МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра информатики и систем управления

Лабораторная работа №1

по дисциплине

Тестирование ПО

(наименование дисциплины)

РУКОВОДИТЕЛЬ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Скорынин С. С.

(подпись) (фамилия, и.,о.)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Саечников Н.И.

(подпись) (фамилия, и.,о.)

С22-СИБ

(шифр группы)

Работа защищена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_

С оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Нижний Новгород 2025

1. Цель работы

Изучить средства тестирования в Java, разработать набор unit-тестов для калькулятора, обеспечить максимальное покрытие кода тестами с использованием mock-объектов.

2. Постановка задачи

**2. Постановка задачи**

Разработать систему “Калькулятор”, которая:

* Реализует основные арифметические операции: сложение, вычитание, умножение, деление.
* Использует интерфейс для определения операций.
* Имеет класс, который координирует выполнение операций.
* Обладает набором unit-тестов, написанных с использованием JUnit и Mockito, для проверки функциональности и надежности.

3. Теоретические сведения

* **Unit-тестирование:** Метод тестирования программного обеспечения, при котором тестируются отдельные модули исходного кода программы. Целью является проверка корректности работы отдельных компонентов.
* **Используемые инструменты:**
  + **JUnit:** Фреймворк для написания и выполнения тестов в Java. Предоставляет аннотации ( @Test, @BeforeEach, @ParameterizedTest и др.) для определения тестовых методов и методов подготовки к тестам, а также методы для утверждений (assertEquals, assertThrows и др.) для проверки ожидаемых результатов.
  + **Mockito:** Библиотека для создания mock-объектов. Позволяет изолировать тестируемые компоненты от их зависимостей, заменяя их контролируемыми mock-объектами. Используется для проверки взаимодействия компонентов и тестирования различных сценариев.
  + **Java (JDK 11):** Язык программирования и платформа для разработки.
  + **Maven/Gradle:** Инструменты для управления зависимостями и сборки проекта.
* **Объектно-ориентированное программирование (ООП):** Парадигма программирования, основанная на концепциях “объектов”, которые содержат данные и методы для работы с этими данными. Использование ООП-принципов (инкапсуляция, наследование, полиморфизм) способствует созданию модульного, расширяемого и поддерживаемого кода.
* **Интерфейсы и Полиморфизм:** Интерфейсы определяют контракты (методы), которые должны реализовывать классы. Полиморфизм позволяет работать с объектами разных классов через общий интерфейс, что делает код более гибким и позволяет легко добавлять новые типы операций.

4. Архитектура решения

4.1 Структура проекта

calculator/

├── src/

│ ├── main/

│ │ ├── java/

│ │ │ └── com/example/calculator/

│ │ │ ├── Calculator.java

│ │ │ ├── Operation.java

│ │ │ ├── Addition.java

│ │ │ ├── Subtraction.java

│ │ │ ├── Multiplication.java

│ │ │ └── Division.java

│ └── test/

│ ├── java/

│ │ └── com/example/calculator/

│ │ ├── CalculatorTest.java

│ │ ├── AdditionTest.java

│ │ ├── SubtractionTest.java

│ │ ├── MultiplicationTest.java

│ │ └── DivisionTest.java

├── pom.xml

4.2 Диаграмма классов

Operation <<interface>>

├── Addition

├── Subtraction

├── Multiplication

└── Division

Calculator

├── operations: Map<String, Operation>

└── calculate(operand1: double, operand2: double, operator: String): double

5. Реализация

package com.example.calculator;

public interface Operation {

double execute(double operand1, double operand2);

}

package com.example.calculator;

public class Addition implements Operation {

@Override

public double execute(double operand1, double operand2) {

return operand1 + operand2;

}

}

package com.example.calculator;

public class Subtraction implements Operation {

@Override

public double execute(double operand1, double operand2) {

return operand1 - operand2;

