МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное
образовательное учреждение высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ПРАКТИЧЕСКИМ РАБОТАМ

«Паттерны проектирования»

Направление подготовки 09.03.04 «Программная инженерия»
Направленность (профиль) «Разработка и сопровождение программного обеспечения»
Квалификация выпускника: бакалавр

Ставрополь
2026

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящая методические рекомендации предназначены для проведения лабораторных работ по дисциплине «Паттерны проектирования», изучаемой студентами направлений, связанных с разработкой программного обеспечения, программной инженерией и архитектурой информационных систем.

Цель дисциплины:

Сформировать у студентов способность проектировать и эволюционировать архитектуру программных систем среднего и крупного масштаба путём осознанного выбора, комбинирования и рефакторинга кода с применением современных паттернов проектирования (GoF, архитектурные, интеграционные, распределённые), а также критической оценки компромиссов (trade-offs) и технического долга.

Задачи дисциплины:

1. Углубить знания о порождающих, структурных и поведенческих паттернах GoF до уровня их вариативности, композиции и промышленного применения в крупных системах.

2. Освоить архитектурные паттерны высокого уровня (Layered, Ports & Adapters, CQRS, Event Sourcing, Saga, Circuit Breaker) в контексте монолитных и модульных систем (без микросервисов).

3. Научить выявлять анти-паттерны (God Object, Lava Flow, Inner Platform, Stairway to Heaven) и проводить рефакторинг к паттернам.

4. Сформировать навык количественной и качественной оценки применимости паттерна (сложность vs гибкость, производительность vs поддерживаемость).

5. Освоить паттерны тестируемости (Test Double, Dependency Injection для тестов, Object Mother) и встраивание их в дизайн.

6. Применить паттерны для асинхронной обработки и распределённой согласованности (на уровне одного процесса с эмуляцией распределённости).

7. Заложить базу для последующих курсов по высоконагруженным системам и микросервисам

Методические рекомендации включают 18 лабораторных работ, которые логически разделены на четыре блока:

Блок 1. Анализ и рефакторинг legacy-кода (ЛР 1)

Знакомство с анти-паттернами и «запахами» кода, методами их выявления и устранения.

Блок 2. Порождающие и структурные паттерны GoF (ЛР 2–5)

Рефакторинг с использованием Factory Method, Abstract Factory, Builder, Decorator, Proxy, Composite. Формирование навыков гибкого создания объектов и динамического расширения поведения.

Блок 3. Поведенческие паттерны и архитектурные приёмы (ЛР 6–10)

Strategy, State, Command, Observer, Chain of Responsibility, а также архитектурная реорганизация к Clean Architecture, Unit of Work и Repository.

Блок 4. Архитектурные, интеграционные паттерны и тестируемость (ЛР 11–17)

Test Doubles, Object Mother, Domain Events, CQRS, Event Sourcing, Saga, Circuit Breaker, Retry. Формирование навыков построения надёжных, тестируемых и событийно-ориентированных систем.

Блок 5. Итоговый проект (ЛР 18)

Комплексное применение не менее шести изученных паттернов в одном проекте, защита архитектурных решений с анализом компромиссов.

Лабораторная работа №1. Анализ legacy-кода и выявление анти-паттернов

Цель работы: очистить и агрегировать навыки распознавания анти-паттернов и «запахов» кода в монолитном приложении, а также предложить направления рефакторинга с использованием паттернов проектирования.

Задачи:

1. Провести статический анализ предоставленного legacy-кода.
2. Идентифицировать анти-паттерны (God Object, Lava Flow, Inner Platform, Stairway to Heaven и др.) и «запахи» кода (длинные методы, жёсткие зависимости, дублирование).
3. Классифицировать выявленные проблемы по типам и оценить критичность каждой.
4. Предложить кандидатов на замену фрагментов кода паттернами GoF или архитектурными паттернами.
5. Составить отчёт о результатах анализа с обоснованием выбора решений.

Теоретическая часть

Анти-паттерны проектирования

Анти-паттерн (anti-pattern) – это шаблон проектирования, который на первый взгляд кажется решением проблемы, но на практике приводит к негативным последствиям: усложнению поддержки, снижению производительности, накоплению технического долга.

Основные анти-паттерны для анализа legacy-кода:

1. **God Object (Божественный объект)** – объект, который «знает слишком много» или «делает слишком много». Он аккумулирует в себе функциональность, которая должна быть распределена между несколькими классами. Признаки: огромный класс с десятками методов, поля для несвязанных сущностей, частые изменения по разным причинам.

2. **Lava Flow (Лавовый поток)** – мёртвый или плохо понятый код, который остаётся в системе, потому что разработчики боятся его удалять. Признаки: закомментированные блоки, неиспользуемые переменные и методы, код, который «работает, но никто не знает зачем».
3. **Inner Platform (Внутренняя платформа)** – система, которая воспроизводит внутри себя возможности используемой платформы (например, собственная реализация работы со строками или коллекциями). Признаки: избыточные абстракции, ручное управление тем, что уже есть в фреймворке.
4. **Stairway to Heaven (Лестница в небо)** – чрезмерно длинная цепочка вызовов методов или наследования, где каждый метод делегирует следующему, не добавляя полезной логики. Признаки: глубокие иерархии (более 4–5 уровней), методы из одного-двух вызовов.
5. **Spaghetti Code (Спагетти-код)** – код с запутанной, хаотичной структурой потока управления (много goto, исключений, условных переходов). Признаки: сложно проследить логику, изменения в одном месте ломают другое.
6. **Copy-Paste Programming** – дублирование кода с незначительными изменениями вместо выделения общего абстракции.

«Запахи» кода (Code Smells)

- **Длинный метод (Long Method)** – метод, который не помещается на одном экране, пытается решить несколько задач.
- **Большой класс (Large Class)** – класс с избыточным количеством полей и методов.
- **Жёсткая зависимость (Tight Coupling)** – класс напрямую создаёт свои зависимости (через new) вместо получения их извне.
- **Комментарии как костыль** – комментарии, объясняющие, почему код работает странно, вместо того чтобы сделать код понятным.

- **Временное поле (Temporary Field)** – поле объекта, которое используется только в определённых сценариях и может оставаться неинициализированным.

Связь анти-паттернов с паттернами GoF

Анти-паттерн	Возможный паттерн для рефакторинга
God Object	Facade (для интерфейса), Strategy (для поведения), Singleton (для отдельных сервисов, но осторожно)
Длинные методы	Template Method, Strategy, Command
Жёсткие зависимости	Dependency Injection, Abstract Factory, Factory Method
Copy-Paste	Template Method, Strategy, Decorator
Spaghetti Code	State, Chain of Responsibility, Command

Методика выполнения работы

Шаг 1. Обзор кода

Получите индивидуальный вариант legacy-кода. Бегло ознакомьтесь со структурой: количество классов, методов, зависимостей. Определите «самые толстые» классы и самые длинные методы.

Шаг 2. Выявление симптомов

Используйте контрольный список для поиска анти-паттернов:

- **God Object**: класс имеет >20 полей или >15 публичных методов, методы различных доменных областей, содержит static поля для глобального состояния.
- **Lava Flow**: наличие закомментированного кода, // TODO: remove, // FIXME: legacy, неиспользуемые импорты.
- **Stairway to Heaven**: цепочка вызовов a().b().c().d() или наследование глубже 4 уровней.

- **Inner Platform**: ручное управление строками, коллекциями, рефлексия для простых задач.
- **Copy-Paste**: два или более похожих блока кода (отличаются только именами переменных).

Шаг 3. Заполнение таблицы анализа

Для каждого выявленного проблемного места заполните таблицу:

ID	Место (класс, метод, строка)	Анти-паттерн / запах	Проявление	Критичность (1–5)	Кандидат на паттерн

Шаг 4. Визуализация зависимостей

Составьте упрощённую диаграмму классов (можно схематично от руки), чтобы увидеть центральные «узлы» зависимостей. Отметьте классы, от которых зависит более 5 других.

Шаг 5. Формулировка предложений

Для трёх самых критичных проблем напишите небольшой псевдокод рефакторинга:

- Какой паттерн предлагается.
- Как изменится структура классов.
- Какие компромиссы возникают (например, увеличение числа классов vs улучшение тестируемости).

Пример выполнения задания

Исходный фрагмент

```
public class OrderProcessor {
    private DatabaseConnection dbConn;
    private EmailSender email;
    private Logger logger;
```

```
    private Configuration config;
    private PaymentGateway payment;
    private InventoryService inventory;
    private AddressValidator addressValidator;
    private DiscountCalculator discountCalc;
    private InvoiceGenerator invoiceGen;
    private AuditService audit;

    public void processOrder(Order order) {
        // validation
        if (order.getCustomer() == null) return;
        if (order.getItems().size() == 0) return;
        if (!addressValidator.isValid(order.getAddress())) return;

        // discount
        double discount = 0;
        if (order.getCustomer().isPremium()) discount = 0.1;
        else if (order.getTotal() > 1000) discount = 0.05;
        order.setDiscount(discount);

        // payment
        boolean paid = payment.charge(order.getTotal() * (1-discount),
order.getPaymentToken());
        if (!paid) {
            logger.log("Payment failed");
            email.sendFailure(order.getCustomer().getEmail());
            return;
        }

        // inventory
```

```
        for (Item i : order.getItems()) {
            if (!inventory.reserve(i.getId(), i.getQty())) {
                logger.log("Out of stock: " + i.getName());
                payment.refund(order.getTotal() * (1-discount));
                return;
            }
        }

        // invoice
        String invoiceHtml = invoiceGen.generate(order);
        dbConn.saveOrder(order);
        email.sendInvoice(order.getCustomer().getEmail(), invoiceHtml);
        audit.record("ORDER_COMPLETED", order.getId());
        logger.log("Order " + order.getId() + " processed");
    }

    // ещё 10+ методов: cancelOrder, updateOrder, getOrderStatus, calculateTax...
}
```

Результаты анализа

ID	Место	Анти-паттерн	Проявление	Критич.	Кандидат
1	OrderProcessor	God Object	12 полей-зависимостей, методы разных зон ответственности	5	Разделение на сервисы: PaymentService, InventoryService, OrderOrchestrator с паттерном Facade

ID	Место	Анти-паттерн	Проявление	Критич.	Кандидат
2	processOrder()	Длинный метод	50+ строк, несколько этапов: валидация, скидка, оплата, резерв, инвойс	4	Template Method или Chain of Responsibility для этапов обработки
3	Жёсткие зависимости через new? (здесь поля)	Tight Coupling (прямое создание зависимостей не показано, но вероятно в реальном коде)	3	Dependency Injection через конструктор + Abstract Factory для создания сервисов	
4	Повторяющиеся проверки if (x == null) return	Guard Clauses без структуры	2	Null Object Pattern	

Предложение по рефакторингу (шаблонный вариант)

Применить **паттерн Chain of Responsibility** для этапов обработки заказа:

```
public interface OrderHandler {
   boolean handle(OrderContext context);
   OrderHandler setNext(OrderHandler next);
}
```

public class ValidationHandler implements OrderHandler { ... }

public class DiscountHandler implements OrderHandler { ... }

public class PaymentHandler implements OrderHandler { ... }

public class InventoryHandler implements OrderHandler { ... }

Компромиссы: код станет более гибким (легко добавить новый этап), но увеличится количество классов (с 1 до 5+) и незначительно снизится производительность за счёт косвенных вызовов.

