Такие инварианты проверяются в конструкторе, вначале и конце всех публичных методов.

Дизайн по контракту

Дизайн по контракту — дизайн системы, которая соблюдает предусловия, постусловия и инварианты. В клиент-серверных приложениях дизайн по контракту играет значимую роль. Для успешной коммуникации клиента с сервером, клиент должен соблюдать предусловия сервера. А сервер гарантирует соблюдение постусловий и инвариантов. На практике

контрактом является интерфейс. Диаграмма на рис. 1.6 иллюстрирует контракт, объявленный интерфейсом и его имплементацию.

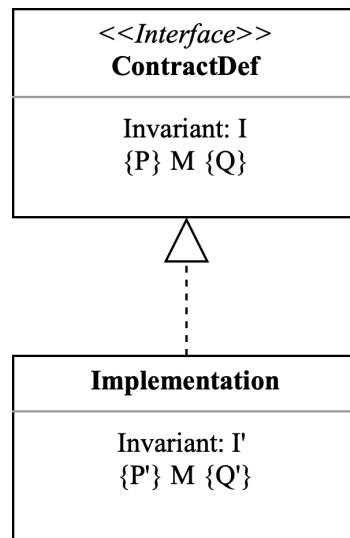


Рис. 1.6. Контракт и его реализация

1.2.6. Тестирование свойств

Тестирование свойств (Property-based testing) — генерация псевдо-случайных наборов входных данных с целью поиска ошибок в программе. Первая реализация этой идеи называется *QuickCheck* [7], написанная для языка Haskell. Сейчас многие языки имеют аналоги данного инструмента. Тестирование свойств в Java производится с помощью библиотеки jqwik [8].

Для того, что бы использовать jqwik, необходимо определить метод и пометить его аннотацией `@Property` (аналог аннотации `@Test`). После этого нужно добавить в аннотированный метод параметры и пометить их аннотацией `@ForAll` (есть множество других аннотаций).

Для примера определим свойство конкатенации двух строк: для любых двух строк результат их конкатенации должен иметь длину, равную сумме длин исходных строк. Пример тестирования свойства с помощью jqwik представлен в листинге 1.16.

Jqwik сгенерирует множество псевдо-случайных входных данных (`s1` и `s2`) и проверит выполнение утверждения.

1.3. Продвинутое тестирование автоматизации тестирования

До настоящего момента под «автоматизацией» понималась автоматизация выполнения тестов. Продвинутое тестирование расширяет понятие автоматизации. Они исключают

Листинг 1.16 Тестирование свойства конкатенации двух строк

```
public class PropertyTest {

    @Property
    void concatenationLength(@ForAll String s1, @ForAll String
        s2) {
        String s3 = s1 + s2;

        Assertions.assertEquals(s1.length() + s2.length(), s3.
            length());
    }
}
```

человеческий фактор в процессе тестирования. Это означает отсутствие необходимости разрабатывать тестовые сценарии. Компьютер сделает это за человека. Иногда такие подходы называются тестированием с помощью искусственного интеллекта.

1.3.1. Статическое тестирование

Статическое тестирование (статический анализ) — автоматизированный процесс ревизии кода без его исполнения. Статическое тестирование позволяет быстро найти ошибки низкой и средней сложности (использование неактуальной библиотеки, обращение к пустому указателю и т. д.). Распространенные инструменты статического анализа: *PMD*, *Checkstyle*, *Checkmarx*.

Классический подход к статическому анализу заключается в проверке исходного кода системы на потенциальные структурные или стилистические уязвимости. Анализатор состоит из парсера и набора правил. Существуют две техники статического анализа:

1. Сопоставление с образцом с помощью регулярных выражений.
2. Синтаксический анализ абстрактного синтаксического дерева.

Сопоставление с образцом

Сопоставление с образцом — проверка кода, в ходе которой происходит поиск соответствия заданному ранее образцу. Образец задается с помощью

регулярного выражения. Регулярное выражение — последовательность символов, представляющая образец.

