

Оглавление

| | |
|--|---|
| Основная информация..... | 2 |
| История редактирования..... | 2 |
| 1. Введение..... | 3 |
| 2. Цель..... | 3 |
| 3. Область тестирования..... | 3 |
| 4. Задачи..... | 3 |
| 5. Тест-план и стратегия тестирования..... | 3 |
| 5.1. Стратегия..... | 3 |
| 5.2. Тест-кейсы..... | 4 |
| 6. Инструменты..... | 4 |
| 7. Риски процесса тестирования..... | 4 |
| 8. Процесс..... | 4 |
| 9. Результаты..... | 6 |
| 9.1. Баг-репорт..... | 6 |
| 9.2. Тест-кейсы..... | 7 |

Основная информация.

| | |
|-----------------|----------------------------------|
| Заказчик | РТУ МИРЭА |
| Автор | Моисееенкова Виолетта Витальевна |
| Дата | 01.11.2024 |
| Версия | 1.0 |
| Статус | Завершен |

История редактирования.

| Версия | Описание | Автор | Дата |
|---------------|-----------------|--------------|-------------|
| | | | |

1. Введение.

Этот документ описывает методы и подходы к тестированию простого калькулятора, реализованного на языке Java, с использованием фреймворка JUnit 5 для тестирования. Проект демонстрирует базовые принципы модульного тестирования, включая проверку основных арифметических операций (сложение, вычитание, умножение, деление), обработку исключительных ситуаций, таких как деление на ноль, а также использование различных аннотаций и возможностей JUnit 5 для организации и проведения тестов. Проект служит практическим примером применения JUnit 5.

2. Цель.

Целью данного пет-проекта является изучение и практическое применение JUnit 5 для проведения модульного тестирования Java-приложения "Калькулятор".

3. Область тестирования.

1. Основные арифметические операции, такие как сложение, вычитание, умножение и деление целых и дробных чисел.
2. Обработка ошибок: деление на ноль, переполнение, некорректный ввод.
3. Тестирование граничных значений.

4. Задачи.

1. Разработка калькулятора на Java.
2. Написание тест-кейсов с использованием JUnit 5.
3. Запуск тестов и анализ результатов.
4. Исправление ошибок в калькуляторе (при необходимости).
5. Повторное тестирование.
6. Документирование результатов.

5. Тест-план и стратегия тестирования.

5.1. Стратегия.

Unit testing с использованием подхода "белый ящик". Основной фокус – тестирование отдельных методов калькулятора в изоляции друг от друга. Каждый метод, выполняющий арифметическую операцию или обработку ошибок, будет покрыт отдельными тестами.

Подход, который используется - "белый ящик" (white-box). При написании тестов будет учитываться внутренняя структура кода калькулятора. Это позволит проверить все ветви выполнения кода, условия и циклы, добиваясь максимального покрытия.

5.2. Тест-кейсы.

Тест-кейсы будут разработаны для каждой арифметической операции, включая следующие сценарии:

1. Позитивные сценарии;
2. Негативные сценарии;
3. Граничные значения;
4. Особые случаи (операции с 0 и с отрицательными числами).

6. Инструменты.

При реализации проекта «Калькулятор на Java. Создание и тестирование» использовались следующие инструменты: Java, JUnit 5, IntelliJ IDEA, Git.

JUnit 5 — это фреймворк для написания и запуска unit-тестов в Java. Он предоставляет аннотации, методы assert и другие инструменты для создания тестов, проверки результатов и организации тестового кода. JUnit 5 использовался для написания и запуска тестов для калькулятора. С помощью аннотации @Test, определялись тестовые методы и их порядок выполнения.

7. Риски процесса тестирования.

Недостаточное покрытие кода тестами - этот риск означает, что не все части кода калькулятора будут проверены тестами. Это может привести к тому, что ошибки в непротестированных участках кода останутся незамеченными и проявятся уже после выпуска приложения. Для минимизации этого риска необходимо стремиться к 100% покрытию кода тестами, используя инструменты анализа покрытия, и уделять особое внимание сложной логике и граничным случаям.

