МИНОБРНАУКИ РОССИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОРОНЕЖСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

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

Автогенерация модульных тестов для Java программ

Магистерская диссертация
Направление 02.04.03 Математическое обеспечение и администрирование
информационных систем
Программа Информационные технологии

Допущено к защите в ГЭК			___.___2021 г.
Зав. кафедрой	_____	д. ф.-м. н., проф.	М. А. Артёмов
Обучающийся	_____		А. С. Пахомов
Руководитель	_____	д. ф.-м. н., проф.	М. А. Артёмов

Воронеж 2021

Аннотация

Работа посвящена исследованию методов и техник тестирования Java программ с помощью искусственного интеллекта.

Целью работы является разработка инструмента автогенерации модульных тестов для Java программ.

Область применения разработанного инструмента — тестирование программных компонентов без участия человека.

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 5
1.2. Подходы к написанию автоматизированных тестов 8
1.2.1. Тестирование на основе спецификации 9
1.2.2. Тестирование границ 12
1.2.3. Структурное тестирование 15
1.2.4. Тестирование на основе модели 22
1.2.5. Тестирование на основе контракта 24
1.2.6. Тестирование свойств 28
1.3. Тестирование с помощью искусственного интелекта 29
1.3.1. Статическое тестирование 29
1.3.2. Мутационное тестирование 31
1.3.3. Генерация псевдослучайных входных данных 33
1.3.4. Тестирование на основе анализа кода 35
Глава 2. Анализ задачи 38
2.1. Постановка задачи 38
2.2. Обзор существующих инструментов автогенерации модульных тестов 38
2.2.1. Randoop 38
2.2.2. EvoSuite 39
2.2.3. Cover 40
Глава 3. Реализация 41
3.1. Средства реализации 41
3.2. Требования к программному и аппаратному обеспечению 41
3.3. Схема работы инструмента 41
3.4. Реализация алгоритма 42
3.5. Вычисление функции приспособленности 44
3.6. Сценарий использования 45
3.7. План тестирования 45
Заключение 47
Список литературы 48

Приложение А. Листинг тестируемого класса 50
Приложение Б. Листинг сгенерированного модульного теста 51

Введение

Разработка программного обеспечения включает в себя процесс тестирования. Самым эффективным способом тестирования на сегодняшний день считается автоматизированное тестирование. Чаще всего автоматизируется сам запуск тестовых сценариев. Процесс анализа и написания тестов производится человеком.

Актуальность работы обусловлена необходимостью производства качественного программного обеспечения с минимальным участием человека в процессе тестирования.

Предметом изучения является тестирование программного обеспечения с помощью искусственного интеллекта.

Целью данной работы является исследование методов тестирования программного обеспечения с помощью искусственного интеллекта и разработка инструмента автогенерации модульных тестов из Java кода.

Практическая значимость работы обусловлена возможностью применения разработанного инструмента в коммерческих программных комплексах, а так же в программном обеспечении с открытым исходным кодом. Автоматизированный процесс генерации модульных тестов повышает качество разрабатываемого программного обеспечения без участия человека.

Глава 1. Анализ существующих подходов к тестированию

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].

Преимущества такого способа тестирования:

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

Основные проблемы ручного тестирования:

- Наличие человеческого труда. Тестировщик 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).

Преимущества автоматизированного тестирования:

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

Недостатки автоматизированного тестирования:

– Трудоемкость. Поддержка и обновление тестов являются трудоемкими процессами.
– 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).

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;
  }
}
```

При разработке тестовых сценариев главным изменяющимся компонентом являются входные данные. Структура теста меняется реже. Чтобы избежать дублирования кода, JUnit 5 предоставляет возможность писать *параметризованные тесты*. Чтобы заменить обычный

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);
  }
}
```

тест параметризованным, достаточное 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.

Разработка тестовых сценариев подразумевает анализ, дизайн и написание кода. Смысл этой фазы состоит в том, что бы разработать

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);
}
```

такое множество тестовых сценариев, которое бы удовлетворяло стандартам качества разрабатываемого программного обеспечения. Понятие «автоматизированное тестирование» не включает в себя автоматизацию этой фазы.

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.

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.

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;
  }
}
```

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

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 интерпретировано 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, основанном 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.

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.

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;
  }
}
```

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.

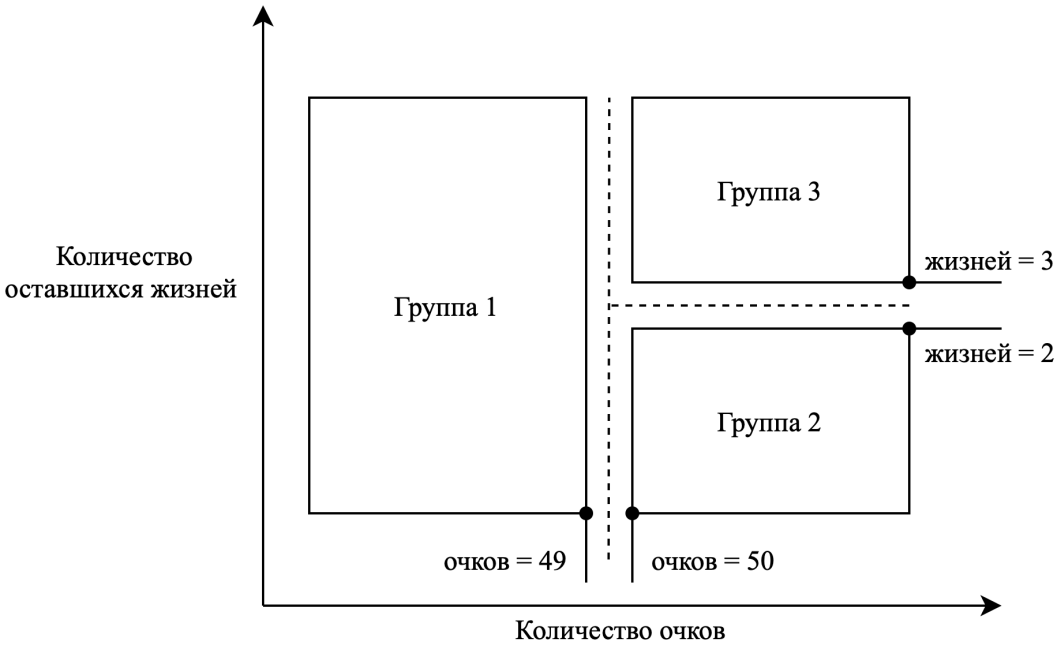

Рис. 1.1. Границы групп

Тестирующий код представлен в листинге 1.8.

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. Это 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.

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);
  }
}
```

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

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

Не исполненной остается только строка 8. Второй тест её исполяет. 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 показыкает 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.