Индивидуальные задания

Вариант 1. Класс ReportGenerator (500+ строк). Содержит методы: generatePDF, generateHTML, generateCSV, sendReport, saveReport, loadTemplate, mergeData, applyStyles, calculateTotals, createChart, addWatermark. Внутри – дублирование кода для разных форматов (похожие циклы по данным). Присутствуют статические методы и поля.

Вариант 2. Система обработки документов: класс DocumentWorker содержит логику импорта из XML, JSON, CSV, ручной парсинг строк, замену тегов, проверку прав доступа, логирование, отправку на печать. Методы по 100–150 строк. Есть закомментированные реализации старых форматов.

Вариант 3. Игровой движок: класс GameEngine управляет рендерингом, физикой, звуком, вводом, сетевыми сообщениями и ИИ. Глубокое наследование: Entity → LivingEntity → Character → Player → NetworkPlayer → BuffedPlayer (6 уровней). Метод update() вызывает updatePhysics()->updateAI()->updateNetwork()->updateSound().

Вариант 4. Класс DataProcessor с методами: loadFromDB, filterByDate, filterByCategory, sortByPrice, sortByName, groupByRegion, exportToExcel, exportToJSON, aggregateSum, aggregateAvg, applyPagination. Все методы статические, оперируют глобальным List<Map<String, Object>> data.

Вариант 5. Класс UserSessionManager – синглтон с полями: currentUser, permissionsCache, preferences, lastActivity, cartItems, orderHistory,

notificationSettings, loggedDevices. Методы: login, logout, checkPermission, updatePref, addToCart, checkout, getHistory, pushNotification, killSession.

Вариант 6. Фрагмент из биллинговой системы: метод calculateBill содержит 15 вложенных if-else для разных типов клиентов, тарифов, сезонных скидок, промокодов, кэшбэка, бонусов и региональных налогов. Есть копипаста расчёта НДС (4 раза).

Вариант 7. Класс NetworkManager сам реализует пул соединений (ручное управление списком сокетов), буферизацию, сериализацию, сжатие и протокол поверх TCP. При этом в проекте используется Apache HttpClient и GZIPOutputStream, но код не обращается к ним.

Вариант 8. Класс Configuration содержит 40 полей (строка-значение) и методы getString, getInt, getBoolean, getList, а также парсинг XML, JSON, .properties файлов. Любое изменение требует правки в 5+ методах.

Вариант 9. Код аутентификации: метод authenticate вызывает checkBlacklist() -> hmacSHA256() -> getSaltFromDB() -> compareHash() -> updateLastLogin() -> logAccess() -> sendWelcomeEmail() -> check2FA() -> issueToken(). Каждый вызов – отдельный класс-сервис, но все они создаются внутри метода через new.

Вариант 10. Класс Order (сущность) содержит бизнес-логику: calculateTax, validate, applyDiscount, ship, cancel, refund, archive. Поля: все атрибуты заказа, плюс tempBuffer, processingFlag, cachedTotal (временные поля для одного метода).

Вариант 11. Код миграции базы данных: класс DataMigrator с 3000 строками и единственным методом migrate(). Внутри – последовательность SQL-запросов, преобразований данных в Java-циклах, вызовов внешних API, с разбросанными try-catch(Throwable).

Вариант 12. Класс UIHelper статический. Содержит методы для отрисовки кнопок, таблиц, текстовых полей, валидации форм, HTTP-запросов, парсинга дат, логирования и шифрования. Любой тест требует загрузки всей среды UI.

Вариант 13. Класс ProductService и ProductService2 – почти идентичные копии (отличаются названиями методов и типом ID: int vs String). Есть общий родитель BaseService, но он пустой.

Вариант 14. Система уведомлений: класс Notifier отправляет email, sms, push, telegram, slack. Для каждого типа своя реализация, но код подготовки сообщения дублируется (шаблоны, подстановка имени пользователя, обработка ошибок). Добавление нового канала требует правки 5 методов.

Вариант 15. Класс Scheduler содержит методы runDailyReport, runHourlySync, runMonthlyCleanup. Внутри каждого – повторяющаяся логика: открытие соединения -> выполнение -> логирование -> закрытие -> обработка исключений -> уведомление администратора.

Требования к отчёту

Отчёт должен быть оформлен в электронном виде (PDF или DOCX) и содержать следующие разделы:

1. **Титульный лист** с названием работы, ФИО студента, номером варианта.
2. **Цель и задачи работы** (выполняется по шаблону из начала документа).
3. **Результаты анализа** – таблица выявленных анти-паттернов и запахов кода с указанием местоположения и критичности.
4. **Диаграмма зависимостей** (схематичная, можно в виде текстового описания или картинки, вставленной в отчёт).
5. **Предложения по рефакторингу** – для 3–5 проблемных мест с описанием:
   - Какой паттерн предлагается.
   - Псевдокод или UML-схема нового дизайна.
   - Обоснование выбора (какие компромиссы учтены).
6. **Вывод** – краткое резюме о состоянии кода и основных направлениях улучшения.

Объём отчёта: не менее 3 страниц без учёта листингов.

Контрольные вопросы

1. Чем анти-паттерн отличается от «запаха» кода? Приведите примеры.
2. Какие основные признаки God Object вы обнаружили в своём варианте?
3. Почему паттерн Facade может помочь при рефакторинге God Object, но не решает проблему полностью?
4. В каком случае «длинный метод» не является проблемой (контрпример)?
5. Какой анти-паттерн чаще всего приводит к нарушению принципа единственной ответственности (SRP)?
6. Почему решение «разбить длинный метод на несколько маленьких» без изменения структуры данных часто неэффективно?
7. Что такое технический долг, и как наличие анти-паттернов его увеличивает?
8. Назовите три паттерна GoF, которые можно применить для устранения жёстких зависимостей.
9. Как оценить компромисс «сложность vs гибкость» при замене копипасты на паттерн Template Method?
10. Почему в реальных проектах Lava Flow редко удаляют полностью? Какие есть стратегии работы с таким кодом?

Лабораторная работа №2. Рефакторинг порождающих паттернов: Factory Method + Abstract Factory

Цель работы: заменить прямые вызовы конструкторов на фабричные методы и абстрактные фабрики для обеспечения гибкости создания и поддержки разных типов продуктов (семейств) без изменения клиентского кода.

Задачи:

1. Проанализировать существующий код с жёсткими зависимостями на конкретные классы продуктов.
2. Реализовать паттерн **Factory Method** для отдельного типа продуктов, устранив прямые вызовы new.
3. Реализовать паттерн **Abstract Factory** для семейства взаимосвязанных продуктов.
4. Обеспечить возможность динамической смены фабрик (например, через конфигурацию или параметры среды).
5. Написать модульные тесты, демонстрирующие замену продуктов без изменения клиента.
6. Оценить компромиссы: увеличение количества классов versus гибкость и тестируемость.

Теоретическая часть

Паттерн Factory Method (Фабричный метод)

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Factory Method позволяет классу делегировать создание экземпляров подклассам.

Структура:

- Product – интерфейс создаваемых объектов.
- ConcreteProduct – реализация продукта.
- Creator – абстрактный создатель с фабричным методом factoryMethod().

- ConcreteCreator – переопределяет фабричный метод, возвращая конкретный продукт.

Применение:

- Класс заранее не знает, объекты каких подклассов ему нужно создавать.
- Класс делегирует создание объектов наследникам, чтобы локализовать выбор.
- Необходимость централизовать логику создания, но без жёсткой привязки к конкретным типам.

Пример кода (до рефакторинга):

```
// Прямое создание
Notification notification = new EmailNotification("text");
notification.send();
```

После (Factory Method):

```
abstract class Notifier {
   abstract Notification createNotification();
   void notify(String message) {
      Notification n = createNotification();
      n.send(message);
   }
}
class EmailNotifier extends Notifier {
   Notification createNotification() { return new EmailNotification(); }
}
```

Паттерн Abstract Factory (Абстрактная фабрика)

Назначение: предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.

Структура:

- AbstractFactory – объявляет методы для создания каждого типа продукта.

- ConcreteFactory1, ConcreteFactory2 – реализуют операции создания конкретных продуктов.
- AbstractProductA, AbstractProductB – интерфейсы семейства продуктов.
- ProductA1, ProductA2, ProductB1, ProductB2 – конкретные реализации.

Отличие от Factory Method: Abstract Factory создаёт семейство продуктов (несколько связанных типов), тогда как Factory Method – один продукт. Abstract Factory часто реализуется через несколько Factory Method.

Компромиссы и trade-offs

Аспект	**Прямое создание (new)**	**Factory Method**	**Abstract Factory**
Гибкость	Низкая (жёсткая привязка)	Средняя (выбор в подклассах)	Высокая (вся семья продуктов)
Количество классов	1	N+2	M*K + 2
Тестируемость	Сложно подменить продукт	Легко через подкласс-заглушку	Легко через фабрику-заглушку
Простота	Высокая	Средняя	Низкая (при избыточности)
Сложность внесения нового типа	Прямое изменение кода клиента	Добавление нового Creator	Добавление новой ConcreteFactory

Ключевой вопрос: когда использовать? Если создаётся только один вид продукта – Factory Method. Если система должна быть независима от способов создания целого семейства – Abstract Factory. Не適用 в случаях, когда семейств продуктов мало или они редко меняются.

Методика выполнения работы

Шаг 1. Изучение исходного кода

Получите индивидуальный вариант задания. Найдите места, где используются прямые вызовы конструкторов (new ProductX()). Определите:

- Какие типы продуктов существуют?
- Есть ли у них общий интерфейс?
- Как часто требуется добавлять новые типы?

Шаг 2. Выделение Factory Method

Для начала выберите один тип продукта (например, Payment). Выполните рефакторинг:

1. Создайте абстрактный класс или интерфейс Creator с фабричным методом.
2. Перенесите существующий клиентский код в метод, использующий фабричный метод.
3. Реализуйте конкретных создателей для каждого существующего продукта.
4. Удалите прямые вызовы new.

Критерий успеха: клиентский код больше не знает о конкретных классах продуктов.

Шаг 3. Расширение до Abstract Factory

Если в системе есть несколько связанных типов продуктов (например, Payment и Receipt), объедините их создание:

1. Определите интерфейс AbstractFactory с методами createPayment() и createReceipt().
2. Для каждой конфигурации (например, CreditCardFactory, PayPalFactory) реализуйте конкретную фабрику.
3. Модифицируйте клиент, чтобы он принимал фабрику через конструктор или сеттер (Dependency Injection).

Шаг 4. Динамическое переключение фабрик

Реализуйте механизм выбора фабрики на основе внешнего параметра (переменная окружения, конфигурационный файл, аргумент командной строки). Например:

```
AbstractFactory factory = Config.isTestMode()
    ? new MockFactory()
    : new ProductionFactory();
```

Шаг 5. Написание тестов

Создайте тесты, которые:

- Проверяют корректность создания продуктов каждой фабрикой.
- Используют **Test Double** (например, MockFactory, возвращающий заглушки) для изоляции тестируемого кода.
- Демонстрируют, что замена фабрики не требует изменения клиентской логики.