Неправильно составленные тест-кейсы, например, некорректные тесты могут не выявить существующие ошибки или, наоборот, сигнализировать об ошибках там, где их нет. Важно тщательно продумывать тестовые сценарии, учитывать все возможные входные данные и ожидаемые результаты.

Ограниченные временные ресурсы, то есть недостаток времени может привести к поверхностному тестированию и пропуску важных сценариев.

8. Процесс.

Первым делом мы должны написать класс Calculator, где будут такие методы как — add, subtract, multiply, divide. (Рис. 1) Данный класс будет являться простым калькулятором написанным на Java, именно на его примере мы будем проводить тестирование с использованием JUnit 5.

```

public class Calculator { 10 usages
    public int add(int a, int b){ 1 usage
        return a+b;
    }

    public int subtract(int a, int b){ 1 usage
        return a-b;
    }

    public int multiply(int a, int b){ 1 usage
        return a*b;
    }

    public int divide(int a, int b){ 2 usages
        if (b==0){
            throw new ArithmeticException("Деление на 0 - НЕЛЬЗЯ!!!");
        }
        return a/b;
    }
}

```

(Рис. 1)

Далее создаем отдельную папку test, где будут написаны тесты для каждого метода. В соответствии со стандартной структурой Maven-проекта, эта папка должна располагаться параллельно папке main и содержать директории java, аналогично структуре папки main. Данное разделение позволяет Maven правильно обрабатывать и запускать тесты. После помечаем анотацией @Test наши методы, а в них уже пишем тесты с использованием метода assertEquals(). (Рис. 2, Рис. 3)

```

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

public class CalculatorTest {
    @Test
    void addTest(){
        Calculator calculator = new Calculator();
        assertEquals( expected: 5, calculator.add( a: 2, b: 3));
    }

    @Test
    void subtractTest() {
        Calculator calculator = new Calculator();
        assertEquals( expected: 1, calculator.subtract( a: 4, b: 3));
    }

    @Test
    void multiplyTest() {
        Calculator calculator = new Calculator();
        assertEquals( expected: 6, calculator.multiply( a: 2, b: 3));
    }
}

```

(Рис. 2)

```

@Test
void divideTest() {
    Calculator calculator = new Calculator();
    assertEquals( expected: 2, calculator.divide( a: 6, b: 3));
}

@Test
void divideByZeroTest() {
    Calculator calculator = new Calculator();
    assertThrows(ArithmeticException.class, () -> calculator.divide( a: 5, b: 0));
}

public static void main(String[] args) {
    System.out.println("Тесты сделаны");
}
}

```

(Рис. 3)

После запуска мы получаем результат наших тестов (Рис. 4).

| | | |
|---|--------------------|-------|
| ✓ | CalculatorTest | 61 ms |
| ✓ | divideTest() | 50 ms |
| ✓ | addTest() | 2 ms |
| ✓ | divideByZeroTest() | 7 ms |
| ✓ | multiplyTest() | 1 ms |
| ✓ | subtractTest() | 1 ms |

(Рис. 4)

9. Результаты.

Конечным результатом является созданный калькулятор, прошедший все тесты с использованием JUnit 5.

9.1. Баг-репорт.

1. Название: Деление на ноль не обрабатывается корректно.
2. Описание: При делении на ноль приложение выдает некорректный результат вместо ожидаемого исключения.
3. Шаг воспроизведения: Вызвать метод `calculator.divide(4, 0)`
4. Ожидаемый результат: ArithmeticException
5. Фактический результат: Infinity
6. Серьезность: Высокая
7. Приоритет: Высокий
8. Статус: Открыт

9.2. Тест-кейсы.

| ID | Метод | Входные данные | Ожидаемый результат | Фактический результат |
|-----|--------------------|----------------|--|--|
| T01 | addTest() | 2, 3 | 5 | 5 |
| T02 | subtractTest() | 4, 3 | 1 | 1 |
| T03 | multiplyTest() | 2, 3 | 6 | 6 |
| T04 | divideTest() | 6, 3 | 2 | 2 |
| T05 | divideByZeroTest() | 5, 0 | Вывод сообщения - "Деление на 0 — НЕЛЬЗЯ!!!". Выброс ошибки. | Вывод сообщения - "Деление на 0 — НЕЛЬЗЯ!!!". Выброс ошибки. |