Недостатком сопоставления с образцом является отсутствие контекста исполнения. Другими словами, сопоставление с образцом не учитывает семантику программы. Например, задав регулярное выражение `\s*System.out.println\(.*\) ;` сопоставление с образцом определит 3 строки в листинге 1.17. Из чего можно сделать вывод, что программа напечатает 3 раза некоторое сообщение. Однако, программа не напечатает ни одного сообщения.

Листинг 1.17 Пример неучитанной семантики

```
boolean DEBUG = false;

if (DEBUG) {
    System.out.println("Debug line 1");
    System.out.println("Debug line 2");
    System.out.println("Debug line 3");
}
```

Синтаксический анализ

Более продвинутый способ анализа кода — синтаксический анализ. В ходе синтаксического анализа текст программы разбивается на поток символов, символы преобразуются в токены, а из токенов составляется *дерево разбора*. Дерево разбора представляет собой синтаксическое дерево конкретной версии кода. В синтаксическом анализе используется *абстрактное синтаксическое дерево* — дерево разбора без синтаксических деталей (точки с запятой, скобки). Пример абстрактного синтаксического дерева кода из листинга 1.17 представлен на рис. 1.7

Статический анализатор, использующий синтаксический анализ, принимает на вход абстрактное синтаксическое дерево и набор правил. Анализатор сообщает о найденных несоответствиях в коде.

1.3.2. Мутационное тестирование

При рассмотрении структурного тестирования были описаны различные критерии покрытия кода. Эти критерии нужны для того, чтобы измерить объем кода, выполненного в ходе тестирования. К сожалению,

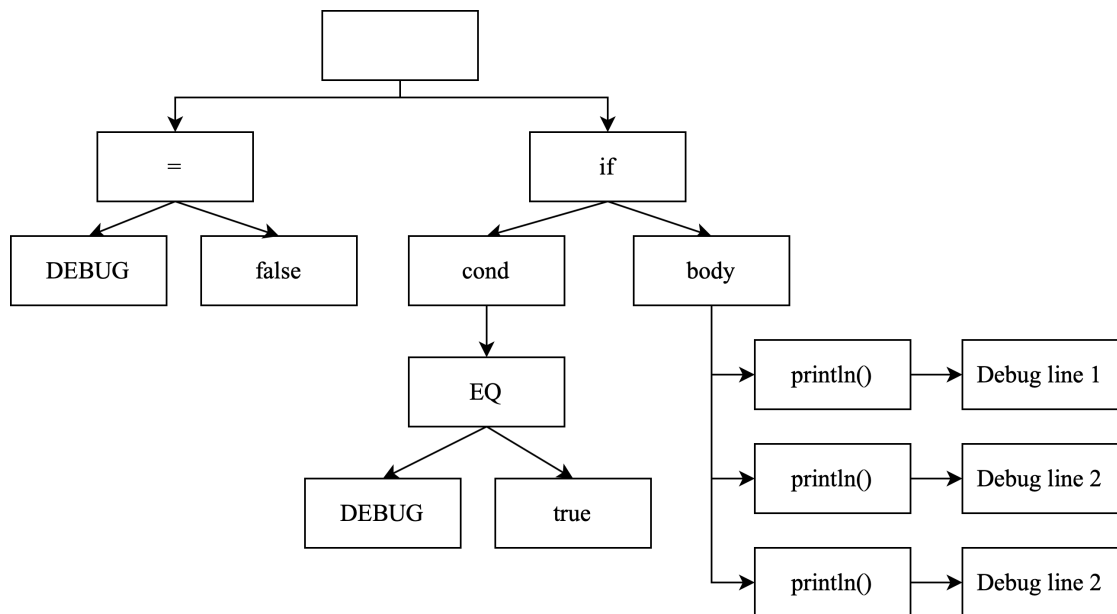


Рис. 1.7. Абстрактное синтаксическое дерево

таких критериев может быть не достаточно для оценки качества тестов. Например, тест может достигать 100 % покрытия кода, но в то же время не содержать проверок (assertions). То есть тест запускает код, но не тестирует результат его исполнения. Мутационное тестирование решает эту проблему.

