МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ

УНИВЕРСИТЕТ им. Р.Е.АЛЕКСЕЕВА

Институт радиоэлектроники и информационных технологий

Кафедра Вычислительные системы и технологии

ОТЧЕТ

По лабораторной работе № 1

по дисциплине

Тестирование ПО

РУКОВОДИТЕЛЬ:

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

(подпись)

СТУДЕНТ:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Захаров Д.С.

(подпись)

С22-СИБ

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

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

Нижний Новгород 2025

**Цель работы:** изучить основы разработки модульных тестов.

**Введение:**

Современная разработка программного обеспечения требует не только создания функционального кода, но и обеспечения его надежности и корректности. Одним из ключевых инструментов для достижения этой цели является модульное тестирование (unit testing), которое позволяет проверять отдельные компоненты программы на соответствие ожидаемому поведению.

В данной лабораторной работе рассматриваются основы разработки модульных тестов, их назначение, принципы написания и применение в процессе разработки ПО. Изучение этих аспектов поможет сформировать навыки создания тестов, которые повышают качество кода, упрощают его поддержку и снижают вероятность ошибок в дальнейшем.

**Описание проделанной работы:**

1. **Изучение инструментов тестирования**:

Для тестирования использовался фреймворк JUnit 5 в сочетании с Mockito для создания mock-объектов.

Для анализа покрытия кода применялся JaCoCo, который интегрирован в Maven-проект через pom.xml.

1. **Разработка unit-тестов**:
   * Написаны тесты для всех ключевых классов:
     + BankAccountServiceImplTest — тестирование бизнес-логики сервиса банковских счетов (создание счетов, пополнение, списание, переводы, начисление процентов, заморозка/разморозка счетов).
     + InMemoryTransactionRepositoryTest — проверка корректности работы репозитория транзакций.
     + TransactionTest — тестирование модели транзакции.
   * Использованы различные виды тестов:
     + Параметризованные тесты (например, для проверки недопустимых процентных ставок).
     + Тесты на обработку исключений (например, InsufficientFundsException).
     + Тесты для проверки логирования (с использованием ListAppender из Logback).

### ****Примеры тестов****

#### ****Тест перевода средств (****transfer****)****

@Test

void transfer\_ValidAmount\_TransfersBetweenAccounts() throws InsufficientFundsException {

BankAccount fromAccount = new BankAccount("acc1", "John", 100.0);

BankAccount toAccount = new BankAccount("acc2", "Jane", 50.0);

when(accountRepository.findById("acc1")).thenReturn(fromAccount);

when(accountRepository.findById("acc2")).thenReturn(toAccount);

bankAccountService.transfer("acc1", "acc2", 30.0);

assertEquals(70.0, fromAccount.getBalance()); // 100 - 30 = 70

assertEquals(80.0, toAccount.getBalance()); // 50 + 30 = 80

verify(accountRepository).save(fromAccount);

verify(accountRepository).save(toAccount);

}

#### ****Тест на недостаток средств (****InsufficientFundsException****)****

#### @Test

#### void withdraw\_InsufficientFunds\_ThrowsException() {

#### BankAccount account = new BankAccount("acc1", "John", 30.0);

#### when(accountRepository.findById("acc1")).thenReturn(account);

#### assertThrows(InsufficientFundsException.class,

#### () -> bankAccountService.withdraw("acc1", 50.0));

#### }

1. **Реализация слабосвязанного кода**:

В рамках лабораторной работы было разработано **консольное банковское приложение** на Java, реализующее основные функции работы с банковскими счетами:

* Создание и управление счетами.
* Пополнение и списание средств.
* Переводы между счетами.
* Начисление процентов.
* Заморозка/разморозка счетов.
* Ведение истории транзакций.