Шаг 6. Оценка компромиссов

В отчёте укажите:

- Сколько классов добавлено в результате рефакторинга.
- Насколько улучшилась тестируемость (приведите пример теста).
- Какие недостатки появились (например, усложнение навигации по коду).

Пример выполнения задания

Исходный код (до рефакторинга)

```
// Продукты
class CreditCardPayment {
    void pay(double amount) { System.out.println("Paid " + amount + " via Credit Card"); }
}
class PayPalPayment {
    void pay(double amount) { System.out.println("Paid " + amount + " via PayPal"); }
}
```

```
// Клиент (процессор заказов)
public class OrderProcessor {
    public void processOrder(double amount, String paymentType) {
        if (paymentType.equals("credit")) {
            CreditCardPayment payment = new CreditCardPayment();
            payment.pay(amount);
        } else if (paymentType.equals("paypal")) {
            PayPalPayment payment = new PayPalPayment();
            payment.pay(amount);
        } else {
            throw new IllegalArgumentException("Unknown payment type");
        }
    }
}
```

Проблемы: жёсткая зависимость от конкретных классов, нарушение OCP (открытости/закрытости), невозможно протестировать processOrder без реальных платежей.

Шаг 1. Введение единого интерфейса

```
interface Payment {
    void pay(double amount);
}
class CreditCardPayment implements Payment { ... }
class PayPalPayment implements Payment { ... }
```

Шаг 2. Factory Method

```
abstract class PaymentFactory {
    abstract Payment createPayment();
}

class CreditCardFactory extends PaymentFactory {
```

```
    Payment createPayment() { return new CreditCardPayment(); }
}
class PayPalFactory extends PaymentFactory {
    Payment createPayment() { return new PayPalPayment(); }
}

// Клиент
public class OrderProcessor {
    private PaymentFactory factory;
    public OrderProcessor(PaymentFactory factory) { this.factory = factory; }

    public void processOrder(double amount) {
        Payment payment = factory.createPayment();
        payment.pay(amount);
    }
}
```

Шаг 3. Abstract Factory (добавим связанный продукт – Receipt)

```
interface Receipt {
    void print();
}
class CreditCardReceipt implements Receipt { ... }
class PayPalReceipt implements Receipt { ... }

interface PaymentSystemFactory {
    Payment createPayment();
    Receipt createReceipt();
}

class CreditCardSystemFactory implements PaymentSystemFactory {
    public Payment createPayment() { return new CreditCardPayment(); }
```

```
    public Receipt createReceipt() { return new CreditCardReceipt(); }
}
class PayPalSystemFactory implements PaymentSystemFactory {
    public Payment createPayment() { return new PayPalPayment(); }
    public Receipt createReceipt() { return new PayPalReceipt(); }
}

// Новый клиент
public class OrderProcessor {
    private PaymentSystemFactory factory;
    public OrderProcessor(PaymentSystemFactory factory) { this.factory = factory; }

    public void processOrder(double amount) {
        Payment payment = factory.createPayment();
        Receipt receipt = factory.createReceipt();
        payment.pay(amount);
        receipt.print();
    }
}
```

Шаг 4. Тест (с использованием Mock-фабрики)

```
class MockPayment implements Payment {
    boolean paid = false;
    public void pay(double amount) { paid = true; }
}
class MockReceipt implements Receipt {
    boolean printed = false;
    public void print() { printed = true; }
}
class MockFactory implements PaymentSystemFactory {
```

```
    public Payment createPayment() { return new MockPayment(); }
    public Receipt createReceipt() { return new MockReceipt(); }
}

@Test
void testProcessOrderUsesPaymentAndReceipt() {
    MockFactory mockFactory = new MockFactory();
    OrderProcessor processor = new OrderProcessor(mockFactory);
    processor.processOrder(100.0);

    MockPayment payment = (MockPayment) mockFactory.createPayment();
    MockReceipt receipt = (MockReceipt) mockFactory.createReceipt();
    assertTrue(payment.paid);
    assertTrue(receipt.printed);
}
```

Компромиссы: добавлено 6 новых классов (вместо 2). Но зато теперь можно легко:

- добавить новый тип платежа (новая фабрика) без изменения OrderProcessor;
- тестировать процессор независимо от реальных платёжных шлюзов;
- переключить всю стратегию оплаты одной строкой.

Индивидуальные задания

Вариант 1. Система уведомлений: есть EmailMessage, SMSMessage, PushMessage. Каждый имеет метод send(). В коде логики (например, AlertService) повсеместно встречается new EmailMessage(), new SMSMessage() в зависимости от условий. Требуется реализовать Factory Method, затем Abstract Factory (добавить второй продукт – NotificationReport).

Вариант 2. Платёжный шлюз: продукты CreditCardTransaction, DebitCardTransaction, CryptoTransaction. У каждого методы authorize(),

capture(), refund(). Клиент – CheckoutService – содержит switch по типу. Выполнить рефакторинг до Abstract Factory (семейство: транзакция + подтверждающий чек Receipt).

Вариант 3. Графический редактор: фигуры Circle, Rectangle, Triangle. Рисовалка (Canvas) в методе draw(shapeType) создаёт фигуры через new. Добавить Factory Method, затем Abstract Factory для поддержки двух стилей отрисовки (векторный, растровый) – каждая фабрика создаёт фигуры определённого стиля.

Вариант 4. Документооборот: типы документов Invoice, Contract, Report. У каждого метод generate(), export(). Система (DocumentGenerator) содержит условные операторы. Реализовать Abstract Factory, добавив семейство Document + DocumentMetadata.

Вариант 5. Игровая логика: враги Zombie, Skeleton, Boss. У них метод attack(), takeDamage(). Клиент – BattleSystem – создаёт врагов в зависимости от уровня. Внедрить Factory Method. Расширить до Abstract Factory, добавив Weapon, соответствующего типу врага.

Вариант 6. Система кэширования: провайдеры RedisCache, MemcachedCache, InMemoryCache. У каждого get(key), set(key, value). Сервис (CacheManager) создаёт провайдера через new. Рефакторинг до Abstract Factory с продуктом Cache + CacheStatistics.

Вариант 7. Парсеры данных: форматы JSONParser, XMLParser, CSVParser. Метод parse(String data). Клиент (DataLoader) выбирает парсер по расширению файла. Заменить на Factory Method, затем Abstract Factory для семейства Parser + SchemaValidator.

Вариант 8. Логирование: FileLogger, DatabaseLogger, CloudLogger. Метод log(message). Система (Application) инициализирует логгер на основе конфигурации. Реализовать Abstract Factory, добавив LogFormatter (форматирование сообщения).

Вариант 9. UI-компоненты: кнопки WindowsButton, MacButton, LinuxButton. Отрисовка в Dialog. Заменить прямые вызовы на Factory Method.

Добавить Abstract Factory для семейства Button + ScrollBar (для каждой ОС своя пара).

Вариант 10. Обработка платежей (расширенный вариант): методы оплаты Card, Cash, GiftCard. Каждый имеет pay(), refund(). Магазин (Store) создаёт их через new в зависимости от ввода пользователя. Рефакторинг до Abstract Factory, добавив ReceiptGenerator.

Вариант 11. Отчёты: форматы PDFReport, HTMLReport, ExcelReport. Метод generate(data). Генератор (ReportEngine) содержит if-else. Внедрить Factory Method, затем Abstract Factory с продуктом Report + ReportStyle.

Вариант 12. Аутентификация: провайдеры LDAPAuth, DatabaseAuth, OAuth2Auth. Метод authenticate(credentials). Сервис (AuthService) создаёт провайдера по настройкам. Реализовать Abstract Factory, добавив UserProfileFetcher.

Вариант 13. Очереди сообщений: брокеры RabbitMQ, Kafka, ActiveMQ. Метод publish(message), consume(). Диспетчер (MessageBus) создаёт брокера через new. Рефакторинг до Abstract Factory с семейством MessageBroker + DeadLetterHandler.

Вариант 14. Шифрование: алгоритмы AES, Blowfish, ChaCha20. Метод encrypt(data), decrypt(data). Менеджер безопасности (CryptoService) содержит вызовы конструкторов. Заменить на Factory Method. Добавить Abstract Factory для связки Cipher + KeyGenerator.

Вариант 15. Базы данных (абстракция доступа): драйверы PostgreSQLDriver, MySQLDriver, MongoDriver. Методы connect(), query(). Фабрика соединений (DBConnectionPool) создаёт драйверы напрямую. Реализовать Abstract Factory с продуктами Connection + Transaction.

Требования к отчёту

Отчёт должен быть оформлен в электронном виде и содержать:

1. **Титульный лист** с указанием номера лабораторной работы, темы, ФИО студента, варианта.
2. **Цель и задачи работы** (копируются из начала документа).
3. **Теоретическая часть** – краткое описание паттернов Factory Method и Abstract Factory с диаграммой классов (можно в виде текстового ASCII-арта или вставленного изображения).
4. **Ход выполнения** (по шагам из методики) с листингами кода до и после рефакторинга для вашего варианта.
5. **Результаты тестирования** – скриншоты или код тестов, демонстрирующий работу разных фабрик и использование тестовых двойников.
6. **Оценка компромиссов** – таблица «было/стало» с количеством классов, строк кода, степени связности, плюсы и минусы.
7. **Вывод** – достигнута ли цель, какие навыки приобретены.
8. **Ответы на контрольные вопросы**.

Объём отчёта: 4–6 страниц.

Контрольные вопросы

1. В чём основное различие между Factory Method и Abstract Factory?

2. Приведите пример, когда лучше использовать Factory Method, а не Abstract Factory.

3. Как внедрение фабрик улучшает тестируемость кода? Опишите сценарий с Test Double.

4. Каковы недостатки паттерна Abstract Factory при добавлении нового типа продукта (например, третьего продукта в семейство)?

5. Какой принцип SOLID нарушает прямой вызов new Product() внутри класса?

6. Можно ли реализовать Abstract Factory без использования Factory Method? Если да, то как?

7. Какой паттерн можно использовать вместе с Abstract Factory для динамического выбора конкретной фабрики во время выполнения?

8. Оцените компромисс: добавление 6 новых классов против поддержки только двух типов продуктов. Когда это оправдано?

9. Как фабричные паттерны связаны с Dependency Injection? В чём разница?

10. Почему в Java часто используют Supplier<T> или Factory<T> (функциональный интерфейс) вместо классического Factory Method? Какие плюсы и минусы?

Лабораторная работа №3. Builder и Fluent API для сложных объектов

Цель работы: очистить и агрегировать код создания сложных объектов с множеством параметров путём внедрения паттерна Builder с Fluent API, обеспечив читаемость, расширяемость и защиту от ошибок при конфигурировании.

Задачи:

1. Проанализировать существующий код с «телескопическими» конструкторами или раздутыми наборами сеттеров для сложного DTO/конфигурационного объекта.
2. Реализовать паттерн Builder с каскадными методами (Fluent API) для пошагового конструирования объекта.
3. Обеспечить обязательные параметры, проверку пост-условий и неизменяемость (immutability) готового объекта.
4. Сравнить три подхода: телескопический конструктор, JavaBeans (сеттеры) и Builder с точки зрения читаемости, безопасности и производительности.
5. Написать юнит-тесты, проверяющие корректность сборки и защиту от неполной/некорректной конфигурации.
6. Провести качественную оценку компромиссов (количество кода vs удобство использования vs производительность).