Преимущество 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.

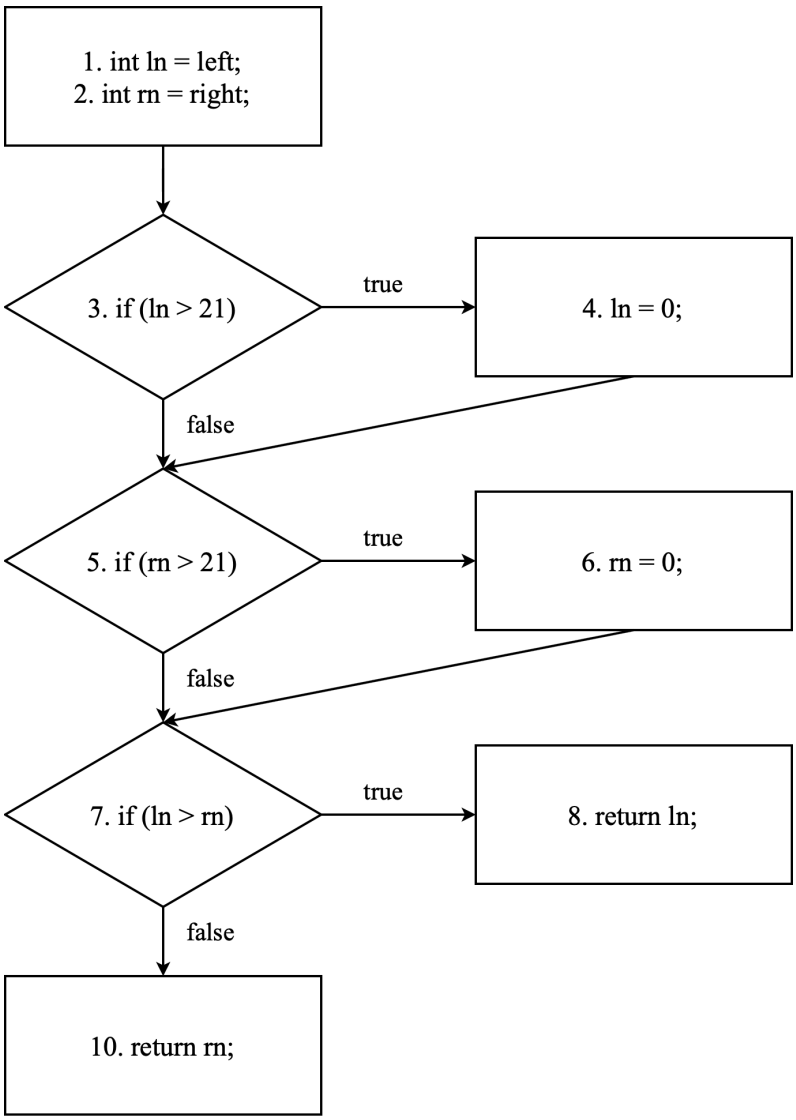

Рис. 1.2. Граф потока управления для программы `BlackJack`

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

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

Комбинация из ребра и блока составляет *решение* или *ветвь*. На основе решений 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) – `true` и (a=5, b=10, c=5) – `false`. Однако, эти наборы вводных данных не покрывают все логические комбинации. Например: (a=20, b=30, c=5) – `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 A в блок B.

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

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 MC/DC: 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 интерпретировать как `(A & (B | C))` следующим образом:

- `A = !Character.isLetter(str.charAt(i));`
- `B = last == 's';`
- `C = last == 'r'.`

Чтобы достичь 100 % покрытия, используя критерий покрытия путей, нужно написать 8 тестовых сценариев (2^3). MC/DC критерий позволяет сократить число тестовых сценариев до 4. Проблема состоит

в том, чтобы 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 = true, B = false, C = true) = true`. Изменяем параметр C.
- `(A = true, B = false, C = false) = false`. Изменяем параметр B.
- `(A = true, B = true, C = false) = true`. Изменяем параметр A.
- `(A = false, B = true, C = false) = false`.

Преимуществом 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.

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, независимо от выбранного критерия покрытия, не способно полностью заменить