Приложение построено по **модульной архитектуре**с использованием принципов **слабой связанности (loose coupling)**и**инверсии зависимостей (Dependency Injection).**

### ****Основные компоненты****

Приложение состоит из следующих ключевых модулей:

|  |  |
| --- | --- |
| **Компонент** | **Назначение** |
| BankAccount | Модель банковского счета (ID, владелец, баланс, статус заморозки). |
| Transaction | Модель транзакции (ID, тип, сумма, время операции). |
| AccountRepository | Интерфейс для работы с хранилищем счетов (CRUD-операции). |
| TransactionRepository | Интерфейс для работы с историей транзакций. |
| BankAccountService | Основной сервисный слой (бизнес-логика операций). |
| BankAccountServiceImpl | Реализация сервиса. |
| InMemoryTransactionRepository | In-memory реализация хранилища транзакций. |

### ****Схема взаимодействия компонентов****

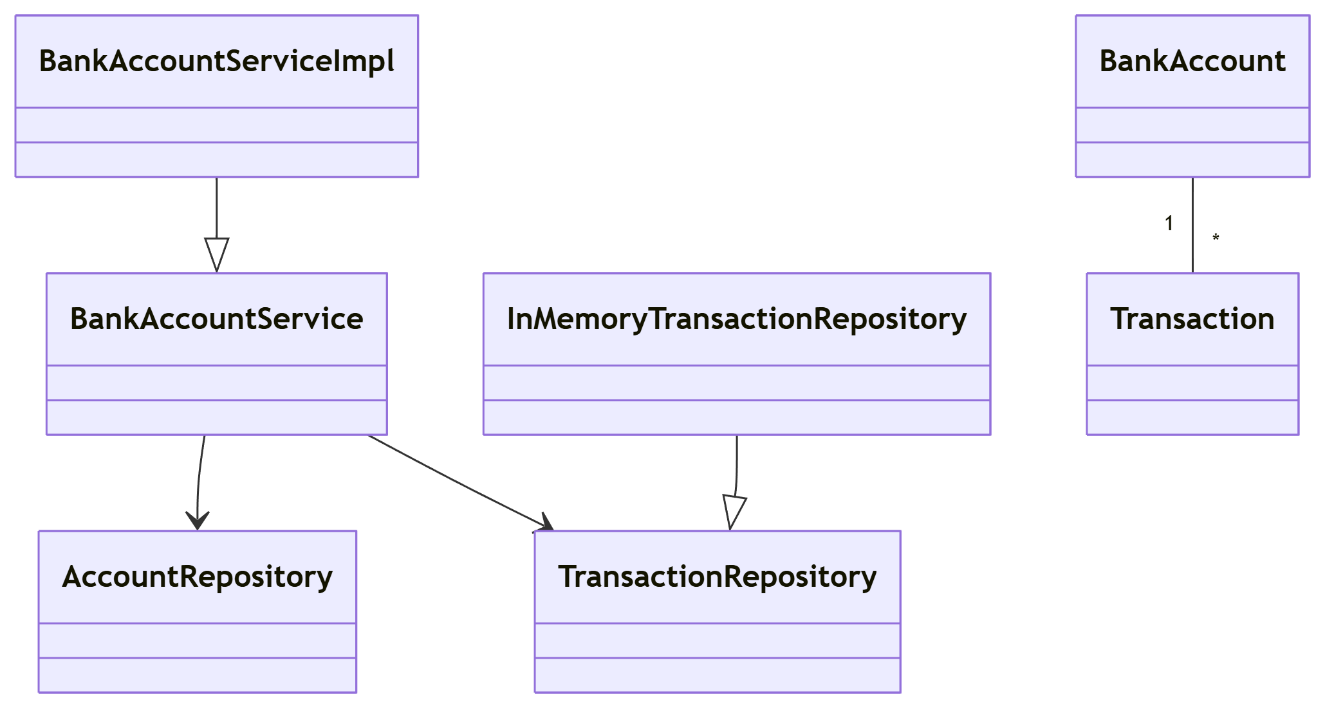
****

Рис 1. (Архитектура банковского приложения)

## **Реализованная функциональность**

### ****1. Управление счетами****

**Создание счета**

**BankAccount createAccount(String ownerName, double initialBalance)**

* + Генерируется уникальный accountId (UUID).
  + Проверяется, что начальный баланс не отрицательный.

**Заморозка/разморозка**

**void freezeAccount(String accountId)**

**void unfreezeAccount(String accountId)**

* + Замороженные счета не участвуют в операциях (пополнение/списание).

### ****2. Финансовые операции****

**Пополнение (**deposit**)**

**void deposit(String accountId, double amount)**

Проверки:

* + - amount > 0 (иначе IllegalArgumentException).
    - Счет не заморожен (иначе IllegalStateException).

**Списание (**withdraw**)**

**void withdraw(String accountId, double amount) throws InsufficientFundsException**

Проверки:

* + - Достаточно ли средств (InsufficientFundsException).
    - Счет не заморожен.