Теоретическая часть

Проблема создания сложных объектов

При проектировании классов с большим количеством параметров (более 4–5) возникают следующие проблемы:

- **Телескопический конструктор** – множество перегруженных конструкторов с растущим числом параметров. Клиентский код становится нечитаемым, легко перепутать порядок параметров,

особенно однотипных (например, int minPrice, int maxPrice, int minRating, int maxRating).

- **JavaBeans с сеттерами** – объект создаётся через пустой конструктор, затем вызываются сеттеры. При этом объект находится в «неконсистентном» состоянии между вызовами, невозможно сделать объект неизменяемым (immutable), усложняется проверка целостности.
- **Параметры по умолчанию** в языках без поддержки именованных параметров (Java до недавнего времени) – отсутствуют.

Паттерн Builder

Назначение: отделяет конструирование сложного объекта от его представления, позволяя одному и тому же процессу конструирования создавать различные представления.

Структура:

- Product – сложный объект, обычно неизменяемый (поля final).
- Builder – интерфейс или статический вложенный класс с методами для установки параметров (каждый метод возвращает this).
- Метод build() – выполняет проверку и возвращает готовый продукт.

Fluent API – стиль интерфейса, при котором методы возвращают сам объект (или контекст) для цепочки вызовов:

```
Product p = new Product.Builder()
    .setParam1(value1)
    .setParam2(value2)
    .setParam3(value3)
    .build();
```

Вариации паттерна Builder

1. **Классический GoF Builder** с директором (Director), управляющим процессом. Используется, когда процесс конструирования фиксирован, но вариативны детали.

2. **Статический вложенный Builder** (Эффективная Java, Джошуа Блох) – самый распространённый в современных Java-проектах.
3. **Гибрид с проверками** – добавление validate() в build(), выбрасывание IllegalStateException при отсутствии обязательных параметров.
4. **Build с опциональными значениями** – использование Optional, типов-маркеров (staged builder) для принудительного порядка вызовов.

Сравнение подходов

Характеристика	**Телескопический конструктор**	**JavaBeans (сеттеры)**	**Builder + Fluent API**
Читаемость кода вызова	Низкая (много параметров)	Средняя (каждый сеттер отдельно)	Высокая (цепочка с именами)
Безопасность (порядок)	Высокая (компиляция)	Низкая (объект не готов промежуточно)	Средняя (build проверяет)
Неизменяемость продукта	Да	Нет	Да
Проверка обязательных параметров	В конструкторе	Распределённая по сеттерам	Централизованная в build()
Расширяемость (новые параметры)	Добавляются новые конструкторы (взрыв)	Новые сеттеры (просто)	Новые методы в Builder
Количество кода	Минимум (для 2–3 параметров)	Среднее	Больше (вложенный класс)
Производительность	Лучшая	Хорошая	Небольшой оверхед (создание Builder)

Trade-offs: Builder оправдан для объектов с 5+ параметрами, несколькими обязательными и группами опциональных. Для простых объектов (2–3 параметра) телескопический конструктор предпочтительнее.

Использование Builder для всех DTO подряд – преждевременная оптимизация наоборот, усложнение кода.

Методика выполнения работы

Шаг 1. Изучение исходного кода

Получите индивидуальный вариант, содержащий класс с «проблемным» конструктором или набором сеттеров. Определите:

- Сколько параметров у объекта (обязательных, опциональных)?
- Есть ли параметры со значениями по умолчанию?
- Используется ли объект в неизменяемом виде после создания?
- Какие ошибки чаще всего возникают при создании (перепутанные параметры, пропущенные поля)?

Шаг 2. Проектирование Builder

Создайте статический вложенный класс Builder внутри класса продукта:

1. Поля Builder’а дублируют поля продукта (или хранят отдельные значения).
2. Для каждого параметра определите метод, возвращающий Builder. Методы именуются как сам параметр (Fluent): builder.setType(type) или напрямую type(type).
3. Метод build() создаёт экземпляр продукта, вызывая приватный конструктор, передавая ссылку на Builder или отдельные значения.
4. В build() реализуйте проверку обязательных полей (например, if (model == null) throw IllegalStateException(...)).

Шаг 3. Реализация с поддержкой значений по умолчанию

Добавьте умолчания через инициализацию полей Builder’а в конструкторе Builder’а:

private int pageSize = 10; *// значение по умолчанию*

Пользователь может переопределить через pageSize(int).

Шаг 4. Сравнение с альтернативами

Напишите небольшой тестовый фрагмент, демонстрирующий создание того же объекта тремя способами:

- Телескопический конструктор (если бы он был).
- Сеттеры (если бы были сеттеры).
- Ваш Builder.

В отчёте приведите визуальное сравнение читаемости.

Шаг 5. Тестирование

Напишите тесты (JUnit) на:

- Успешное создание объекта с минимальным набором обязательных параметров.
- Успешное создание с опциональными параметрами.
- Выброс исключения при отсутствии обязательного параметра.
- Проверку неизменяемости (попытка изменить поля через рефлексию или отсутствие сеттеров).

Шаг 6. Анализ компромиссов

Оцените количественно: сколько строк кода занял Builder (включая вложенный класс). Сравните с объёмом конструкторов. Оцените, как изменилась сложность поддержки при добавлении нового параметра.

Пример выполнения задания

Исходный код (проблемный)

```
public class SearchFilter {
    private final String query;          // обязательный
    private final int limit;             // опциональный, по умолч. 20
    private final int offset;            // опциональный, 0
    private final String sortBy;         // опциональный, "relevance"
    private final boolean asc;           // опциональный, true
    private final List<String> categories; // опциональный, пустой список
    private final Double minPrice;       // опциональный, null
    private final Double maxPrice;       // опциональный, null
```

```
        // Телескопический конструктор (ужас)
        public SearchFilter(String query) { this(query, 20); }
        public SearchFilter(String query, int limit) { this(query, limit, 0); }
        public SearchFilter(String query, int limit, int offset) { this(query, limit, offset, "relevance"); }
        public SearchFilter(String query, int limit, int offset, String sortBy) {
            this(query, limit, offset, sortBy, true);
        }
        public SearchFilter(String query, int limit, int offset, String sortBy, boolean asc) {
            this(query, limit, offset, sortBy, asc, Collections.emptyList());
        }
        public SearchFilter(String query, int limit, int offset, String sortBy, boolean asc, List<String> categories) {
            this(query, limit, offset, sortBy, asc, categories, null, null);
        }
        public SearchFilter(String query, int limit, int offset, String sortBy, boolean asc, List<String> categories, Double minPrice, Double maxPrice) {
            // валидация
            if (query == null || query.isBlank()) throw new IllegalArgumentException("query required");
            this.query = query;
            this.limit = limit;
            this.offset = offset;
            this.sortBy = sortBy != null ? sortBy : "relevance";
            this.asc = asc;
            this.categories = categories != null ? categories : Collections.emptyList();
            this.minPrice = minPrice;
```

```
        this.maxPrice = maxPrice;
    }
}
```

Клиентский код:

java

```
// Какой параметр что означает? Трудно читать.
SearchFilter filter = new SearchFilter("laptop", 50, 10, "price", false, Arrays.asList("electronics"), 100.0, 2000.0);
```

Реализация Builder (Fluent API)

```
public class SearchFilter {
    private final String query;
    private final int limit;
    private final int offset;
    private final String sortBy;
    private final boolean asc;
    private final List<String> categories;
    private final Double minPrice;
    private final Double maxPrice;

    private SearchFilter(Builder builder) {
        // валидация обязательных
        if (builder.query == null || builder.query.isBlank())
            throw new IllegalStateException("query is required");
        this.query = builder.query;
        this.limit = builder.limit;
        this.offset = builder.offset;
        this.sortBy = builder.sortBy;
        this.asc = builder.asc;
```

```
        this.categories = Collections.unmodifiableList(new
ArrayList<>(builder.categories));
        this.minPrice = builder.minPrice;
        this.maxPrice = builder.maxPrice;
        // дополнительная проверка minPrice <= maxPrice
        if (minPrice != null && maxPrice != null && minPrice > maxPrice)
            throw new IllegalStateException("minPrice must be <= maxPrice");
    }

    public static Builder builder() { return new Builder(); }

    public static class Builder {
        private String query;
        private int limit = 20;
        private int offset = 0;
        private String sortBy = "relevance";
        private boolean asc = true;
        private List<String> categories = new ArrayList<>();
        private Double minPrice;
        private Double maxPrice;

        public Builder query(String query) { this.query = query; return this; }
        public Builder limit(int limit) { this.limit = limit; return this; }
        public Builder offset(int offset) { this.offset = offset; return this; }
        public Builder sortBy(String sortBy) { this.sortBy = sortBy; return this;
}
        public Builder asc(boolean asc) { this.asc = asc; return this; }
        public Builder addCategory(String category) {
            this.categories.add(category);
            return this;
```

```
            }
            public Builder categories(List<String> categories) {
                this.categories = new ArrayList<>(categories);
                return this;
            }
            public Builder minPrice(Double minPrice) { this.minPrice = minPrice;
return this; }
            public Builder maxPrice(Double maxPrice) { this.maxPrice = maxPrice;
return this; }

            public SearchFilter build() {
                return new SearchFilter(this);
            }
        }

        // геттеры (без сеттеров – immutable)
    }
```

Клиентский код после рефакторинга:

```
    SearchFilter filter = SearchFilter.builder()
        .query("laptop")
        .limit(50)
        .offset(10)
        .sortBy("price")
        .asc(false)
        .addCategory("electronics")
        .minPrice(100.0)
        .maxPrice(2000.0)
        .build();
    // Читаемость идеальная, параметры явно подписаны.
```

Сравнительная таблица для примера

Критерий	Телескопический конструктор	Builder
Строк кода продукта	35	55
Строк кода клиента (создание)	1 (нечитаемая)	8 (читаемая)
Вероятность ошибки при создании	Высокая	Низкая
Добавление нового параметра	Новый конструктор/перегрузка	Новый метод в Builder
Неизменяемость	Да	Да
Проверка min <= max	В конструкторе	В build()

Компромиссы: объём кода продукта увеличился на ~60% (20 строк). Но систематизированное создание сложных объектов экономит время отладки и сопровождения. Для high-load систем могут быть заметны накладные расходы на создание дополнительного объекта Builder, но в подавляющем большинстве приложений это незначительно.

Индивидуальные задания

Вариант 1. Класс ReportConfig с параметрами: title (обяз.), format (обяз. enum: PDF, HTML), includeHeader (default true), includeFooter (default true), pageSize (default A4), marginLeft, marginRight, marginTop, marginBottom (default 20), watermarkText (optional). Реализовать Builder с Fluent API.

Вариант 2. Класс EmailMessage со сложной конфигурацией: to (обяз., список), from (обяз.), subject (обяз.), body (обяз.), cc (опционально), bcc (опционально), attachments (список путей), priority (enum: HIGH, NORMAL, LOW, default NORMAL), replyTo (опционально). Builder с проверкой валидности email.