Критерий, который отражает насколько эффективен тест, называется критерием *способности определения ошибок (fault detection capability)*. Этот критерий составляет основу **мутационного тестирования**. В процессе мутационного тестирования в код программы добавляются (искусственно) ошибки. После чего запускается набор тестов и проверяется, смогли ли тесты найти ошибку. Цель мутационного тестирования — повышение качества тестирующего кода. Чем больше ошибок тест может найти, тем выше его эффективность.

Терминология в мутационном тестировании:

- **Мутант.** Для данной программы P , мутантом называется программа P' , полученная из программы P путём *синтаксической трансформации*. Мутант считается убитым, если он не проходит хотя бы один тест.
- **Синтаксическая трансформация.** Небольшое изменение кода, при котором код все еще компилируется.
- **Изменение.** Имитация типичной человеческой ошибки.

Для автоматизации мутационного тестирования необходимо определить **мутационный оператор**. Мутационный оператор — грамматическое

правило, которое может быть использовано для создания синтаксической трансформации. Например, замена знака + на – в арифметических выражениях. Распространенные мутационные операторы:

- **Оператор арифметической замены.** Замена арифметического оператора на другой. Арифметические операторы: +, –, *, /, %.
- **Оператор замены сравнений.** Замена операторов <=, >=, !=, ==, >, <.
- **Оператор логической замены.** Замена операторов &&, ||, &, |, !, ^.
- **Оператор замены присваивания.** Замена операторов =, +=, -=, /=.
- **Оператор замены скалярных переменных.** Замена одной переменной на другую.

Помимо описанных выше метационных операторов, существуют операторы, ориентированные на особенности языка. Например, оператор замены модификаторов видимости, переопределение метода и т. д.

Для оценки качества тестового сценария с помощью мутационного анализа используется *критерий устойчивости к мутациям*.

$$\text{критерий устойчивости к мутациям} = \frac{\text{количество убитых мутантов}}{\text{общее количество мутантов}}$$

Чем больше мутантов убивает тест, тем выше его критерий устойчивости к мутациям. Соответственно, этот тест найдет больше реальных ошибок программистов.

Недостатком мутационного тестирования является время исполнения. На проектах средней сложности (300 файлов) мутационное тестирование занимает около 10 минут [9].

В языке Java для мутационного тестирования используется инструмент PIT [10].

1.3.3. Генерация псевдослучайных входных данных

Генерация псевдослучайных входных данных или фаззинг (англ. Fuzzing) — техника тестирования программного обеспечения, основанная на автоматической генерации псевдослучайных входных данных [11]. Цель

фаззинга — поиск *ошибок, утечек памяти, неудачных обработок ошибок и уязвимостей безопасности.*

Существует несколько способов генерации входных данных:

- **Случайная генерация.** Тестируемая система принимается за черный ящик, предположений об формате входных данных не делается. Генерируется большое количество случайных данных и подается на вход программе. На практике этот способ редко используется. Для построения более эффективных фаззеров используется *структурированные входные данные*, которые predetermined.
- **Мутирующие фаззеры.** На вход фаззеру даётся пример входных данных. Фаззер применяет различные мутации к данным и проверяет поведение системы. Мутации бывают разные: замена символов в строке, замена битов. Некоторые инструменты (American Fuzzy Lop (AFL)) используют генетические алгоритмы для повышения качества входных данных.
- **Генеративные фаззеры.** Генеративные фаззеры или *Протокольные фаззеры* принимают на вход формат входных данных. Например, если программа обрабатывает файлы в формате jpeg, то генеративный фаззер будет генерировать файлы только формата jpeg.

