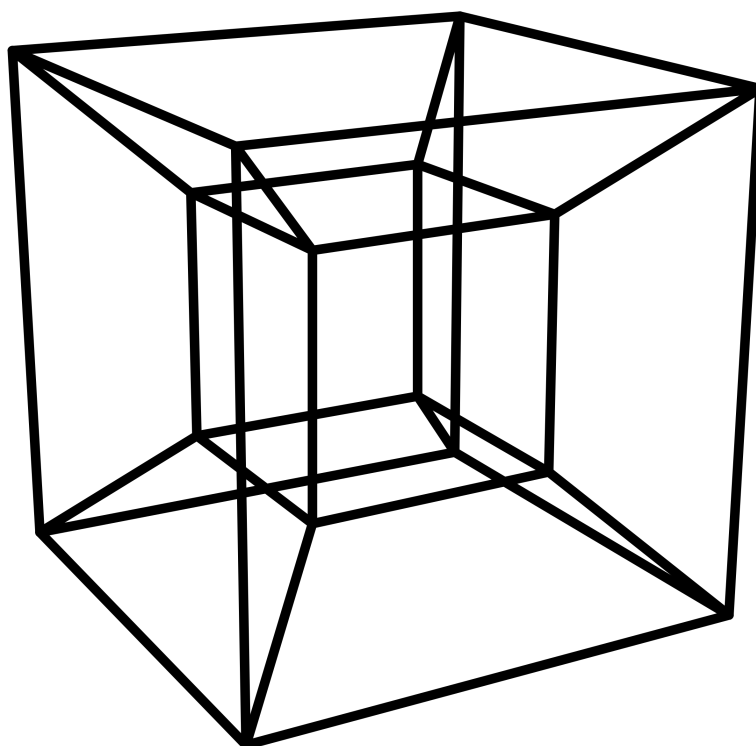


ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК И КИБЕРБЕЗОПАСНОСТИ
ВЫСШАЯ ШКОЛА ПРОГРАММНОЙ ИНЖЕНЕРИИ

ОТЧЕТ ПО МОДУЛЬНОМУ ТЕСТИРОВАНИЮ

АГРЕГАТОР ЦИФРОВЫХ ФИНАНСОВЫХ АКТИВОВ «ТЕССЕРАКТ»

по дисциплине «Технологии разработки качественного программного обеспечения»



Выполнили студенты группы: 5130904/00104:

Почернин В. С.
Шиляев В. С.
Мурзаканов И. М.
Разукрантов В. Е.

Преподаватель:

Маслаков А. П.

Санкт-Петербург
2024

Содержание

1	Постановка задачи	2
1.1	Обязательные требования	2
1.2	Содержание отчета	2
2	Ход работы	2
2.1	Описание выполненной работы, использованных инструментов, применённых техник тест-дизайна	2
2.1.1	Использованные инструменты	2
2.1.2	Примененные техники тест-дизайна	3
2.2	Отчет о прохождении тестов с результатами и оценкой покрытия кода тестами	6
2.2.1	Серверная часть	6
2.3	Описание процедуры расширения тестового набора на примере добавления нового блока, кода, алгоритма, метода	8

1 Постановка задачи

Необходимо выполнить модульное тестирование разработанного программного продукта. Кодовая база всего продукта должна быть покрыта тестами на 80% и более, но не менее 25 тестов на каждого члена команды.

1.1 Обязательные требования

- 1) Тестирование должно проводиться автоматически при сборке проекта с помощью сборщика **Gradle**. Необходимо использовать фреймворк для написания модульных тестов **JUnit**.
- 2) Необходимо применить несколько техник тест-дизайна, таких как классы эквивалентности и граничные условия.

1.2 Содержание отчета

Отчет по модульному тестированию должен содержать:

- 1) Описание выполненной работы, использованных инструментов, примененных техниках тест-дизайна.
- 2) Отчет о прохождении тестов с результатами и оценкой покрытия кода тестами.
- 3) Описание процедуры расширения тестового набора на примере добавления нового блока кода, алгоритма, метода.

2 Ход работы

2.1 Описание выполненной работы, использованных инструментов, применённых техник тест-дизайна

В рамках проведенной работы, код клиентской и серверной частей приложения был покрыт модульными тестами.