Вариант 3. Класс DatabaseConnectionConfig: host (обяз.), port (обяз.), database (обяз.), username (обяз.), password (обяз.), useSSL (default false), timeout (default 30), poolSize (default 10), connectionProperties (Map, optional). Builder с fluent-стилем. Обязательно наличие метода build() с проверкой порта (1-65535).

Вариант 4. Класс HttpRequest (для эмуляции запроса): url (обяз.), method (enum GET,POST,PUT,DELETE, default GET), headers (Map), body (опционально), timeout (default 5), retryCount (default 0). Builder должен поддерживать добавление заголовков по одному.

Вариант 5. Класс ScheduleTask (крон-подобная задача): name (обяз.), cronExpression (обяз.), enabled (default true), maxRetries (default 3), retryDelay (default 1000), timeout (optional), onFailureAction (enum), tags (список строк). Builder с проверкой формата cron.

Вариант 6. Класс PizzaOrder (система заказа пиццы): size (обяз., enum SMALL/MEDIUM/LARGE), dough (обяз., thin/thick), sauce (обяз.), cheese (optional, boolean default true), toppings (список), deliveryAddress (обяз.), contactPhone (обяз.), specialInstructions (опционально). Builder с проверкой номера телефона.

Вариант 7. Класс FileProcessorConfig: inputPath (обяз.), outputPath (обяз.), encoding (default UTF-8), bufferSize (default 8192), compression (enum NONE,GZIP,ZIP, default NONE), deleteSource (default false), maxFileSizeMB (optional). Builder с проверкой существования входного пути.

Вариант 8. Класс DashboardWidgetConfig: widgetType (обяз., enum CHART,TABLE,GAUGE), title (обяз.), refreshIntervalSec (default 30), dataSource (обяз., строка), width (default 2), height (default 2), colorTheme (default "light"), filters (Map). Builder должен иметь метод applyDefaultTheme().

Вариант 9. Класс CacheConfig: maxSize (обяз.), ttlSeconds (обяз.), evictionPolicy (enum LRU, LFU, FIFO, default LRU), enableMetrics (default false), serializer (enum JAVA, JSON, default JAVA), diskPersistent (default false). Builder с проверкой maxSize > 0.

Вариант 10. Класс SearchRequest (эластик-подобный): query (обяз.), fields (список полей для поиска), highlight (default false), fuzzy (default false), minScore (optional), aggregations (Map), size (default 10), from (default 0), sort (поле и направление). Builder с fluent-синтаксисом.

Вариант 11. Класс NotificationTemplate: name (обяз.), subject (обяз.), bodyText (обяз.), locale (default en_US), attachments (list), variables (Map, variables for substitution), priority (default MEDIUM). Builder должен клонировать шаблоны (метод cloneBuilder()).

Вариант 12. Класс GameSettings: difficulty (enum EASY,NORMAL,HARD, default NORMAL), soundVolume (0-100, default 80), musicVolume (0-100, default 50), fullscreen (default false), resolutionWidth (default 1920), resolutionHeight (default 1080), controls (Map<Action,KeyCode>), language (default "en"). Builder с валидацией диапазонов громкости.

Вариант 13. Класс DataImportJob: tableName (обяз.), sourceFile (обяз.), delimiter (default ','), skipHeader (default true), batchSize (default 1000), errorHandling (enum SKIP, STOP, LOG, default SKIP), targetSchema (optional). Builder с проверкой существования файла.

Вариант 14. Класс UserPreferences: theme (default "system"), notificationsEnabled (default true), notifyEmail (optional), notifySMS (optional), timezone (default UTC), dateFormat (default "yyyy-MM-dd"), itemsPerPage (default 20), language (default "en"). Builder с проверкой: если notificationsEnabled true, то хотя бы один способ уведомления должен быть задан.

Вариант 15. Класс PaymentGatewayConfig: merchantId (обяз.), secretKey (обяз.), apiUrl (обяз.), timeoutMs (default 30000), retryAttempts (default 3), webhookUrl (опционально), sandboxMode (default false), supportedCurrencies (список, default USD,EUR). Builder с проверкой, что secretKey не пустой и не менее 16 символов.

Требования к отчёту

Отчёт должен содержать следующие разделы:

1. **Титульный лист** с указанием ФИО, группы, номера варианта.
2. **Цель и задачи работы** (воспроизводятся из шапки).
3. **Теоретическая часть** – описание проблемы «телескопических конструкторов» и паттерна Builder, примеры из литературы (Bloch), сравнение подходов в виде таблицы.
4. **Ход выполнения**:
    - Листинг исходного класса с проблемой (телескопический конструктор или сеттеры).
    - Листинг рефакторированного класса с Builder.
    - Пример создания объекта в клиентском коде до и после.
5. **Результаты тестирования** – код тестов (JUnit) и скриншот выполнения.
6. **Сравнительный анализ** – таблица «было/стало» по метрикам: количество конструкторов, строк кода, читаемость, безопасность.
7. **Оценка компромиссов** – описание trade-offs, применительно к вашему конкретному объекту (например: «добавлено 30 строк кода, но теперь невозможно создать объект с неверным типом сортировки»).
8. **Вывод** – достигнута ли цель, какие получены навыки.
9. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Контрольные вопросы

1. В чём основные недостатки телескопического конструктора? Приведите пример, когда он всё же предпочтительнее Builder’а.

2. Какая проблема паттерна JavaBeans (setter-методов) решается Builder’ом?

3. Почему Builder позволяет сделать продукт неизменяемым (immutable), а подход с сеттерами – нет?

4. Что такое Fluent API? Как он связан с паттерном Builder?

5. Как в вашей реализации Builder’а обеспечено требование обязательных параметров? Как бы вы сделали проверку на этапе компиляции (staged builder)?

6. Как добавление нового параметра в класс повлияет на существующие Builder’ы? Насколько это дорого?

7. Каковы накладные расходы (memory, CPU) при использовании Builder? В каких системах это может быть критично?

8. Как можно использовать Builder для создания объектов с иерархией наследования? (например, для подтипов одного базового класса)

9. В каких случаях следует избегать Builder’а и оставить простой конструктор?

10. Как паттерн Builder связан с принципом единственной ответственности (SRP)? Кто отвечает за создание, а кто – за бизнес-логику?

Лабораторная работа №4. Decorator и Proxy: динамическое расширение без наследования

Цель работы: применить паттерны Decorator и Proxy для динамического добавления сквозной функциональности (логирование, кэширование, контроль доступа) к существующему сервису без изменения его кода и наследования, а также оценить влияние на производительность и тестируемость.

Задачи:

1. Проанализировать существующий сервис получения данных с жёстко зашитой логикой (без возможности расширения).
2. Реализовать паттерн **Decorator** для добавления кэширования (in-memory cache) к сервису, не затрагивая исходный класс.
3. Реализовать паттерн **Proxy** для добавления проверки прав доступа (авторизации) к сервису.
4. Обеспечить возможность комбинирования декораторов и прокси (например, прокси + декоратор кэша).
5. Написать модульные тесты для проверки логики кэша и контроля доступа.
6. Провести простейшие тесты производительности (замеры времени выполнения с кэшем и без).
7. Сравнить подходы Decorator и Proxy по назначению, структуре и последствиям.

Теоретическая часть

Проблема горизонтального расширения

В объектно-ориентированных системах часто требуется добавить сквозную функциональность (cross-cutting concerns) – логирование, кэширование, контроль доступа, мониторинг – к существующим классам. Наследование приводит к взрывному росту числа классов (комбинации

признаков). Паттерны Decorator и Proxy предлагают композицию вместо наследования.

Паттерн Decorator (Декоратор)

Назначение: динамически добавляет объекту новые обязанности, являясь гибкой альтернативой подклассам.

Структура:

- Component – интерфейс, определяющий операции.
- ConcreteComponent – исходный класс, реализующий Component.
- Decorator – абстрактный класс, содержащий ссылку на Component и реализующий его интерфейс. Делегирует операции компоненту.
- ConcreteDecorator – добавляет поведение до и/или после вызова компонента.

Особенности:

- Декораторы могут быть вложены друг в друга (множественное расширение).
- Порядок декораторов важен (например, кэш поверх логирования или наоборот).

Паттерн Proxy (Заместитель)

Назначение: предоставляет суррогатный объект, контролирующий доступ к другому объекту.

Варианты прокси:

- **Виртуальный прокси** – откладывает создание тяжёлого объекта.
- **Защитный прокси** (Access Proxy) – проверяет права доступа.
- **Кэширующий прокси** – аналогичен декоратору кэша (но прокси обычно управляет жизненным циклом).
- **Логирующий прокси** – записывает вызовы.

Ключевое отличие от Decorator: прокси контролирует доступ и часто сам создаёт или управляет реальным объектом; декоратор фокусируется на добавлении поведения, не скрывая сам объект.

Сравнение Decorator и Proxy

Характеристика	Decorator	Proxy
Намерение	Добавить новые обязанности	Контролировать доступ / управлять жизненным циклом
Отношение к компоненту	Декоратор получает компонент извне (через конструктор)	Прокси может сам создавать или искать реальный объект
Прозрачность для клиента	Полная (клиент работает с Component)	Полная (клиент также работает с интерфейсом)
Количество экземпляров	Несколько декораторов могут оборачивать один компонент	Обычно один прокси на один реальный объект
Типичные применения	Логирование, кэширование, сжатие, шифрование	Ленивая загрузка, авторизация, удалённый доступ
Композиция с другими	Декораторы можно комбинировать	Прокси обычно один на объект

Важно: в конкретной реализации грань размывается. Например, кэширующий декоратор и кэширующий прокси структурно похожи. Различие в намерении и способе получения целевого объекта.

Принцип открытости/закрытости (OCP)

Оба паттерна позволяют расширять поведение без модификации существующих классов. Это ключевое преимущество при работе с legacy-кодом или сторонними библиотеками.

Компромиссы и trade-offs

- **Плюсы:** гибкость, соблюдение OCP, возможность динамического включения/выключения функциональности (собрать цепочку декораторов по условию).
- **Минусы:** увеличение количества классов (каждый декоратор – отдельный класс), усложнение отладки (стеки вызовов становятся глубже), незначительное снижение производительности (дополнительные вызовы).

Критический вопрос: что выбрать, если нужно добавить и кэш, и логирование, и авторизацию? Обычно используют комбинацию: прокси для авторизации (контроль доступа на уровне входа) и декораторы для кэша и логирования внутри. Или наоборот – всё можно сделать декораторами.

Методика выполнения работы

Шаг 1. Изучение исходного сервиса

Получите индивидуальный вариант – интерфейс DataService (или аналогичный) и класс BasicDataService с медленной операцией (например, имитация задержки или вызова внешнего API). Определите методы, которые нужно расширить (обычно getData(id) или find(query)).

Пример исходного интерфейса:

```
interface DataService {
   String getData(String id);
}
```

Шаг 2. Реализация декоратора кэширования

1. Создайте абстрактный класс DataServiceDecorator, реализующий DataService и содержащий ссылку на DataService.
2. Реализуйте CachingDecorator, который перед вызовом проверяет наличие данных в Map<String, String>, при попадании возвращает из кэша, иначе вызывает целевой сервис, сохраняет результат и возвращает.
3. Учтите, что кэш должен быть потокобезопасным (для простоты можно ConcurrentHashMap).