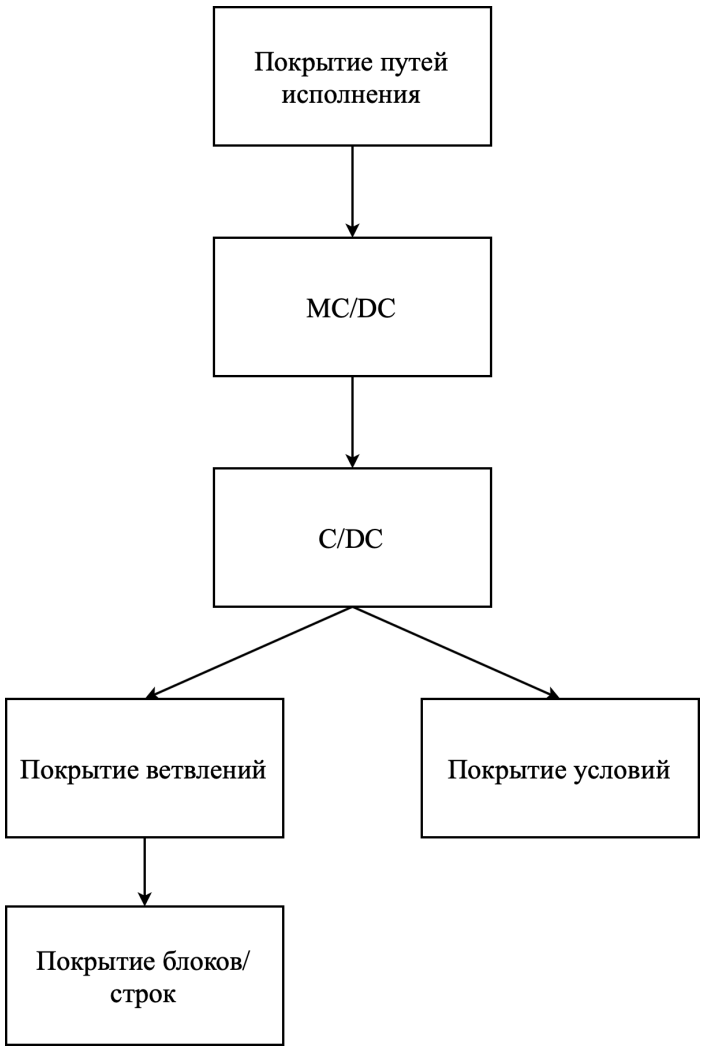

Рис. 1.3. Соотношение критериев покрытия

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			
Условия	<Условие1>	T	T	F	F
	<Условие2>	T	F	T	F
Действие	<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

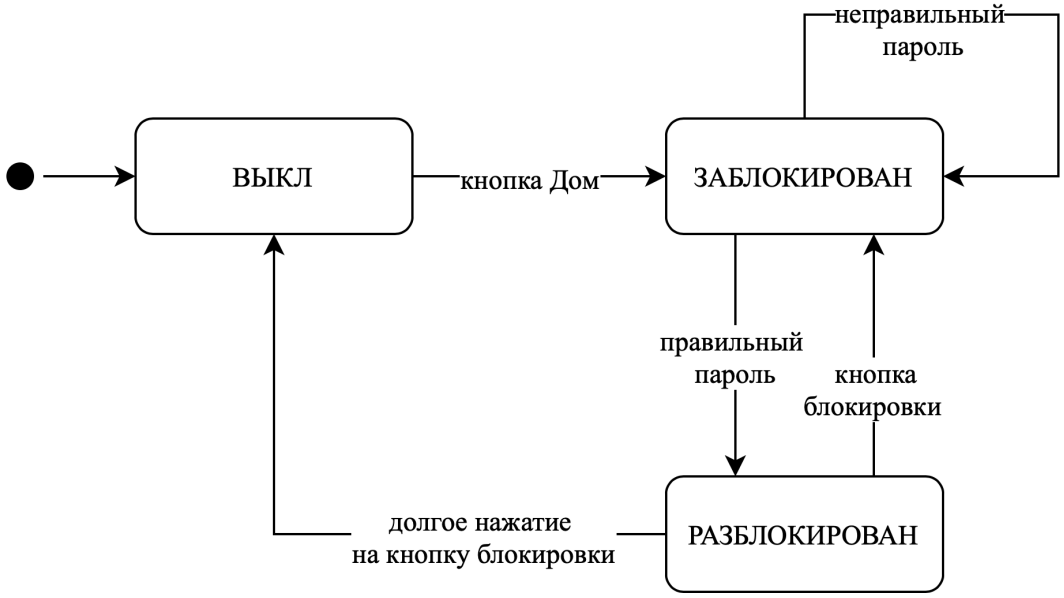

Рис. 1.4. Конечный автомат главного экрана телефона

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.

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

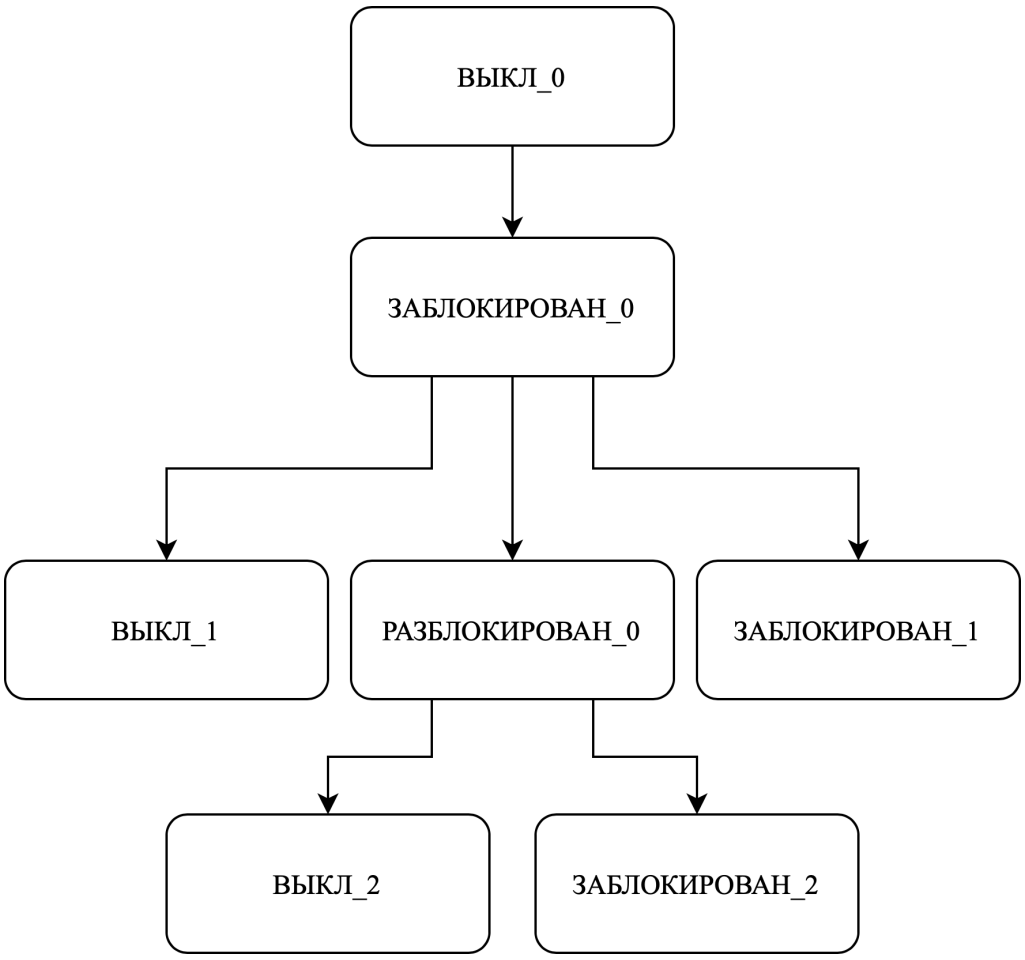

Рис. 1.5. Дерево переходов главного экрана телефона

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.