2.1.1 Используемые инструменты

Были использованы следующие инструменты:

- **JUnit** - фреймворк для языков программирования **Java** и **Kotlin**, предназначенный для автоматического unit-тестирования. Данный фреймворк позволяет удобно создавать, организовывать и выполнять тесты, благодаря широкому набору аннотаций и встроенной поддержки в популярных IDE. Для серверной части применялся **JUnit5**, для клиентской части - **JUnit4**.
- **AssertJ** - библиотека утверждений для **Java**, обеспечивающая более выразительный и удобный синтаксис для тестовых утверждений, чем встроенные утверждения в **JUnit**. Она делает код более читаемым, а тесты более гибкими.
- **Mockito** - фреймворк для создания заглушек в **Java**. Он используется для имитации поведения компонентов системы, что необходимо для изоляции тестируемого компонента от внешних зависимостей и воздействий. Это упрощает написание модульных тестов и повышает их надежность.
- **MockK** - аналогичный фреймворк для создания заглушек, но уже для языка **Kotlin**.
- **IntelliJ IDEA Coverage** - встроенный в среду разработки **IntelliJ IDEA** инструмент, позволяющий анализировать покрытие кода тестами.
- **Kover** - аналогичный инструмент, но уже для языка **Kotlin**.

2.1.2 Примененные техники тест-дизайна

Тест-дизайн - это процесс разработки техник и методов тестирования. Главная задача тест-дизайна - разработать сценарии, которые позволяют протестировать максимальное количество функций за минимальное время. Таким образом, разрабатывается тестовая документация, опирающаяся на общие принципы и логику тестирования с поправкой на особенности продукта. Суть всех техник тест-дизайна - оптимизировать процесс тестирования.

Нами были применены следующие техники тест-дизайна:

- **Классы эквивалентности:** данная техника тестирования состоит в том, что все тестовые кейсы группируются и разделяются на некие эквивалентные классы. При этом классы одной тестовой группы ведут себя одинаково. То есть, если система корректно обработает одно значение из класса, то она корректно обработает и все остальные значения из этого класса. Согласно одному из принципов тестирования - «полное тестирование программы невозможно, или займет недопустимо длительное время, поскольку в этом случае необходимо проверять слишком много комбинаций тестовых данных».

Данная техника позволяет получить четкие результаты за ограниченное время, улучшает качество тест-кейсов, устраняя избыточность, при этом покрывая тестовые сценарии.

Приведем пример теста клиента:

```
@Test
fun `isPasswordValid should return true for valid passwords`() {
    assertTrue(Validation.isPasswordValid("password123"))
    assertTrue(Validation.isPasswordValid("password!"))
}

@Test
fun `isPasswordValid should return false for passwords with length less than 6`() {
    assertFalse(Validation.isPasswordValid("pa123"))
}

@Test
fun `isPasswordValid should return false for passwords with length more than 30`() {
    assertFalse(Validation.isPasswordValid("password01password01password012"))
}

@Test
fun `isPasswordValid should return false for passwords without digits or special chars`() {
    assertFalse(Validation.isPasswordValid("password"))
}

@Test
fun `isPasswordValid should return false for passwords without letters`() {
    assertFalse(Validation.isPasswordValid("01234567"))
}

@Test
fun `isPasswordValid should return false for passwords with invalid chars`() {
    assertFalse(Validation.isPasswordValid("пароль123"))
}
```

В данном примере тестируется корректность работы функции, проверяющей правильность введенного пароля. Вместо того, чтобы тестировать всевозможные комбинации паролей, мы разбиваем их на классы эквивалентности, такие как:

- Верный пароль;
- Слишком короткий пароль;
- Слишком длинный пароль;
- Пароль без цифр или специальных символов;

- Пароль без букв;
 - Пароль с некорректными символами;
- **Граничные условия:** при использовании данной техники тестирования, основное внимание уделяется значениям на границах допустимого диапазона, поскольку зачастую ошибки возникают именно в граничных точках - такая проверка помогает их быстро находить. Граничные значения - это такие места, в которых один класс эквивалентности переходит в другой. Это значения на границе допустимого диапазона входных данных, которые могут привести к изменению поведения программы.
При использовании данной техники на каждой границе диапазона следует проверить по три значения:
 - Граничное значение;
 - Значение перед границей;
 - Значение после границы;