}

}

package com.example.calculator;

public class Multiplication implements Operation {

@Override

public double execute(double operand1, double operand2) {

return operand1 \* operand2;

}

}

package com.example.calculator;

public class Division implements Operation {

@Override

public double execute(double operand1, double operand2) {

if (operand2 == 0) {

throw new IllegalArgumentException("Division by zero is not allowed");

}

return operand1 / operand2;

}

}

package com.example.calculator;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class Calculator {  
  
 private Map<String, Operation> operations = new HashMap<>();  
 public Calculator(Map<String, Operation> operations) {  
 this.operations = operations;  
 }  
 public Calculator() {  
 operations.put("+", new Addition());  
 operations.put("-", new Subtraction());  
 operations.put("\*", new Multiplication());  
 operations.put("/", new Division());  
 }  
  
  
 public double calculate(double operand1, double operand2, String operator) {  
 Operation operation = operations.get(operator);  
 if (operation == null) {  
 throw new IllegalArgumentException("Invalid operator: " + operator);  
 }  
 return operation.execute(operand1, operand2);  
 }  
}

6. Тестирование

6.1 Стратегия тестирования

Для обеспечения надежности и корректности работы системы использовались следующие подходы:

1. **Unit-тесты:** Тестирование каждого класса и метода по отдельности для проверки его функциональности в изоляции.
2. **Позитивные тесты:** Проверка корректной работы с валидными входными данными.
3. **Негативные тесты:** Проверка обработки невалидных входных данных (например, деление на ноль, неверные операторы).
4. **Mock-объекты:** Использование Mockito для создания mock-объектов интерфейса Operation, чтобы изолировать тесты Calculator от конкретных реализаций операций и проверить взаимодействие между компонентами.
5. **Параметризованные тесты:** Использование @ParameterizedTest и @CsvSource для уменьшения дублирования кода и тестирования с различными входными данными.

6.2 Типы тестов

Интеграционные тесты (или тесты с реальными объектами):

* CalculatorTest(): Проверка правильности выполнения арифметических операций с использованием реальных реализаций Operation ( Addition, Subtraction и т.д.). Включает в себя:
  + testAddition(): Проверка сложения.
  + testSubtraction(): Проверка вычитания.
  + testMultiplication(): Проверка умножения.
  + testDivision(): Проверка деления.
  + testDivisionByZero(): Проверка обработки деления на ноль.
  + testInvalidOperator(): Проверка обработки невалидного оператора.

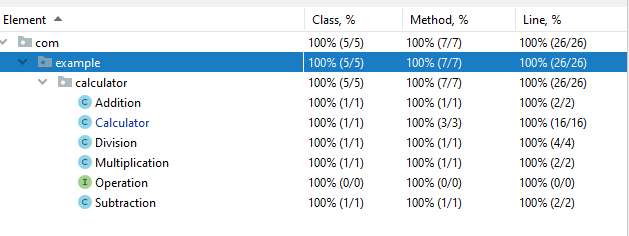
Тесты с Mockito:

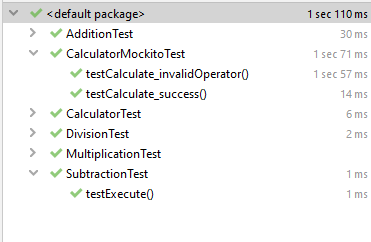
* CalculatorMockitoTest(): Проверка взаимодействия компонентов между собой с использованием mock-объектов. Включает в себя:
  + testCalculate\_success(): Проверка успешного выполнения операции.
  + testCalculate\_invalidOperator(): Проверка обработки неверного оператора (с верификацией, что метод execute у Operation не вызывается).

Unit тесты:

* AdditionTest: Проверка правильности принятия значений и их сложения. Включает в себя:
* testExecute(): Проверка правильности сложения.

Результат покрытия кода:





**Вывод:**

Были изучены средства тестирования в Java, разработаны модульные тесты для калькулятора с достижением максимального покрытия кода и использованы mock-объекты