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

Факультет прикладной математики, информатики и механники
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 особенно- сти. 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

Введение содержит в сжатой форме положения, обоснованию кото- рых 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 ис- пользованием системы критериев. Результаты анализа могут быть пред- ставлены 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]. Существеут множество техник и подходов к тестированию программноо 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 видет пользователь.

Основные проблемы ручного тестирования:

– Наличие человеческого труда. Тестировщих может допустить ошибку в процессе ручных действий.
– Выполнение ручных дейсвий 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 обеспесения 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 разработки программного обеспечиния.

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

JUnit

JUnit — самый распространненый 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).

1.2. Подходы к написанию автоматизированных тестов

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

Разработка тестовых сценариев подразуевает 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 1.1 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;
  }
}
```

Вторая фаза подразумевает инсталяцию 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

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);
  }
}
```

должен быть конечен и выполняться достаточно быстро. Далее будут рассмотренны 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 висококосным 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 представленна в листинге 1.3.

Листинг 1.3 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 интерптитированно как корректное или не корректное.

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, провоцирует одно и то же поведение 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 1.4.

1.2.2. Тестирование границ

Ошибки, основанные на граничных условиях очень распростаннены. 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 групе B.

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

Листинг 1.4 Теструющий 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);
  }
}
```

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 1.5.

Листинг 1.5 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 1.6.

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 меньше6 чем 3, тогда набор данных относится к группе 2, иначе он относится к группе 3.

   Получившиеся границы представленны на рис. 1.1.

   Тестирующий код представлен в листинге 1.7.

1.2.3. Структурное тестирование

TBD

1.2.4. Тестирование на основе модели

TBD

Листинг 1.6 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));
  }
}
```

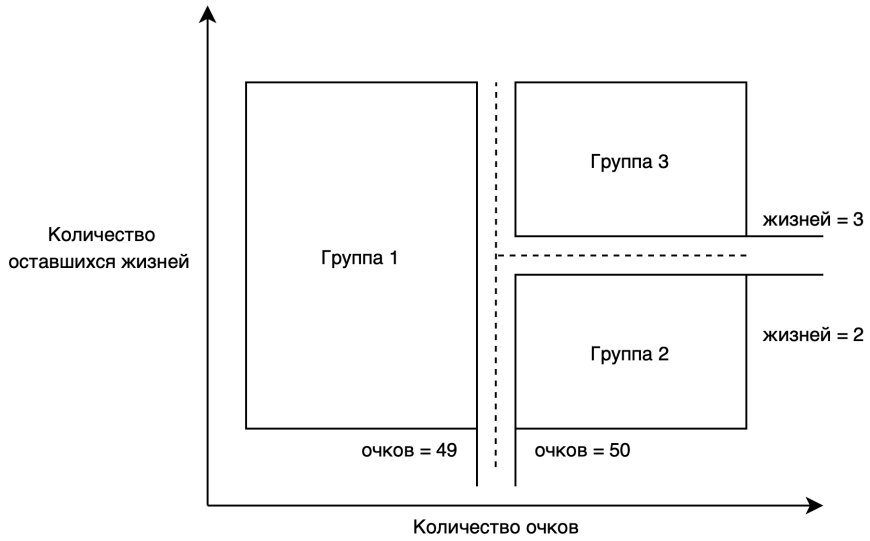

Рис. 1.1. Границы групп

1.2.5. Тестирование на основе контракта

TBD

1.2.6. Тестирование свойств

TBD

Листинг 1.7 Тестирующий код

```
@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));
}
```

1.3. Продвинутые подходы написания автоматизированных тестов

1.3.1. Статическое тестирование

TBD

1.3.2. Мутационное тестирование

TBD

1.3.3. Генерация входных данных

TBD

1.3.4. Тестирование на основе анализа кода

TBD

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 си- стем и использованием информационных технологий, в содержательной постановке приводятся ссылки на документы, регламентирующие процесс функционирования информационной системы, основные показатели, ко- торые должны быть достигнуты в условиях эксплуатации информационной системы; ограничения на время решения поставленной задачи, сроки выдачи информации, способы организации диалога человека с инфор- мационной системой средствами имеющегося инструментария, описание входной и выходной информации (форма представления сообщений, описа- ние структурных единиц, периодичность выдачи информации или частота поступления), требования к организации сбора и передачи входной инфор- мации, ее контроль и корректировка. В математической постановке (при наличии) выполняется формали- зация задачи, в результате которой определяется состав переменных, кон- стант, их классификация, виды ограничений на переменные и математиче- ские зависимости между переменными. Устанавливается класс, к которому относится решаемая задача, и приводится сравнительный анализ методов решения для выбора наиболее эффективного метода. Приводится обоснова- ние выбора метода решения. Вместо математической модели для формализации задачи может быть выбран любой иной вид моделей, в том числе функциональные, информа- ционные, событийные, структурные. Могут быть представлены модели «как есть» и «как должно быть». В этом случае также следует предложить спо- собы перехода. В целом, во второй главе определяется общая последовательность решения задачи. Здесь же приводятся результаты теоретических исследова- ний. Описание разработанных алгоритмов, анализ их эффективности мо- жет 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 эффек- тивности исследований и разработок. Кроме того, в заключении можно представить практическую значи- мость и результаты реализации работы, подразумевающие разработку ма- тематического, алгоритмического, программного обеспечения для решения определенной задачи или класса задач, наличие внедрения в учебный, ис- следовательский, производственный процесс, регистрацию программных средств, наличие патента, рекомендации к использованию. В заключении приводится список публикаций автора и апробация ра- боты на конференциях различного уровня.

Список литературы

а) https://ru.wikipedia.org/wiki/Тестирование
б) https://ru.wikipedia.org/wiki/Ручное
в) https://ru.wikipedia.org/wiki/Автоматизированное
г) https://junit.org/junit5/

Приложение А. Листинг кода

TBD