Шаг 3. Реализация Proxy для контроля доступа

1. Реализуйте AccessProxy, также реализующий DataService.
2. В конструктор передайте реальный сервис и, например, объект User или токен.
3. Перед вызовом проверяйте права (например, есть ли у пользователя роль ADMIN или соответствует ли ID пользователя запрашиваемому).
4. При отсутствии прав выбрасывайте исключение SecurityException или возвращайте сообщение об ошибке.

Шаг 4. Комбинирование

Продемонстрируйте создание цепочек:

DataService service = new BasicDataService();

DataService cached = new CachingDecorator(service);

DataService secure = new AccessProxy(cached, user);

Или другой порядок: сначала прокси, потом кэш. Объясните разницу в поведении.

Шаг 5. Тестирование

Напишите JUnit-тесты:

- Проверка работы кэша: два одинаковых вызова – второй должен быть быстрее (или не вызывать реальный сервис).
- Проверка прокси: вызов с правильными правами – успех, с неправильными – исключение.
- Тест комбинации: кэш работает поверх прокси или наоборот.

Шаг 6. Тесты производительности

Напишите простой бенчмарк (замер времени через System.nanoTime()):

- 100 вызовов без кэша.
- 100 вызовов с кэшем (первые 50 уникальных, последующие 50 повторяющихся).
- Сравните среднее время.

Шаг 7. Оценка компромиссов

В отчёте ответьте на вопросы:

- Сколько новых классов создано?
- Как изменилась читаемость архитектуры?
- Как тестировать каждый декоратор изолированно?
- Что произойдёт, если порядок декораторов поменять?

Пример выполнения задания

Исходный код

```
// Интерфейс
public interface WeatherService {
   String getWeather(String city);
}

// Реализация (симулирует медленный внешний вызов)
public class OpenWeatherService implements WeatherService {
   @Override
   public String getWeather(String city) {
      // Имитация задержки
      try { Thread.sleep(500); } catch (InterruptedException e) {}
      return "Weather in " + city + ": sunny, +25°C";
   }
}
```

Декоратор кэширования

```
public abstract class WeatherServiceDecorator implements WeatherService {
   protected WeatherService wrapped;
   public WeatherServiceDecorator(WeatherService ws) { this.wrapped = ws;
}
   }

public class CachingWeatherDecorator extends WeatherServiceDecorator {
```

```
    private final Map<String, String> cache = new ConcurrentHashMap<>();

    public CachingWeatherDecorator(WeatherService ws) { super(ws); }

    @Override
    public String getWeather(String city) {
        return cache.computeIfAbsent(city, wrapped::getWeather);
    }
}
```

Proxy контроля доступа (предположим, что только администратор может смотреть погоду)

```
public class AccessProxy implements WeatherService {
    private final WeatherService target;
    private final User currentUser;

    public AccessProxy(WeatherService target, User user) {
        this.target = target;
        this.currentUser = user;
    }

    @Override
    public String getWeather(String city) {
        if (!currentUser.hasRole("ADMIN")) {
            throw new SecurityException("Access denied: ADMIN role required");
        }
        return target.getWeather(city);
    }
}
```

Пример использования

```
WeatherService basic = new OpenWeatherService();
// Добавляем кэш
WeatherService cached = new CachingWeatherDecorator(basic);
// Добавляем контроль доступа
User user = new User("john", Set.of("USER"));
WeatherService secure = new AccessProxy(cached, user);

try {
   String weather = secure.getWeather("Moscow"); // SecurityException
} catch (SecurityException e) {
   System.out.println(e.getMessage());
}

User admin = new User("admin", Set.of("ADMIN"));
WeatherService adminProxy = new AccessProxy(cached, admin);
String weather = adminProxy.getWeather("Moscow"); // работает + кэширует
String weather2 = adminProxy.getWeather("Moscow"); // из кэша, быстро
```

Тест производительности

```
@Test
void testCachingPerformance() {
   WeatherService service = new OpenWeatherService();
   WeatherService cached = new CachingWeatherDecorator(service);

   long start = System.nanoTime();
   for (int i = 0; i < 10; i++) service.getWeather("City");
   long withoutCache = System.nanoTime() - start;
```

```
        start = System.nanoTime();
        for (int i = 0; i < 10; i++) cached.getWeather("City");
        long withCache = System.nanoTime() - start;

        assertTrue(withCache < withoutCache / 2); // с кэшем значительно быстрее
    }
```

Компромиссы в примере

- Добавлено 3 новых класса (Decorator, CachingDecorator, AccessProxy) – повышена сложность навигации.
- Кэширование через декоратор: легко отключается (не использовать декоратор) или заменяется на другую стратегию.
- Прокси для доступа: локализует проверки авторизации в одном месте.
- При изменении порядка: если сначала прокси, потом кэш – проверка прав выполняется до кэша, что правильно (не кэшируем запросы неавторизованных). Если наоборот – то кэш может сохранить данные, которые недоступны другому пользователю (потенциальная утечка). Порядок имеет значение.

Индивидуальные задания

Вариант 1. UserService с методом getUserProfile(String userId). Реализовать декоратор кэширования (TTL 60 секунд) и прокси логирования (логировать каждый вызов с аргументами). Комбинировать.

Вариант 2. ProductService с методом getPrice(String productId). Исходный сервис делает запрос к БД (эмулировать Thread.sleep). Добавить декоратор кэширования и защитный прокси, разрешающий доступ только определённым ролям.

Вариант 3. DocumentService с методом getDocumentContent(String docId). Реализовать декоратор сжатия (возвращает сжатую строку – base64 gzip) и прокси авторизации по владельцу документа.

Вариант 4. FinancialService с методом getBalance(String accountId). Декоратор кэширования (слабая ссылка) + прокси аудита (записывает время и пользователя в лог). Протестировать производительность.

Вариант 5. SearchService с методом search(String query, int maxResults). Декоратор кэширования по query (без учёта maxResults) – ошибка? Исправить. Прокси для ограничения частоты запросов (rate limiting) – реализовать как второй декоратор (демонстрация вложенности).

Вариант 6. ImageService с методом loadImage(String url). Декоратор кэширования изображений в памяти (хранить byte[]). Прокси для валидации URL (только HTTPS). Написать тесты.

Вариант 7. TranslationService с методом translate(String text, String fromLang, String toLang). Декоратор кэширования (ключ – текст+языки). Прокси для проверки лимитов (лимит символов в день). Комбинировать.

Вариант 8. CurrencyConverter с методом convert(double amount, String from, String to). Декоратор кэширования курса (обновлять раз в час) и виртуальный прокси для ленивой загрузки внешнего API.

Вариант 9. EmailService с методом sendEmail(Email email). Декоратор для добавления заголовка (X-Processed-By) и прокси для проверки отправителя в белом списке.

Вариант 10. StockQuoteService с методом getQuote(String symbol). Декоратор кэширования с протуханием (30 секунд) и защитный прокси, требующий API-ключа. Продемонстрировать подмену ключа в тесте.

Вариант 11. ConfigService с методом getConfig(String key). Декоратор кэширования (in-memory) и прокси для перехвата и маскировки паролей (если key содержит "password", заменяет значение на "***").

Вариант 12. AnalyticsService с методом recordEvent(Event e). Декоратор для асинхронной обработки (запуск в отдельном потоке) и прокси для проверки, что событие не null.

Вариант 13. ReportService с методом generateReport(ReportRequest req). Декоратор кэширования отчётов (по параметрам). Прокси для проверки прав: только пользователи с ролью ANALYST.

Вариант 14. NotificationService с методом notify(User user, String message). Декоратор для ограничения частоты (не чаще 1 уведомления в минуту на пользователя) и прокси логирования в БД.

Вариант 15. GeoLocationService с методом getCoordinates(String address). Декоратор кэширования (LRU размером 100) и прокси для слепоты (для тестового режима возвращает координаты 0,0). Сравнить производительность.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи** (из начала работы).
3. **Теоретическая часть** – описание паттернов Decorator и Proxy, их сравнение, диаграмма классов UML (можно текстом или рисунком).
4. **Ход выполнения**:
   - Исходный интерфейс и класс сервиса.
   - Реализация декоратора кэширования.
   - Реализация прокси контроля доступа (или иного).
   - Комбинирование (пример кода).
5. **Результаты тестирования**:
   - Модульные тесты (листинг + скриншоты выполнения).
   - Тесты производительности (таблица с временами).
6. **Оценка компромиссов** – обсуждение trade-offs в вашем варианте.

7. **Вывод** – достигнуты ли цели, какие навыки сформированы.
8. **Ответы на контрольные вопросы** (письменно).

Контрольные вопросы

1. В чём ключевое различие между паттернами Decorator и Proxy по их намерению?

2. Может ли один объект быть одновременно и декоратором, и прокси? Приведите пример.

3. Как порядок наложения декораторов влияет на результат? Опишите сценарий с кэшем и логированием.

4. Почему при использовании декораторов важно, чтобы компонент имел общий интерфейс?

5. В каком случае лучше использовать прокси вместо декоратора для кэширования?

6. Как реализовать прозрачную замену декораторов во время выполнения? (динамическая композиция)

7. Какие проблемы может вызвать кэширование через декоратор в многопоточной среде? Как их решили в работе?

8. Как тестировать отдельный декоратор, не вызывая реальный компонент? Какие паттерны тестирования при этом используются?

9. Сравните накладные расходы на вызов через 2 декоратора (декоратор -> декоратор -> компонент) с прямым вызовом. Это критично для высоконагруженной системы?

10. Как принцип подстановки Барбары Лисков (LSP) связан с использованием Decorator и Proxy?

Лабораторная работа №5. Composite для иерархических структур

Цель работы: очистить и агрегировать код работы с иерархическими структурами путём внедрения паттерна Composite, обеспечивающего единообразный интерфейс для отдельных объектов и их групп, а также упрощающего обход, агрегацию и отображение дерева.

Задачи:

1. Проанализировать существующий код, который по-разному обрабатывает простые и составные объекты (например, товары и категории).
2. Реализовать паттерн Composite: общий интерфейс (Component), листовые узлы (Leaf) и составные узлы (Composite).
3. Реализовать методы агрегации (подсчёт общей цены, количества элементов) для всего дерева.
4. Реализовать обход дерева (например, отрисовку в виде текстовой иерархии или поиск элементов).
5. Обеспечить возможность добавления/удаления дочерних узлов в составных объектах.
6. Написать юнит-тесты, проверяющие корректность работы с деревьями различной вложенности.
7. Оценить компромиссы: единообразие vs безопасность типов, прозрачность vs надёжность операций.

Теоретическая часть

Проблема работы с иерархиями

В программных системах часто встречаются древовидные структуры: меню (пункты и подменю), каталоги товаров (категории и товары), организационная структура (сотрудники и отделы), UI-компоненты (контейнеры и виджеты). Типичная проблема: клиентский код вынужден

различать простые и составные объекты, используя разные интерфейсы и реализуя рекурсивные алгоритмы отдельно. Это приводит к дублированию кода и сложности расширения.

Паттерн Composite (Компоновщик)

Назначение: компонует объекты в древовидные структуры для представления иерархий «часть-целое». Позволяет клиентам единообразно обращаться с отдельными объектами и их композициями.