Стоит отметить, что утверждения не заменяют модульное тестирование.

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";
  }
}
```

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.

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

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";
  }
}
```

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, в начале 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, обьявленный интерфейсом и его имплементацию.

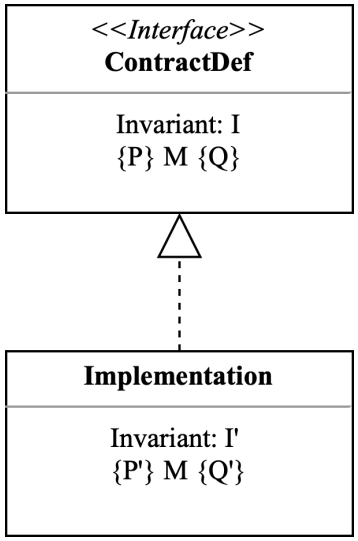

Рис. 1.6. Контракт и его реализация

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, чтобы 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 множесто других аннотаций).

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

Листинг 1.16 Тестирование свойста 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());
  }
}
```

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

1.3. Тестирование с помощью искусственного интелекта

До настоящего момента под «автоматизацией» понималась автоматизация выполнения тестов. Продвинутые подходы к автоматизации тестирования расширяют понятие атоматизиции. Они исключают человеческий фактор в процессе тестирования. Это означает отсутсвие необходимости разрабатывать тестовые сценарии. Компьютер сделает это за человека. Иногда такие подходы называются тестированием с помощью искусственного интелекта.

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 не учитанной 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

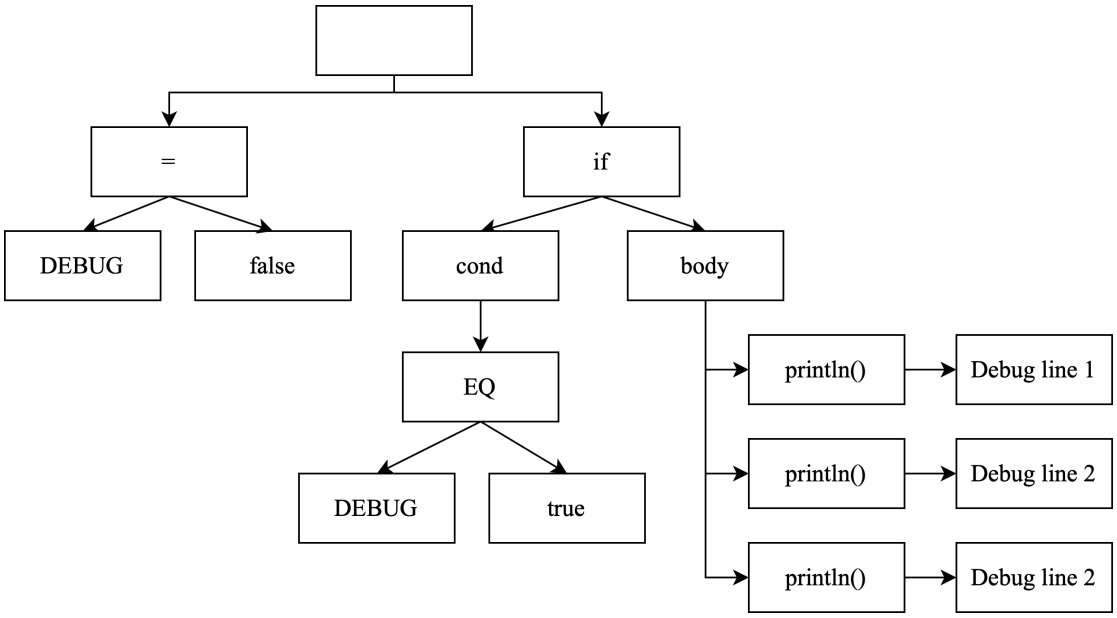

Рис. 1.7. Абстрактное синтаксическое дерево

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, таких критериев может быть недостаточно 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 присваевания.* Замена операторов `=`, `+=`, `-=`, `/=`.
- *Оператор замены скалярных переменных.* Замена одной переменной на другую.

Помимо описанных выше мутационных 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 о 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 предопределены.
- *Мутирующие фаззеры.* На вход фаззеру даётся пример входных данных. Фаззер применяет различные мутации к данным и проверяет поведение системы. Примеры мутаций: замена символов в строке, замена битов. Некоторые инструменты (American Fuzzy Lop) 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 это процесса используется *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 1.18 определяется функция с двумя входными параметрами a и b. Для удовлетворения предиката `else if` нужно удовлетворить ограничениям $((N + M <= 2)\&(N < 100))$. 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 $N + M <= 2$.
4. Ограничение ветки `else if`: $N < 100$.
5. Итоговое ограничение ветки `else if` является комбинацией двух предыдущих: $(N + M <= 2)\&(N < 100)$

Листинг 1.18 Пример кода

```
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;
}
```

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

1.3.4. Тестирование на основе анализа кода

Тестирование на основе анализа кода (англ. Search-based software testing, SBST) — автоматизированный процесс анализа исходного кода и генерации для него тестовых сценариев.

Основная цель SBST — достижение 100 % покрытия кода минимальным набором тестов. Это задача оптимизации [12,13]. Прежде чем рассмотреть как она решается, нужно понять как работают инструменты анализа и генерации кода. Для класса из листинга 1.19 алгоритм генерации тестов будет выглядеть следующим образом:

1. Создать экземпляр класса.
2. Если конструктор имеет параметры, сгенерировать случайные параметры и подставить их.
3. Вызвать тестируемый метод.

4. Если метод имеет параметры, сгенерировать их случайным образом и подставить.
5. Если метод вернул результат, сохранить его в переменную.
6. Использовать полученный результат для написания утверждений (assertions).
7. Замерить полученное покрытие кода.
8. Повторить все действия, пока не будет достигнуто суммарное покрытие тестами 100 % или закончится время.

Листинг 1.19 Пример кода, определяющего тип треугольника