Мутирующие фаззеры более гибкие и проще в использовании, чем генеративные. Но генеративные фаззеры достигают большего покрытия кода и производят более эффективные тестовые сценарии.

Генерация случайных данных — затратный процесс с точки зрения процессорного времени. Для оптимизации этого процесса используется техника *техники сокращения времени генерации.*

Использование нескольких фаззеров

Самый простой способ увеличить покрытие кода — использование нескольких фаззеров одновременно. Каждый фаззер генерирует входные данные по собственному алгоритму. Это приводит к повышению качества тестовых сценариев и сокращению общего времени генерации (100 % покрытия достигается быстрее).

Использование телеметрии кода

Телеметрия кода (данные о покрытии) может быть полезна для коррекции стратегии генерации. Например, фаззер может использовать только тот набор входных данных, который повышает показатель покрытия кода и отбросить большую часть сгенерированных данных.

Символическое исполнение

Для определения набора входных параметров, необходимых для достижения определенного участка кода, используется символическое исполнение. Можно составить формулу для определенного пути, которая отвечает на вопрос: существует ли такой набор данных, который приводит к исполнению определенной строки кода? Если да, то какой.

Z3 является самым популярным инструментом для символического исполнения. На вход он получает участки кода, которые нужно достичь. Результатом работы является набор ограничений для входных данных, которые должны выполняться, чтобы достичь определенных участков кода. Результат работы Z3 учитывается генеративным или мутационным фаззером с целью оптимального подбора входных параметров.

В листинге ?? определяется функция с двумя входными параметрами a и b . Для удовлетворения предиката `else if` нужно удовлетворить ограничениям $((N + M \leq 2) \& (N < 100))$. Ограничение выводится следующим образом:

1. a и b преобразуются в символы $a = N, b = M$.
2. Операторы присвоения трансформируют a и b в $a = N + M, b = (N + M) - M = N$.
3. Ограничение оператора `if`: $(N + M > 2)$, для остальных веток исполнения это $N + M \leq 2$.
4. Ограничение ветки `else if`: $N < 100$.
5. Итоговое ограничение ветки `else if` является комбинацией двух предыдущих: $(N + M \leq 2) \& (N < 100)$

Стоит отметить, что не всегда код является разрешимым с точки зрения символического исполнения.

1.3.4. Тестирование на основе анализа кода

TBD

Листинг 1.18 Пример неучитанной семантики

```
public String func(int a, int b){  
    a = a + b;  
    b = a - b;  
    String str = "No";  
    if (a > 2)  
        str = "Yes!";  
    else if (b < 100)  
        str = "Maybe!";  
    return str;  
}
```

1.4. Лексическая генерация случайных входных данных**1.4.1. Fuzzing: Breaking Things with Random Inputs**

TBD

1.4.2. Code Coverage

TBD

1.4.3. Mutation-Based Fuzzing

TBD

1.4.4. Greybox Fuzzing

TBD

1.4.5. Search-Based Fuzzing

TBD

1.4.6. Mutation Analysis

TBD

1.5. Синтаксическая генерация случайных входных данных**1.5.1. Fuzzing with Grammars**