Структура:

- Component – интерфейс, определяющий общие операции для листьев и контейнеров.
- Leaf – листовой узел, не имеет дочерних элементов. Реализует операции компонента.
- Composite – контейнер, содержит коллекцию дочерних компонентов. Реализует операции компонента, часто делегируя их детям. Также предоставляет методы управления дочерними элементами (добавление, удаление).

Типичные методы Component:

- operation() – основная операция, выполняемая над узлом.
- add(Component) / remove(Component) – методы управления детьми (можно определить на уровне Component с пустой реализацией по умолчанию или только в Composite, в зависимости от прозрачности).

Прозрачность vs безопасность

- **Прозрачный подход (transparent)**: методы add, remove объявлены в интерфейсе Component. Клиент может единообразно добавлять или удалять детей, но листовые узлы должны выбрасывать исключения, если эти методы вызваны. Клиент проще, но менее безопасен.
- **Безопасный подход (safe)**: методы управления детьми находятся только в интерфейсе Composite. Клиент должен проверять тип узла перед операцией. Безопаснее, но требует явного приведения типов.

В данной работе рекомендуется прозрачный подход с выбрасыванием UnsupportedOperationException **в листьях** (как в Java Collections Framework), либо оставить методы только в Composite, но для упрощения демонстрации можно сделать общий интерфейс с дефолтными методами, выдающими исключение.

Типичные применения Composite

- Графические редакторы: группировка фигур.
- Файловые системы: файлы и папки.
- Каталоги (категории и продукты).
- Организационные диаграммы.
- UI-деревья.

Компромиссы

Аспект	Без Composite	С Composite
Сложность клиента	Высокая (приходится писать рекурсию для каждого типа)	Низкая (единый вызов)
Добавление нового типа узла	Нужно менять все алгоритмы обработки	Достаточно реализовать Component
Простота реализации	Простая для простых структур	Выше начальная сложность
Гибкость	Низкая	Высокая (можно строить деревья любой глубины)
Безопасность типов	Высокая (разные типы)	Ниже (все узлы одного типа)

Trade-off: унификация упрощает клиентский код, но стирает различия между листьями и контейнерами, что может привести к ошибкам времени выполнения (например, вызов add на листе). Выбор между прозрачностью и безопасностью зависит от контекста.

Методика выполнения работы

Шаг 1. Изучение исходного кода

Получите индивидуальный вариант, содержащий классы для работы с иерархией (например, Product (товар) и Category (категория)). Исходный код, вероятно, обрабатывает их по отдельности, без единого интерфейса. Определите:

- Какие операции нужно выполнять над всей иерархией (подсчёт цены, отрисовка, поиск)?
- Как сейчас клиент вызывает эти операции для категорий и товаров?
- Какие методы потребуются для обхода дерева?

Шаг 2. Определение общего интерфейса

Создайте интерфейс CatalogComponent (или Component). Определите методы:

- double getPrice() (для товаров – цена товара, для категорий – сумма цен всех вложенных элементов).
- String getName().
- void print(int indent) или String renderTree() для отрисовки.
- Опционально: void add(CatalogComponent component), void remove(CatalogComponent component).

Шаг 3. Реализация листового узла (Leaf)

Создайте класс Product, реализующий CatalogComponent:

- Поля: name, price
- Методы: getPrice() возвращает цену; print(int indent) выводит имя и цену с отступом.
- Методы add/remove выбрасывают UnsupportedOperationException.

Шаг 4. Реализация составного узла (Composite)

Создайте класс Category, реализующий CatalogComponent:

- Поле List<CatalogComponent> children.
- Метод add(CatalogComponent) добавляет в список.
- Метод remove(CatalogComponent) удаляет.

- Метод getPrice() суммирует цены всех детей (рекурсивно, через вызов child.getPrice()).
- Метод print(int indent) выводит название категории, затем для каждого ребёнка вызывает print(indent+2).

Шаг 5. Реализация операций над деревом

В клиентском коде создайте дерево (категории, подкатегории, товары). Продемонстрируйте:

- Подсчёт общей стоимости товаров в категории (без знания внутренней структуры).
- Отрисовку дерева в консоль.
- Поиск товара по имени (обход дерева).

Шаг 6. Тестирование

Напишите тесты:

- Создание пустой категории: цена = 0.
- Категория с одним товаром: цена = цене товара.
- Вложенные категории: суммарная цена корректна.
- Добавление/удаление элементов.
- Проверка, что вызов add на товаре выбрасывает исключение.

Шаг 7. Анализ компромиссов

В отчёте ответьте:

- Какие изменения потребовались в клиентском коде после внедрения Composite?
- Как изменилась сложность добавления новой операции (например, countItems())?
- Какие проблемы могут возникнуть при работе с большими деревьями (производительность)?

Пример выполнения задания

Исходный код (до рефакторинга)

```
public class Product {
```

```
    private String name;
    private double price;
    public Product(String name, double price) { ... }
    public double getPrice() { return price; }
    public String getName() { return name; }
}

public class Category {
    private String name;
    private List<Product> products = new ArrayList<>();
    private List<Category> subcategories = new ArrayList<>();
    public Category(String name) { ... }
    public void addProduct(Product p) { products.add(p); }
    public void addSubcategory(Category c) { subcategories.add(c); }
    public double getTotalPrice() {
        double total = 0;
        for (Product p : products) total += p.getPrice();
        for (Category c : subcategories) total += c.getTotalPrice();
        return total;
    }
}

// Клиент
Category electronics = new Category("Electronics");
Category laptops = new Category("Laptops");
laptops.addProduct(new Product("MacBook", 1500));
laptops.addProduct(new Product("Dell XPS", 1200));
electronics.addSubcategory(laptops);
double total = electronics.getTotalPrice();
```

Недостатки: клиент должен знать о двух типах контейнеров (products и subcategories), при добавлении нового типа (например, Service) придётся менять все методы.

Рефакторинг с Composite

```
// Component
public interface CatalogComponent {
    String getName();
    double getPrice();
    void print(int indent);
    default void add(CatalogComponent c) {
        throw new UnsupportedOperationException("Cannot add to leaf");
    }
    default void remove(CatalogComponent c) {
        throw new UnsupportedOperationException("Cannot remove from leaf");
    }
}

// Leaf
public class Product implements CatalogComponent {
    private String name;
    private double price;
    public Product(String name, double price) { this.name = name; this.price = price; }
    @Override public String getName() { return name; }
    @Override public double getPrice() { return price; }
    @Override public void print(int indent) {
        System.out.println(" ".repeat(indent) + name + " - $" + price);
    }
}
```

```
// Composite
public class Category implements CatalogComponent {
    private String name;
    private List<CatalogComponent> children = new ArrayList<>();
    public Category(String name) { this.name = name; }
    @Override public String getName() { return name; }
    @Override public double getPrice() {
        return
children.stream().mapToDouble(CatalogComponent::getPrice).sum();
    }
    @Override public void add(CatalogComponent c) { children.add(c); }
    @Override public void remove(CatalogComponent c) {
children.remove(c); }
    @Override public void print(int indent) {
        System.out.println(" ".repeat(indent) + "📁 " + name);
        for (CatalogComponent child : children) {
            child.print(indent + 2);
        }
    }
}
```

Клиентский код после рефакторинга

```
// Построение дерева
Category electronics = new Category("Electronics");
Category laptops = new Category("Laptops");
laptops.add(new Product("MacBook", 1500));
laptops.add(new Product("Dell XPS", 1200));
electronics.add(laptops);
electronics.add(new Product("Mouse", 25));
```

```
// Единообразные операции
double total = electronics.getPrice(); // 2725
System.out.println("Total: $" + total);
electronics.print(0);
// Вывод:
//  Electronics
//    Laptops
//     MacBook - $1500.0
//     Dell XPS - $1200.0
//   Mouse - $25.0
```

Дополнительная операция (поиск)

Добавим метод поиска товара по имени в интерфейс CatalogComponent:

```
default CatalogComponent findByName(String name) {
   return getName().equals(name) ? this : null;
}
```

Переопределим в Category для рекурсивного обхода:

```
java
@Override
public CatalogComponent findByName(String name) {
   if (getName().equals(name)) return this;
   for (CatalogComponent child : children) {
      CatalogComponent result = child.findByName(name);
      if (result != null) return result;
   }
   return null;
}
```

Тест

```
@Test
void testCompositePrice() {
    Category root = new Category("root");
    Product p1 = new Product("A", 10);
    Product p2 = new Product("B", 20);
    Category sub = new Category("sub");
    sub.add(p2);
    root.add(p1);
    root.add(sub);
    assertEquals(30, root.getPrice(), 0.001);
}
```

Компромиссы в примере

- **Плюсы:** клиентский код стал проще, добавление нового типа (например, ServicePackage) не требует изменения существующих категорий. Легко добавить новую операцию (например, countItems()), реализовав её в интерфейсе.
- **Минусы:** теперь каждый CatalogComponent можно добавить в категорию, даже другой Category, что допустимо. Но если случайно добавить категорию в товар (Product.add(category)), получим исключение времени выполнения. Безопасность типов снижена. Также увеличилось число классов.

Индивидуальные задания

Вариант 1. Файловая система: интерфейс FileSystemNode. Листья: File (имя, размер). Составные: Directory (список узлов). Реализовать подсчёт общего размера и отрисовку дерева (файлы и папки). Добавить метод поиска файлов с заданным расширением.

Вариант 2. Организационная структура: Employee (имя, зарплата) и Department (название, список сотрудников и подотделов). Composite считает общую зарплату по отделу, включая вложенные. Отрисовать иерархию.

Вариант 3. Меню ресторана: MenuItem (название, цена), MenuCategory (название, список блюд и подкатегорий). Рассчитать общую стоимость заказа (выбрать набор пунктов). Реализовать подсчёт количества вегетарианских блюд (добавить флаг isVegetarian).

Вариант 4. Система доставки: Package (вес, габариты) и Container (спись пакетов и контейнеров). Рассчитать общий вес. Отрисовать вложенность. Добавить ограничение на максимальный вес контейнера.

Вариант 5. Учебный план: Course (название, кредиты) и Program (название, список курсов и подпрограмм). Composite считает общее количество кредитов. Отрисовать структуру. Добавить метод поиска курса по названию.

Вариант 6. Компьютерная периферия: Device (название, потребляемая мощность) и PowerStrip (список устройств и других удлинителей). Рассчитать общую мощность. Добавить метод включения/выключения всех устройств.

Вариант 7. Система контроля версий: FileVersion (имя, автор, размер) и Commit (список файлов и подкоммитов). Composite считает общий размер изменений. Отрисовать дерево коммитов.

Вариант 8. UI-компоненты: Widget (координаты, отрисовка) и Container (список виджетов). Реализовать отрисовку (вывод HTML-тегов) и подсчёт общего количества виджетов.

Вариант 9. Сетевая топология: Host (IP, нагрузка) и Subnet (список хостов и подсетей). Composite рассчитывает суммарную нагрузку. Отрисовать иерархию в виде текстового дерева. Добавить метод поиска хоста по IP.

Вариант 10. Конфигурация приложения: ConfigValue (ключ, значение) и ConfigSection (список параметров и подразделов). Отрисовка в формате JSON-like. Подсчёт количества параметров.