```
public class Triangle {

  private final int a, b, c;

  enum TriangleType {
    EQUILATERAL, ISOSCELES, SCALENE
  }

  public Triangle(int a, int b, int c) {
    this.a = a;
    this.b = b;
    this.c = c;
  }

  public TriangleType classify(int a, int b, int c) {
    if (a == b && b == c)
      return TriangleType.EQUILATERAL;
    else if (a == b || b == c)
      return TriangleType.ISOSCELES;
    else
    return TriangleType.SCALENE;
  }
}
```

Пример случайно сгенерированного тестового сценария представлен в листинге 1.20.

Для оптимизации времени генерации и качества тестовых сценариев используется ряд эвристик. Самый популярный способ оптимизации — генетический алгоритм. Функцией оптимизации является критерий

Листинг 1.20 Случайно сгенерированный тест

```
@Test
void t1() {
  Triangle t = new Triangle(5, 7, 10);
  Triangle.TriangleType type = t.classify();
  assertThat(type).isEqualTo(Triangle.TriangleType.SCALENE);
}
```

покрытия. Чем он выше, тем больше шансов у определенного теста пройти этап селекции и попасть в итоговый набор тестов. Критерий покрытия может быть разный. Самый распространенный — критерий покрытия веток исполнения.

Тестирование на основе анализа кода — область компьютерных наук, которая находится в стадии активного изучения. На данный момент не существует широко распространенных инструментов автоматической генерации тестов. В научных сообществах есть два инструмента: Randoop [14] и Evosuite [15]. Существует платный инструмент Cover, разработанный компанией Diffblue, которая занимается разработкой программных продуктов для анализа кода [16]. Одну из первых попыток использовать SBST в промышленной разработке стала компания Facebook со своим инструментом Sapienz [17] для автоматической генерации тестовых сценариев для мобильного приложения.

Глава 2. Анализ задачи

2.1. Постановка задачи

Проанализировать существующие инструменты автогенерации модульных тестов. Разработать инструмент автогенерации модульных тестов для Java программ. Инструмент должен удовлетворять следующим требованиям:

- поставляться в виде отдельной библиотеки;
- для указанного Java класса генерировать соответствующий модульный тест или несколько модульных тестов;
- набор тестовых сценариев должен обеспечивать максимально возможное покрытие по критерию покрытия веток исполнения;
- сгенерированный код должен быть понятен человеку.

2.2. Обзор существующих инструментов автогенерации модульных тестов

Автогенерация модульных тестов — свежая область компьютерных наук. Большинство инструментов автогенерации кода используются исследователями и служат примерами идей, таких как фаззинг или SBST.

2.2.1. Randoop

Randoop — первый инструмент автогенерации модульных тестов для Java программ [14]. Сценарий использования:

1. скачать архив;
2. разархивировать скаченный файл;
3. указать соответствующие переменные окружения RANDOOP_PATH и RANDOOP_JAR;
4. запустить java программу и указать набор классов для генерации.

Преимущества Randoop:

- открытый исходный код;
- лицензия, не ограничивающая коммерческое использование;

Недостатки:

- сложность использования;
- низкое качество сгенерированного кода: сложно читать, слабое покрытие;
- не поддерживаются версии Java, выше 11;

– не поддерживается библиотека для тестирования JUnit версии 5.

2.2.2. EvoSuite

EvoSuite — инструмент автогенерации модульных тестов, который используется исследователями для проверки гипотез и разработки новых алгоритмов автогенерации кода [15].

Сценарий использования:

1. скачать jar файл;
2. запустить java программу и указать набор классов для генерации.

Преимущества EvoSuite:

– открытый исходный код;
– высокое качество сгенерированного кода;

Недостатки:

– лицензия, ограничивающая коммерческое использование;
– сложность использования;
– не поддерживаются версии Java, выше 11;
– не поддерживается библиотека для тестирования JUnit версии 5.

2.2.3. Cover

Cover — коммерческий продукт, разработанный компанией Diffblue [16]. Является самым удобным и качественным инструментом на данный момент.

Сценарий использования:

1. скачать IntelliJ IDEA плагин или консольную программу;
2. запустить консольную программу или нажать на соответствующую кнопку в idea.

Преимущества Cover:

– высокое качество сгенерированного кода;
– поддержка современных библиотек, таких как Spring;
– удобство использования.

Недостатком инструмента является его платная основа.

Сравнительный анализ описанных ранее инструментов приведен в табл. 2.1.

Таблица 2.1. Сравнительный анализ инструментов

Инструмент	**Randoop**	**EvoSuite**	**Cover**
коммерческое использование	+	–	платно
удобство использования	–	–	+
открытый исходный код	+	+	–
качество сгенерированного кода	низкое	высокое	высокое
поддержка новых библиотек	–	–	+

Глава 3. Реализация

3.1. Средства реализации

– Intellij IDEA 2021.1;
– система контроля версий Git;
– язык программирования Java 11;
– библиотека разбора java файлов JavaParser;
– библиотека подсчета покрытия кода JaCoCo;
– библиотека автоматизации тестирования JUnit 5.