**Перевод (**transfer**)**

**void transfer(String fromAccountId, String toAccountId, double amount)**

Проверки:

* + - fromAccountId != toAccountId.
    - Достаточно средств на исходном счете.

**Начисление процентов (**applyInterest**)**

**void applyInterest(String accountId, double rate)**

Процентная ставка rate > 0.

Создается транзакция типа INTEREST.

### ****3. История транзакций****

* Каждая операция (пополнение, списание, перевод, начисление процентов) фиксируется в TransactionRepository.
* Возможность запроса истории по accountId:

**List<Transaction> getTransactionHistory(String accountId)**

1. **Анализ покрытия кода**:

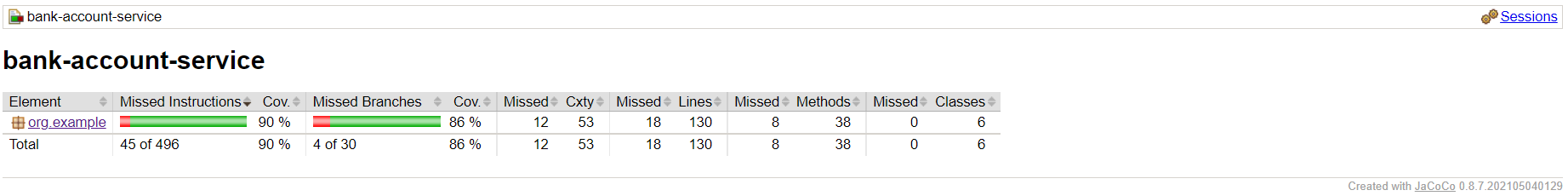
****

Рис 2. (Отчет JaCoCo о прохождении тестов)

Согласно отчету JaCoCo, общее покрытие кода тестами составляет **90%** (пропущено 45 инструкций из 496).

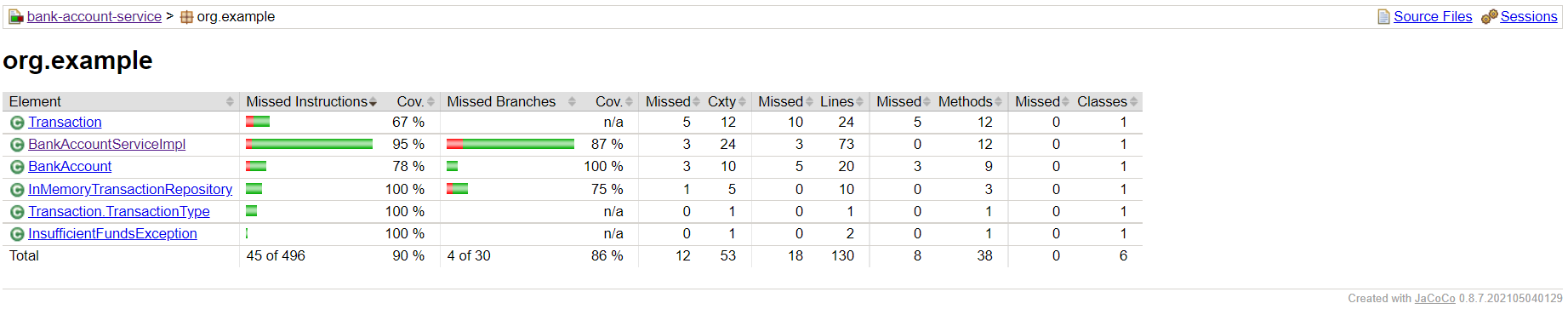


Рис 3. (Покрытие основных классов приложения)

Покрытие ветвей (branches) — **86%** (пропущено 4 ветки из 30).

Наиболее сложные методы (например, обработка транзакций и валидация) покрыты полностью.

**Вывод к работе:**

В ходе работы были успешно выполнены все поставленные задачи:

* Изучены современные инструменты тестирования в Java (JUnit 5, Mockito, JaCoCo).
* Разработаны comprehensive unit-тесты с использованием mock-объектов.
* Обеспечено высокое покрытие кода (90%), включая сложные сценарии.
* Продемонстрирована слабосвязанная архитектура, упрощающая тестирование.

Результаты показывают, что код соответствует требованиям надежности, а тесты эффективно проверяют его корректность.