Введение в Unit-тестирование

Unit-тестирование (модульное тестирование) проверяет работу отдельных частей программы в изоляции. Оно помогает находить ошибки, упрощает отладку и делает код более устойчивым к изменениям. Хорошие тесты должны быть **изолированными** (не зависеть от внешних факторов), **детерминированными** (давать одинаковый результат при каждом запуске), **автоматизированными** (без необходимости ручного вмешательства) и **понятными** (чтобы их можно было легко поддерживать).

Покрытие тестами (Code Coverage) — важный показатель, который показывает, какая часть кода выполняется во время тестирования. Чем выше покрытие, тем меньше вероятность пропущенных ошибок.

Пример Unit-тестов на Java

Рассмотрим простую программу — калькулятор, выполняющий сложение чисел:

Основной код (Calculator.java)

```
public class Calculator {
    // Метод сложения двух чисел
    public int add(int a, int b) {
        return a + b;
    }
}
```

Этот класс содержит метод `add`, который принимает два числа и возвращает их сумму.

Unit-тест (CalculatorTest.java)

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class CalculatorTest {
    @Test // Аннотация указывает, что метод является тестом
    void testAddition() {
        Calculator calculator = new Calculator(); // Создаем объект
калькулятора
        int result = calculator.add(2, 3); // Вызываем метод сложения
        assertEquals(5, result, "2 + 3 должно быть 5"); // Проверяем, что
результат равен 5
    }
}
```

В этом коде:

- Создается объект класса `Calculator`.
- Вызывается метод `add(2, 3)`, который должен вернуть 5.
- Используется `assertEquals(expected, actual, message)`, который проверяет, что возвращенное значение соответствует ожидаемому. Если нет — тест завершится с ошибкой.

Определение покрытия тестами

Покрытие кода тестами показывает, сколько строк кода, методов или ветвей условий было выполнено во время тестирования. Один из популярных инструментов для измерения покрытия — **JaCoCo**.

Подключение JaCoCo к Maven

В файле `pom.xml` нужно добавить плагин:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.jacoco</groupId>
            <artifactId>jacoco-maven-plugin</artifactId>
            <version>0.8.8</version>
            <executions>
                <execution>
                    <goals>
                        <goal>prepare-agent</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

Запуск тестов и генерация отчета о покрытии

Выполняем тесты и создаем отчет:

```
mvn test
mvn jacoco:report
```

После выполнения отчета он будет доступен в `target/site/jacoco/index.html`.

JaCoCo анализирует:

- **Покрытие строк (Line Coverage)** – сколько строк кода было выполнено.
- **Покрытие методов (Method Coverage)** – сколько методов было вызвано.
- **Покрытие ветвей (Branch Coverage)** – сколько логических ветвлений (`if`, `switch`) было пройдено.

Плюсы и минусы Unit-тестирования

Плюсы

- Раннее обнаружение ошибок – тесты помогают выявлять баги на этапе разработки, что снижает стоимость их исправления.
- Упрощение рефакторинга – с тестами можно безопасно изменять код, зная, что существующий функционал не сломан.
- Документирование кода – тесты могут служить примерами использования методов и классов.
- Автоматизация проверки – тесты можно запускать автоматически при каждом изменении кода.
- Повышение надежности – снижается вероятность появления неожиданных ошибок в продакшене.

Минусы

- Затраты времени на написание – требует дополнительного времени и ресурсов.
- Поддержка тестов – при изменении логики приложения тесты тоже нужно обновлять.
- Не заменяют интеграционные тесты – unit-тесты проверяют только отдельные модули, но не гарантируют, что вся система работает корректно.
- Ложное чувство безопасности – хорошее покрытие кода тестами не означает, что в программе нет ошибок.

Задача к выполнению

Требуется разработать программный модуль на Java, реализующий заданный функционал, а также покрыть его unit-тестами с уровнем покрытия не менее **80%** по метрике **Line Coverage** (покрытие строк кода) согласно отчету JaCoCo.

Требования к коду и исполняемым тестам

- Разработанный код должен следовать принципам **чистой архитектуры** и **инкапсуляции**.
- Реализовать класс с основным функционалом.
- Использовать **JUnit 5** для написания тестов.
- Покрытие кода тестами должно быть проверено с помощью **JaCoCo**.
- Код должен быть читаемым и документированным.
- Использовать **Maven** или **Gradle** для автоматизации тестирования и анализа покрытия.

Вариант 1: Управление задачами (Task Manager)

Описание:
Создать класс `TaskManager`, который управляет списком задач. Каждая задача представлена классом `Task`, содержащим название, описание и статус выполнения.

Функциональные требования:

- Создание новой задачи.
- Удаление задачи.
- Получение списка всех задач.
- Изменение статуса задачи (например, "выполняется", "завершено").
- Поиск задачи по названию.

Вариант 2: Калькулятор выражений

Описание:
Реализовать класс `ExpressionCalculator`, который выполняет базовые математические операции.

Функциональные требования:

- Сложение, вычитание, умножение, деление.
- Обработка деления на ноль (выбрасывание исключения).
- Возможность вычислять выражения вида `5 + 3 * 2`.

Вариант 3: Валидация паролей

Описание:
Создать класс `PasswordValidator`, проверяющий надежность пароля.

Функциональные требования:

- Длина пароля не менее 8 символов.
- Должен содержать хотя бы одну цифру.
- Должен содержать хотя бы одну заглавную букву.
- Должен содержать хотя бы один специальный символ (`@, #, !, etc.`).

Вариант 4: Конвертер температур

Описание:
Реализовать класс `TemperatureConverter`, выполняющий конвертацию температур между шкалами Цельсия и Фаренгейта.

Функциональные требования:

- Перевод из Цельсия в Фаренгейт: `F = C * 9/5 + 32`.
- Перевод из Фаренгейта в Цельсий: `C = (F - 32) * 5/9`.

Примеры инструментов для работы

- **Язык:** Java 11+
- **Фреймворк для тестирования:** JUnit 5
- **Система сборки:** Maven / Gradle
- **Проверка покрытия кода:** JaCoCo