JavaParser — популярная библиотека для разбора и генерации java файлов. Ее использование обусловлено наличием открытого исходного кода и поддержкой последних версий Java.

Для подсчета покрытия кода была выбрана библиотека JaCoCo. Она поддерживает необходимый критерий покрытия веток исполнения, а так же обладает обширной документацией.

Для запуска сгенерированных тестов используется библиотека JUnit 5.

3.2. Требования к программному и аппаратному обеспечению

– RAM: 1 Гб минимум, 2 Гб рекомендовано;
– свободное место на диске: 300 Мб;
– JDK 8 и выше.

3.3. Схема работы инструмента

Схема работы инструмента представлена на рис. 3.1.

На первой фазе происходит парсинг исходных java файлов. В ходе разбора составляется модель данных, которая содержит следующую информацию:

– пакет тестируемого класса;
– имя тестируемого класса;
– список конструкторов;
– список публичных методов с параметрами.

В результате получается модель данных, представленная на рис. 3.2.

После анализа кода происходит создание потенциальных представлений (скелетов) теста. Скелет теста представляет собой прототип кода модульного теста, но без конкретных параметров. На месте параметров находятся специальные заглушки.

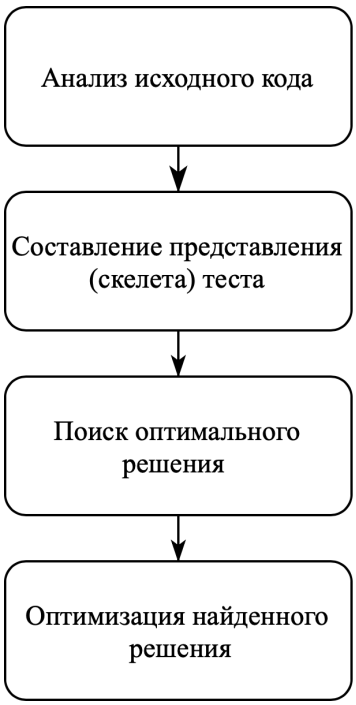

Рис. 3.1. Схема работы инструмента

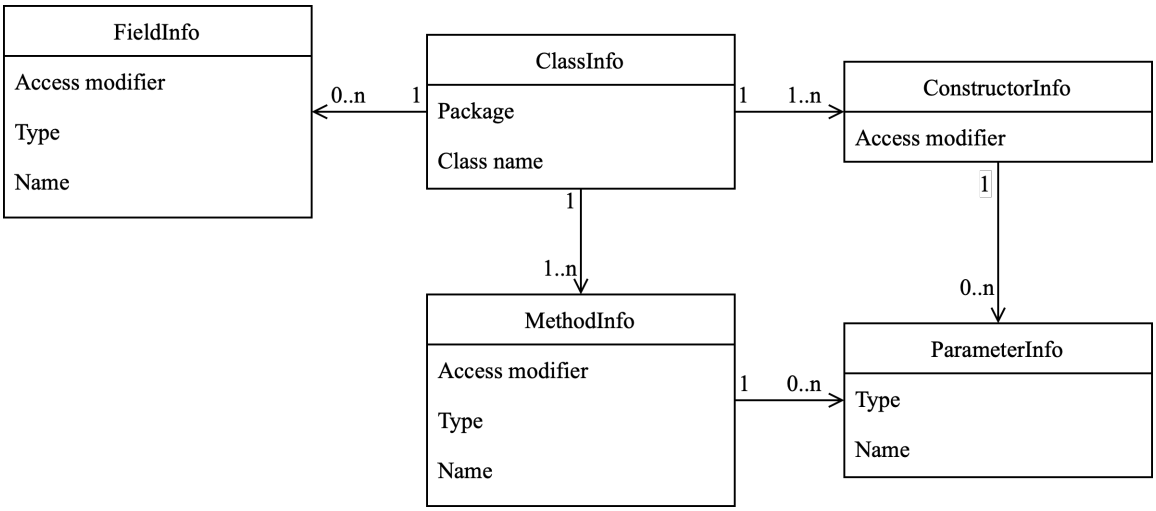

Рис. 3.2. Модель данных

Третья фаза — поиск оптимального решения. В качестве реализации поиска был выбран генетический алгоритм.

В конце происходит оптимизация найденного решения. В результате работы алгоритма может получиться тестовый сценарий, который является избыточным. Избыточный тестовый сценарий — сценарий, из которого можно убрать часть кода без потери покрытия.

3.4. Реализация алгоритма

Основным алгоритмом поиска оптимального решения является генетический алгоритм. Визуализация его работы представлена на рис. 3.3.

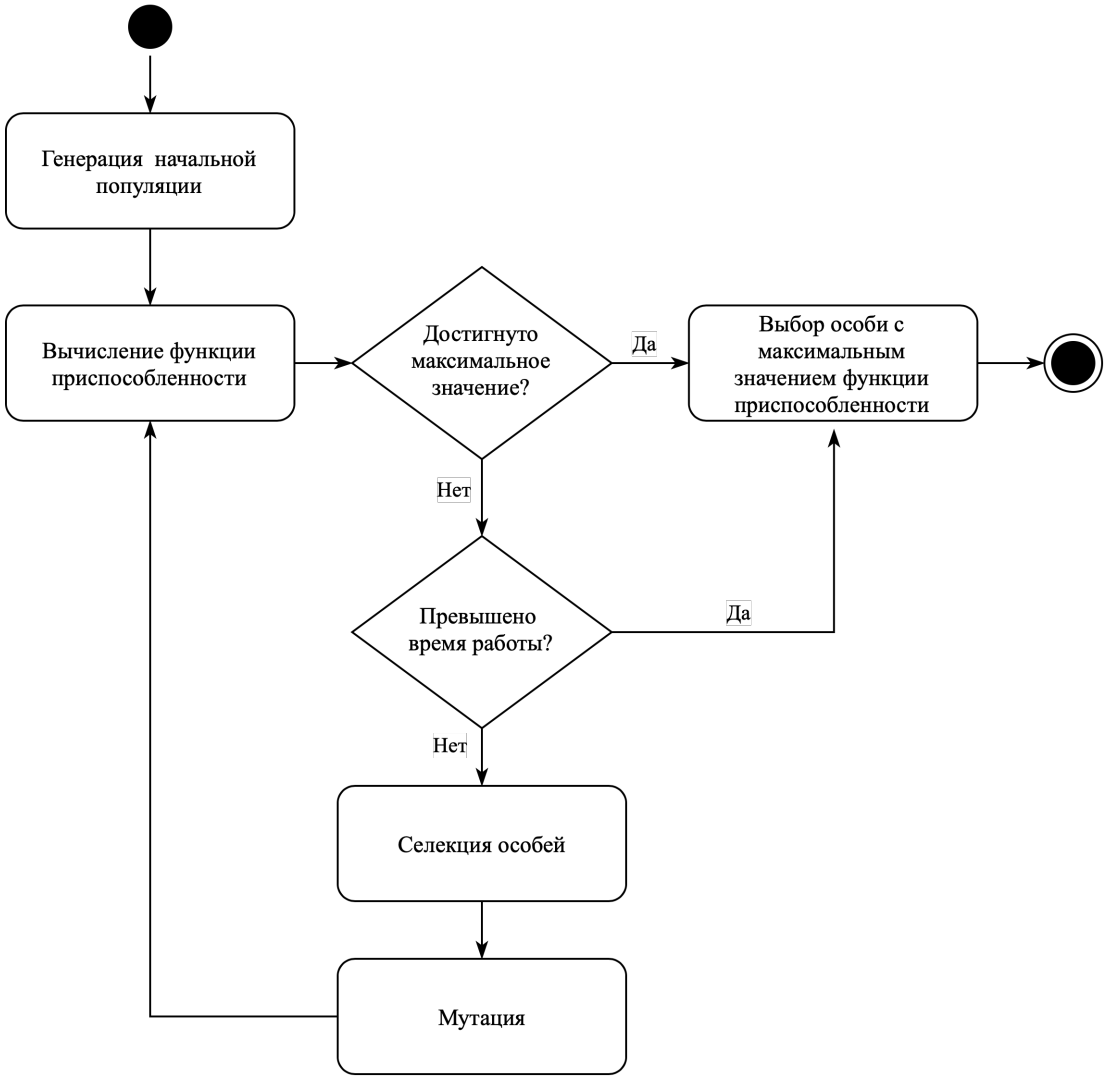

Рис. 3.3. Визуализация работы генетического алгоритма

Вначале генерируется популяция, которая представляет собой набор скелетов тестов с конкретными параметрами. Затем для каждой особи (конкретный модульный тест) вычисляется функция приспособленности (фитнес функция). Результат функции приспособленности сравнивается с максимальным (1.0), после чего принимается решение о выходе из алгоритма или продолжении.

Для поиска более удачной комбинации модульных тестов из популяции выбираются две особи с максимальным значением функции приспособленности (селекция). Выбранные особи скрещиваются, в результате чего появляются два новых модульных теста, которые объединяют гены (конкретные тестовые методы) особей-предков.

Для избежания попадания в локальные максимумы (ситуация, когда вся популяция сбивается вокруг одно набора генов) на каждой итерации

алгоритма используется процесс мутации. Для мутации случайным образом выбираются несколько особей. У выбранных особей запускается процесс мутации, в ходе которого случайный параметр вызова метода заменяется на другой. Это может быть случайный параметр или мутированный (в случае чисел +1 или -1, в случае строк это замена битов).

3.5. Вычисление функции приспособленности

Функция приспособленности — одна из важнейших частей генетического алгоритма. Качество ее работы определяет результат и скорость работы алгоритма.

Последовательность действий, выполняемая в ходе вычисления функции приспособленности представлена на рис 3.4.

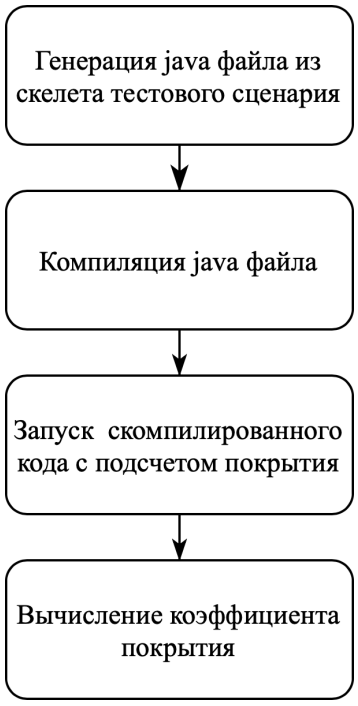

Рис. 3.4. Вычисление функции приспособленности

Для того, чтобы оценить качество генерируемого теста, нужно:

1. сгенерировать java файл из скелета теста с параметрами;
2. скомпилировать полученный java файл;
3. запустить скомпилированный код с подсчетом покрытия по критерию покрытия ветвей исполнения;
4. вычислить коэффициент покрытия.

Вычисление коэффициента покрытия происходит по формуле:

$$\text{коэффициент покрытия} = \frac{\text{количество задействованных ветвей}}{\text{общее количество ветвей}} \times 100\%$$

Коэффициент покрытия является результатом работы функции приспособленности.

3.6. Сценарий использования

Для того, чтобы сгенерировать модульный тест для произвольного Java класса, необходимо:

1. добавить библиотеку в зависимости проекта: `testCompile project(':sbst-java-core');`
2. создать java класс, в методе main которого нужно запустить алгоритм, листинг 3.1;
3. результат работы алгоритма будет положен в java файл, имя которого совпадает с именем тестируемого класса, с добавлением постфикса `Test`.

Тестируемый класс и результат работы алгоритма представлены в приложениях А и Б соответственно.

3.7. План тестирования

Для проверки корректности работы инструмента необходимо провести ряд тестов.

Тест 1. «Генерация модульного теста для класса без конструктора с одним публичным методом без параметров»

Цель: проверить корректность работы инструмента для класса без конструктора с одним публичным методом без параметров.

Порядок проведения: создать java класс без конструктора с одним публичным методом без параметров, запустить инструмент.

Результат: сгенерирован модульный тест, который достигает полного покрытия.