TBD

1.5.2. Efficient Grammar Fuzzing

TBD

1.5.3. Grammar Coverage

TBD

1.5.4. Parsing Inputs

TBD

1.5.5. Probabilistic Grammar Fuzzing

TBD

1.5.6. Fuzzing with Generators

TBD

1.5.7. Greybox Fuzzing with Grammars

TBD

1.5.8. Reducing Failure-Inducing Inputs

TBD

1.6. Семантическая генерация случайных входных данных

1.6.1. Mining Input Grammarss

TBD

1.6.2. Tracking Information Flow

TBD

1.6.3. Concolic Fuzzing

TBD

1.6.4. Symbolic Fuzzing

TBD

1.6.5. Mining Function Specifications

TBD

1.7. Доменная генерация случайных входных данных

1.7.1. Testing Configurations

TBD

1.7.2. Fuzzing APIs

TBD

1.7.3. Carving Unit Tests

TBD

1.7.4. Testing Web Applications

TBD

1.7.5. Testing Graphical User Interfaces

TBD

Глава 2. Постановка задачи

Во второй главе приводится постановка задачи, ее содержательное и формализованное описание. Например, если работа связана с разработкой информационных систем и использованием информационных технологий, в содержательной постановке приводятся ссылки на документы, регламентирующие процесс функционирования информационной системы, основные показатели, которые должны быть достигнуты в условиях эксплуатации информационной системы; ограничения на время решения поставленной задачи, сроки выдачи информации, способы организации диалога человека с информационной системой средствами имеющегося инструментария, описание входной и выходной информации (форма представления сообщений, описание структурных единиц, периодичность выдачи информации или частота поступления), требования к организации сбора и передачи входной информации, ее контроль и корректировка. В математической постановке (при наличии) выполняется формализация задачи, в результате которой определяется состав переменных, констант, их классификация, виды ограничений на переменные и математические зависимости между переменными. Устанавливается класс, к которому относится решаемая задача, и приводится сравнительный анализ методов решения для выбора наиболее эффективного метода. Приводится обоснование выбора метода решения. Вместо математической модели для формализации задачи может быть выбран любой иной вид моделей, в том числе функциональные, информационные, событийные, структурные. Могут быть представлены модели «как есть» и «как должно быть». В этом случае также следует предложить способы перехода. В целом, во второй главе определяется общая последовательность решения задачи. Здесь же приводятся результаты теоретических исследований. Описание разработанных алгоритмов, анализ их эффективности может присутствовать как во второй главе, так и вынесено в отдельную главу (алгоритмическое обеспечение). Все зависит от объема представляемого материала.

Глава 3. Реализация

3.1. Средства реализации

TBD

- IntelliJ IDEA 2019.1;
- система контроля версий Git;
- TBD

3.2. Требования к программному и аппаратному обеспечению

Требования к аппаратному и программному обеспечению:

- RAM: 1 Гб минимум, 2 Гб рекомендовано;
- свободное место на диске: 300 Мб + не менее 1 Гб для кэша;
- минимальное разрешение экрана — 1024×768;
- JDK 8 и выше; TBD
- IntelliJ IDEA 9 и выше.

3.3. Реализация

TBD

3.4. План тестирования

TBD

Заключение

В заключении логически последовательно излагаются теоретические и практические выводы, результаты и предложения, которые получены в результате исследования. Они должны быть краткими, четкими, дающими полное представление о содержании, значимости, обоснованности и эффективности исследований и разработок. Кроме того, в заключении можно представить практическую значимость и результаты реализации работы, подразумевающие разработку математического, алгоритмического, программного обеспечения для решения определенной задачи или класса задач, наличие внедрения в учебный, исследовательский, производственный процесс, регистрацию программных средств, наличие патента, рекомендации к использованию. В заключении приводится список публикаций автора и апробация работы на конференциях различного уровня.

Список литературы

1. https://ru.wikipedia.org/wiki/Тестирование_программного_обеспечения
2. https://ru.wikipedia.org/wiki/Ручное_тестирование
3. https://ru.wikipedia.org/wiki/Автоматизированное_тестирование
4. <https://junit.org/junit5/>
5. Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys (csur), 29(4), 366-427.
6. https://ru.wikipedia.org/wiki/Логика_ХоараТройки_Хоара
7. <https://en.wikipedia.org/wiki/QuickCheck>
8. <https://jqwik.net>
9. <https://sttp.site/chapters/intelligent-testing/mutation-testing.html>
10. <http://pitest.org>
11. <https://ru.wikipedia.org/wiki/Фаззинг>

Приложение А. Листинг кода**TBD**