Приведем пример теста сервера:

```
@Test
void givenTooLittleAmount_whenCreateDiversification_thenTooLittleAmountThrown() {
    CreateDiversificationRequestDto request = requestWithAmount(99L);
    List<Asset> assets = getAssetsWithMinPrice100();
    when(assetRepository.findAll()).thenReturn(assets);

    assertThatThrownBy(() -> diversificationService.createDiversification(request))
        .isInstanceOf(TesseractException.class)
        .extracting("errorType")
        .isEqualTo(TesseractErrorType.TOO_LITTLE_AMOUNT);
}

@Test
void givenMinPossibleAmount_whenCreateDiversification_thenDiversificationIsCreated() {
    CreateDiversificationRequestDto request = requestWithAmount(100L);
    List<Asset> assets = getAssetsWithMinPrice100();
    when(assetRepository.findAll()).thenReturn(assets);

    diversificationService.createDiversification(request);

    verify(diversificationRepository).save(any(Diversification.class));
}

@Test
void givenCorrectAmount_whenCreateDiversification_thenDiversificationIsCreated() {
    CreateDiversificationRequestDto request = requestWithAmount(500_000L);
    List<Asset> assets = getAssetsWithMinPrice100();
    when(assetRepository.findAll()).thenReturn(assets);

    diversificationService.createDiversification(request);

    verify(diversificationRepository).save(any(Diversification.class));
}

@Test
void givenMaxPossibleAmount_whenCreateDiversification_thenDiversificationIsCreated() {
    CreateDiversificationRequestDto request = requestWithAmount(1_000_000_000L);
    List<Asset> assets = getAssetsWithMinPrice100();
    when(assetRepository.findAll()).thenReturn(assets);

    diversificationService.createDiversification(request);
}
```

```

        verify(versificationRepository).save(any(Diversification.class));
    }

    @Test
    void givenTooBigAmount_whenCreateDiversification_thenTooBigAmountThrown() {
        CreateDiversificationRequestDto request = requestWithAmount(1_000_000_001L);
        List<Asset> assets = getAssetsWithMinPrice100();
        when(assetRepository.findAll()).thenReturn(assets);

        assertThatThrownBy(() -> diversificationService.createDiversification(request))
            .isInstanceOf(TesseractException.class)
            .extracting("errorType")
            .isEqualTo(TesseractErrorType.TOO_BIG_AMOUNT);
    }

```

Для создания диверсификаций у нас работает следующая логика: сумма диверсификации не должна быть меньше, чем стоимость самого дешевого актива и не должна быть больше, чем определенная граница (10 миллионов рублей).

Для того, чтобы применить технику граничных условий - мы тестируем 5 возможных случаев:

- Слишком маленькая сумма;
- Минимально возможная сумма;
- Средняя сумма;
- Максимально возможная сумма;
- Слишком большая сумма;

Таким образом, мы проверяем правильность работы алгоритма на границах диапазонов.

- **Попарное тестирование:** эта методика тестирования, при которой создаются тестовые случаи для проверки комбинаций каждой пары входных параметров. Это помогает значительно сократить количество тестов при сохранении высокого уровня покрытия, поскольку исследования показывают, что большинство ошибок вызывается взаимодействием между парой параметров.

Приведем пример теста сервера:

```

@Test
public void givenVVV_whenValidateFields_thenSuccess() {
    String login = "correctLogin";
    String email = "correct@email.com";
    String password = "qwe123";

    assertThatNoException().isThrownBy(() ->
        FieldValidator.validateFields(login, email, password));
}

@Test
public void givenVII_whenValidateFields_thenThrow() {
    String login = "correctLogin";
    String email = "incorrectemail.com";
    String password = "123";

    assertThatThrownBy(() -> FieldValidator.validateFields(login, email, password))
        .isInstanceOf(TesseractException.class);
}

@Test
public void givenIIV_whenValidateFields_thenThrow() {
    String login = "1";

```

```

String email = "incorrectemail.com";
String password = "qwe123";

assertThatThrownBy(() -> FieldValidator.validateFields(login, email, password))
    .isInstanceOf(TesseractException.class);
}

@Test
public void givenIVI_whenValidateFields_thenThrow() {
    String login = "1";
    String email = "correct@email.com";
    String password = "123";

    assertThatThrownBy(() -> FieldValidator.validateFields(login, email, password))
        .isInstanceOf(TesseractException.class);
}

```

Мы тестируем метод, который проверяет корректность трех полей - логина, адреса электронной почты и пароля. Если брать полную таблицу состояний - мы будем должны написать 8 тестов:

	login	email	password
1	valid	valid	valid
2	valid	valid	invalid
3	valid	invalid	valid
4	valid	invalid	invalid
5	invalid	valid	valid
6	invalid	valid	invalid
7	invalid	invalid	valid
8	invalid	invalid	invalid

Однако, используя технику попарного тестирования, мы можем сократить количество тестов до 4-х:

	login	email	password
1	valid	valid	valid
2	valid	invalid	invalid
3	invalid	invalid	valid
4	invalid	valid	invalid

2.2 Отчет о прохождении тестов с результатами и оценкой покрытия кода тестами

2.2.1 Серверная часть

Для серверной части было написано **95** тестов, которые покрыли **81%** кода:

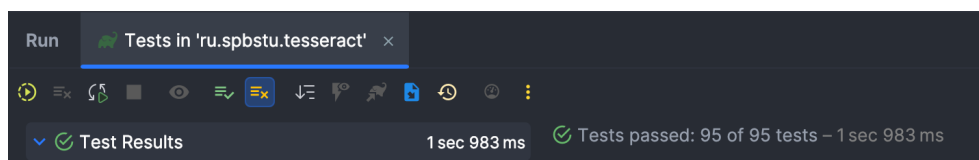
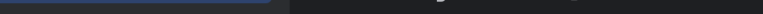


Рис. 1: Тесты серверной части

Coverage Tests in 'ru.spbstu.tesseract' x			
Element ^	Class, %	Method, %	Line, %
<div> <div>ru.spbstu.tesseract</div> <div> <div>configuration</div> <div>controller</div> <div>dto</div> <div>entity</div> <div>exception</div> <div>filter</div> <div>repository</div> <div>service</div> <div>utils</div> <div>TesseractApplication</div> </div> </div>	90% (50/55)	83% (236/2...	81% (601/741)
> configuration	100% (2/2)	90% (9/10)	95% (22/23)
> controller	100% (4/4)	100% (16/16)	100% (55/55)
> dto	86% (19/22)	81% (77/95)	69% (101/145)
> entity	100% (15/15)	90% (94/104)	94% (217/229)
> exception	100% (4/4)	94% (18/19)	95% (58/61)
> filter	100% (1/1)	50% (1/2)	6% (1/16)
> repository	100% (0/0)	100% (0/0)	100% (0/0)
> service	80% (4/5)	51% (15/29)	65% (117/180)
> utils	100% (1/1)	100% (6/6)	96% (30/31)
TesseractApplication	0% (0/1)	0% (0/1)	0% (0/1)

Рис. 2: Покрывание кода тестами серверной части

Для серверной части было написано **55** тестов, которые покрыли **91.8%** кода:



✓ Tests passed: 55 of 55 tests – 2 sec 581 ms

✓ Test Results 2 sec 581 ms

Executing tasks: [testReleaseUnitTest]

Рис. 3: Тесты клиентской части

Current scope: app | all classes

app: Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	82.1% (69/84)	54.8% (125/228)	70% (105/150)	91.8% (356/388)	92% (2944/3201)

Coverage Breakdown

Package 📦	Class, %	Method, %	Branch, %	Line, %	Instruction, %
ru.tesseract	100% (3/3)	57,1% (4/7)		100% (5/5)	100% (64/64)
ru.tesseract.api	100% (9/9)	100% (13/13)	75% (6/8)	100% (44/44)	96,9% (339/350)
ru.tesseract.assets.domain	100% (7/7)	100% (25/25)		100% (67/67)	100% (199/199)
ru.tesseract.assets.ui	100% (6/6)	53,8% (7/13)	75% (3/4)	92,3% (24/26)	92,6% (188/203)
ru.tesseract.diversifications.domain	77,8% (7/9)	74,1% (20/27)	50% (1/2)	85,4% (41/48)	90,3% (233/258)
ru.tesseract.diversifications.ui	77,8% (7/9)	33,3% (8/24)	33,3% (4/12)	93,5% (29/31)	92,1% (314/341)
ru.tesseract.login.api	100% (4/4)	100% (4/4)		100% (4/4)	100% (18/18)
ru.tesseract.login.ui	73,3% (11/15)	29,2% (14/48)	61,5% (32/52)	94,1% (64/68)	93,9% (740/788)
ru.tesseract.settings.data	0% (0/3)	0% (0/7)		0% (0/7)	0% (0/49)
ru.tesseract.settings.ui	72,7% (8/11)	29,7% (11/37)	60,7% (17/28)	86,5% (45/52)	86,1% (439/510)
ru.tesseract.ui	85,7% (6/7)	81% (17/21)	95,5% (42/44)	90,6% (29/32)	97,1% (374/385)
ru.tesseract.ui.theme	100% (1/1)	100% (2/2)		100% (4/4)	100% (36/36)

generated on 2024-02-19 21:55

Рис. 4: Покрытие кода тестами клиентской части

2.3 Описание процедуры расширения тестового набора на примере добавления нового блока, кода, алгоритма, метода

Для того, чтобы добавить новый тест в код приложения в обоих случаях (клиент, сервер) достаточно в файле, предназначенном для тестов создать функцию, помеченную аннотацией `@Test` из фреймворка JUnit.

Далее следует выполнить действия, которые составляют суть теста и завершить тест необходимыми проверками.