Тест 2. «Генерация модульного теста для класса без конструктора с одним публичным методом с одним параметром типа int»

Цель: проверить корректность работы инструмента для класса без конструктора с одним публичным методом с одним параметром типа int.

Порядок проведения: создать java класс без конструктора с одним публичным методом с одним параметров типа int, запустить инструмент.

Результат: сгенерирован модульный тест, который достигает полного покрытия.

Тест 3. «Генерация модульного теста для класса без конструктора с одним публичным методом с двумя параметрами типа int»

Цель: проверить корректность работы инструмента для класса без конструктора с одним публичным методом с двумя параметрами типа int.

Порядок проведения: создать java класс без конструктора с одним публичным методом с двумя параметрами типа int, запустить инструмент.

Результат: сгенерирован модульный тест, который достигает полного покрытия.

Тест 4. «Генерация модульного теста для класса с одним конструктором и с одним публичным методом без параметров»

Цель: проверить корректность работы инструмента для класса с одним конструктором и с одним публичным методом без параметров.

Порядок проведения: создать java класс с одним конструктором и с одним публичным методом без параметров, запустить инструмент.

Результат: сгенерирован модульный тест, который достигает полного покрытия.

Тест 5. «Генерация модульного теста для класса с одним конструктором и с одним публичным методом с одним параметром типа String»

Цель: проверить корректность работы инструмента для класса с одним конструктором и с одним публичным методом с одним параметром типа String.

Порядок проведения: создать java класс с одним конструктором и с одним публичным методом с одним параметром типа String, запустить инструмент.

Результат: сгенерирован модульный тест, который достигает полного покрытия.

Тест 6. «Корректное завершение работы при ошибке пользователя»

Цель: проверить корректность работы инструмента в случае указания пользователем несуществующего класса.

Порядок проведения: запустить инструмент с указанием несуществующего класса.

Результат: вывод ошибки в стандартный поток вывода.

Заключение

В результате работы были проанализированы существующие инструменты автогенерации модульных тестов. Разработан инструмент автогенерации модульных тестов для Java программ. Инструмент удовлетворяет следующим требованиям:

- поставка в виде отдельной библиотеки;
- генерация соответствующего модульного теста или нескольких модульных тестов для указанного Java класса;
- набор тестовых сценариев обеспечивает максимально возможное покрытие по критерию покрытия веток исполнения;
- сгенерированный код понятен человеку.

Публикация результатов. Материалы по исследованию техник модульного тестирования были опубликованы в статье: Брусенцев И.М., Пахомов А.С. Анализ техники ведения разработки через тестирование на языке Java // Актуальные проблемы прикладной математики, информатики и механики : сборник трудов международной научной конференции, Воронеж, 7-9 декабря. – Воронеж, 2020. – С. 601-603; URL: http://www.amm.vsu.ru/conf/index.php?page=Doklads (дата обращения: 12.06.2021).

Список литературы

1. Тестирование программного обеспечения. – URL: https://ru.wikipedia.org/wiki/Тестирование_программного_обеспечения (дата обращения 12.06.2021).
2. Ручное тестирование. – URL: https://ru.wikipedia.org/wiki/Ручное_тестирование (дата обращения 12.06.2021).
3. Автоматизированное тестирование. – URL: https://ru.wikipedia.org/wiki/Автоматизированное_тестирование (дата обращения 12.06.2021).
4. JUnit 5. – URL: https://junit.org/junit5/ (дата обращения 12.06.2021).
5. Zhu H. Software unit test coverage and adequacy / H. Zhu, P. A. Hall, J. H. May // ACM computing surveys (csur). – 1997. – 29(4). – C. 366-427.
6. Тройки Хоара. – URL: https://ru.wikipedia.org/wiki/Логика_ХоараТройки_Хоара (дата обращения 12.06.2021).
7. QuickCheck. – URL: https://en.wikipedia.org/wiki/QuickCheck (дата обращения 12.06.2021).
8. Jqwik. – URL: https://jqwik.net (дата обращения 12.06.2021).
9. Мутационное тестирование. – URL: https://sttp.site/chapters/intelligent-testing/mutation-testing.html (дата обращения 12.06.2021).
10. Pitest. – URL: http://pitest.org (дата обращения 12.06.2021).
11. Фаззинг. – URL: https://ru.wikipedia.org/wiki/Фаззинг (дата обращения 12.06.2021).
12. Harman M. Search Based Software Engineering: Trends, Techniques and Applications / M. Harman, S. Afshin Mansouri, Y. Zhang // ACM Computing Surveys.
13. Phil McMinn. Search-Based Software Testing: Past, Present and Future / Phil McMinn // University of Sheffield, Department of Computer Science Regent Court, 211 Portobello, Sheffield, S1 4DP. – UK.
14. Randoop. – URL: https://randoop.github.io/randoop/manual/dev.html (дата обращения 12.06.2021).
15. EvoSuite. – URL: https://www.evosuite.org (дата обращения 12.06.2021).

16. Cover. – URL: https://www.diffblue.com (дата обращения 12.06.2021).
17. Sapienz. – URL: https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/ (дата обращения 12.06.2021).

Приложение А. Листинг тестируемого класса

```
package com.github.sbst.java.example;

public class ClassUnderTest {

    public int calculateSumOrZero(int a, int b) {
        if (a > 100  && b > 100) {
            return 100;
        }
        if (a > 100 && b > 50) {
            return 50;
        }
        if (b < 0 && a < 0) {
            return -1;
        }
        if (b < 0) {
            return -10;
        }
        if (b > 110) {
            return 110;
        }
        return a + b;
    }
}
```

Приложение Б. Листинг сгенерированного модульного теста

```
package com.github.sbst.java.example.generated;

import org.junit.jupiter.api.Test;

public class ClassUnderTestTest {

    com.github.sbst.java.example.ClassUnderTest classUnderTest
    = new com.github.sbst.java.example.ClassUnderTest();

    @Test()
    void shouldCalculateSumOrZero2267474524() {
        int a = 2;
        int b = 67317;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = 110;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }

    @Test()
    void shouldCalculateSumOrZero4477528() {
        int a = 1126;
        int b = -15;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = -10;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }

    @Test()
    void shouldCalculateSumOrZero5531() {
        int a = -6;
        int b = 4;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = -2;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }
```

```
    @Test()
    void shouldCalculateSumOrZero6533() {
        int a = 120;
        int b = 261;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = 100;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }

    @Test()
    void shouldCalculateSumOrZero7483534() {
        int a = -7;
        int b = -1;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = -1;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }

    @Test()
    void shouldCalculateSumOrZero226747452443453() {
        int a = 70;
        int b = 67317;
        int calculateSumOrZeroResult = classUnderTest.
   calculateSumOrZero(a, b);
        int expectedResult = 50;
        org.assertj.core.api.Assertions.assertThat(
   calculateSumOrZeroResult).isEqualTo(expectedResult);
    }
}
```