Вариант 11. Генеалогическое дерево: Person (имя, возраст) и Family (родители и дети). Composite: семья содержит людей и другие семьи (например, родственные ветви). Рассчитать средний возраст.

Вариант 12. Каталог книг: Book (название, цена, жанр) и Bookshelf (список книг и полок). Composite считает общую стоимость. Добавить логирование количества книг в категории.

Вариант 13. Географическая иерархия: City (название, население) и Region (список городов и регионов). Composite рассчитывает общее население. Отрисовать дерево с уровнями.

Вариант 14. Система проектов: Task (название, время в часах) и Project (список задач и подпроектов). Composite считает общее время. Добавить метод поиска задач с временем > заданного.

Вариант 15. Каталог автомобилей: Car (модель, цена, мощность) и CarDealership (название, список автомобилей и дилерских центров). Composite считает среднюю цену. Отрисовать иерархию.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с указанием темы, ФИО студента, варианта.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттерна Composite, его структуры, вариаций (прозрачный/безопасный), примеры использования.
4. **Ход выполнения**:
   - Исходный код без Composite (из вашего варианта) с анализом проблем.
   - Реализованные интерфейс и классы (листинги).
   - Пример построения дерева и вызова операций.
5. **Результаты тестирования** – код модульных тестов (2–3 примера) и скриншоты выполнения.

6. **Оценка компромиссов** – обсуждение плюсов и минусов выбранного подхода, анализ производительности рекурсивных операций.
7. **Вывод** – итоги работы, сформированные компетенции.
8. **Ответы на контрольные вопросы** (развёрнутые).

Объём отчёта: 4–6 страниц.

Контрольные вопросы

1. В чём основная проблема, которую решает паттерн Composite?
2. Что такое «прозрачность» в Composite и как она влияет на безопасность типов?
3. Как в вашей реализации организована рекурсия при подсчёте цены? Где происходит остановка?
4. Почему методы add и remove в интерфейсе Component имеют пустую реализацию по умолчанию, которая выбрасывает исключение?
5. Как изменится структура Composite, если добавить новый тип узла (например, DiscountPackage)? Нужно ли изменять существующие классы?
6. Какие ограничения накладывает Composite на жизненный цикл дочерних объектов (кто владелец, удаление)?
7. Можно ли с помощью Composite реализовать циклические ссылки? Почему это опасно?
8. Как влияет глубина дерева на производительность операций (например, getPrice)? Сложность по времени и памяти.
9. Сравните паттерны Composite и Decorator. В чём их сходство и различие?
10. Какие шаблоны проектирования часто используются вместе с Composite? (Подсказка: Visitor, Iterator)

Лабораторная работа №6. Strategy и State: замена условных операторов

Цель работы: очистить и агрегировать код, устранив разветвлённую условную логику (if-else/switch) путём внедрения паттернов Strategy для взаимозаменяемых алгоритмов и State для управления поведением объекта в зависимости от его внутреннего состояния.

Задачи:

1. Проанализировать существующий код с множественными условными операторами для расчёта скидок и обработки статусов заказа.
2. Реализовать паттерн **Strategy** для семейства алгоритмов расчёта скидки (например, сезонная, промокод, клубная карта), обеспечив возможность подстановки любой стратегии во время выполнения.
3. Реализовать паттерн **State** для конечного автомата статусов заказа (например, NEW → PAID → SHIPPED → DELIVERED → CANCELLED), инкапсулируя поведение каждого состояния в отдельном классе.
4. Обеспечить динамическое переключение стратегий и состояний без изменения клиентского кода.
5. Написать модульные тесты для каждой стратегии и каждого состояния, проверяя корректность переходов и вычислений.
6. Сравнить подходы Strategy и State по назначению, структуре и области применения.
7. Оценить компромиссы (увеличение числа классов против устранения условной логики, гибкость против сложности).

Теоретическая часть

Проблема условных операторов

Длинные цепочки if-else или switch – один из самых распространённых «запахов» кода. Они:

- Нарушают принцип открытости/закрытости (добавление новой ветки требует изменения существующего класса).
- Затрудняют тестирование (нельзя изолированно проверить одну ветку).
- Приводят к дублированию кода, если похожие условия встречаются в нескольких местах.
- Делают код менее читаемым по мере роста числа условий.

Паттерны Strategy и State предлагают объектно-ориентированную альтернативу: вынос условной логики в отдельные иерархии классов.

Паттерн Strategy (Стратегия)

Назначение: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Strategy позволяет изменять алгоритм независимо от клиентов, которые его используют.

Структура:

- Strategy – интерфейс, общий для всех алгоритмов.
- ConcreteStrategyA, ConcreteStrategyB – конкретные реализации алгоритмов.
- Context – класс, который использует стратегию. Содержит ссылку на Strategy и метод для её смены.

Когда использовать:

- Нужно использовать разные варианты одного алгоритма в одном объекте.
- Есть много связанных условных операторов, выбирающих поведение.
- Не нужно, чтобы клиент знал о деталях реализации алгоритмов.

Пример: системы расчёта налогов, компрессии данных, валидации, маршрутизации.

Паттерн State (Состояние)

Назначение: позволяет объекту изменять своё поведение при изменении внутреннего состояния. Создаётся впечатление, что объект изменил свой класс.

Структура:

- State – интерфейс, определяющий методы, зависящие от состояния.
- ConcreteStateA, ConcreteStateB – реализации поведения для конкретных состояний. Каждое состояние обычно имеет ссылку на Context для возможности переключения.
- Context – объект, чьё состояние меняется. Содержит ссылку на текущее State и делегирует ему вызовы.

Важное отличие от Strategy: в State переходы между состояниями могут быть определены внутри самих состояний (автомат), тогда как в Strategy смена стратегии обычно инициируется клиентом.

Сравнение Strategy и State

Характеристика	**Strategy**	**State**
Инициатива смены	Клиент (или внешний код)	Само состояние (или контекст)
Количество «алгоритмов»	Фиксированный набор, клиент выбирает	Заранее известный конечный автомат
Зависимость от контекста	Минимальная (стратегия обычно не знает о контексте)	Высокая (состояние имеет ссылку на контекст для переходов)
Типичное применение	Расчёт скидок, сортировка, сжатие, валидация	Конечные автоматы: заказы, подключения, игровые состояния
Структурное различие	Стратегии не управляют переключением	Состояния сами управляют переходами

Важно: оба паттерна устраняют условные операторы, но в Strategy условная логика вынесена в момент выбора стратегии (например, фабрика стратегий), а в State – размазана по методам переходов между состояниями.

Компромиссы

- **Плюсы**:
  - Код становится более модульным и тестируемым (каждую стратегию/состояние можно проверить изолированно).
  - Добавление новой стратегии или состояния не требует изменения существующего кода (OCP).
  - Упрощается отладка и понимание логики.
- **Минусы**:
  - Увеличение числа классов (каждый вариант – отдельный класс).
  - Усложнение навигации по проекту (больше файлов).
  - Для State может быть избыточным, если состояний мало (2–3) и переходы простые.

Trade-off: для простых случаев (2–3 варианта) if-else может быть более читаемым. Strategy/State оправданы, когда число вариантов ≥4 и/или они могут расширяться в будущем.

Методика выполнения работы

Работа состоит из двух независимых частей: рефакторинг системы скидок (Strategy) и рефакторинг статусов заказа (State). Рекомендуется выполнять последовательно.

Часть 1. Паттерн Strategy (скидки)

Шаг 1.1. Изучение исходного кода

Получите индивидуальный вариант (часть «Расчёт скидок»). Найдите метод с if-else или switch, который вычисляет скидку в зависимости от типа клиента, сезона, промокода и т.п. Определите:

- Какие факторы влияют на скидку?
- Какие алгоритмы скидок существуют?

- Как часто добавляются новые типы скидок?

Шаг 1.2. Выделение интерфейса стратегии

Создайте интерфейс DiscountStrategy с методом double calculate(double originalPrice).

Шаг 1.3. Реализация конкретных стратегий

Для каждого типа скидки создайте отдельный класс:

- NoDiscountStrategy – скидка 0%.
- SeasonalDiscountStrategy – например, 20% летом.
- LoyaltyDiscountStrategy – скидка за баллы.
- PromoCodeDiscountStrategy – фиксированный процент или сумма.

Шаг 1.4. Интеграция в контекст

Модифицируйте класс Order или ShoppingCart, чтобы он принимал DiscountStrategy через конструктор или сеттер. В методе calculateTotal() вызывайте стратегию.

Шаг 1.5. Демонстрация подстановки

Покажите, как можно динамически подставить разные стратегии без изменения кода заказа.

Часть 2. Паттерн State (статусы заказа)

Шаг 2.1. Изучение исходного кода

Получите исходный код класса Order с полем status (например, строка или enum) и методами next(), cancel(), pay(), которые содержат switch по статусу.

Шаг 2.2. Определение интерфейса состояния

Создайте интерфейс OrderState с методами:

- void pay(Order order)
- void ship(Order order)
- void deliver(Order order)

- void cancel(Order order) Каждый метод должен либо выполнить действие, либо выбросить исключение, если действие недопустимо в данном состоянии.

Шаг 2.3. Реализация конкретных состояний

Для каждого статуса (NewState, PaidState, ShippedState, DeliveredState, CancelledState) создайте класс, реализующий OrderState. В методах:

- Изменяйте состояние заказа через order.setState(new NextState()).
- Выполняйте соответствующие действия (логирование, отправка уведомлений).

Шаг 2.4. Модификация контекста

Класс Order теперь хранит текущее состояние (объект OrderState), а методы pay(), ship(), deliver(), cancel() просто делегируют вызов текущему состоянию.

Шаг 2.5. Тестирование автомата

Продемонстрируйте корректные и некорректные переходы (например, попытка отменить доставленный заказ должна выбрасывать исключение).

Шаг 3. Тестирование

Напишите JUnit-тесты:

- Для каждой стратегии скидки (проверка корректного расчёта).
- Для каждого состояния (проверка допустимых и недопустимых переходов).

Шаг 4. Анализ компромиссов

В отчёте сравните исходный код (с условными операторами) и рефакторированный. Оцените:

- На сколько строк увеличился код?
- Сколько новых классов добавлено?
- Как изменилась тестируемость?
- Когда овчинка выделки не стоит?

Пример выполнения задания

Исходный код (проблемный)

```
// До рефакторинга (скидки)
public class Order {
    private double amount;
    private String customerType;
    private String season;
    private String promoCode;

    public double getTotal() {
        double discount = 0;
        if (customerType.equals("VIP")) {
            discount = 0.2;
        } else if (customerType.equals("REGULAR")) {
            discount = 0.05;
        } else if (season.equals("SUMMER")) {
            discount = 0.1;
        } else if (promoCode != null && promoCode.equals("SAVE10")) {
            discount = 0.1;
        }
        return amount * (1 - discount);
    }
}

// До рефакторинга (статусы)
public class OrderV2 {
    private String status; // "NEW", "PAID", "SHIPPED", "DELIVERED",
"CANCELLED"

    public void pay() {
```

```
        switch (status) {
            case "NEW":
                System.out.println("Payment accepted");
                status = "PAID";
                break;
            case "PAID":
                System.out.println("Already paid");
                break;
            default:
                throw new IllegalStateException("Cannot pay in " + status);
        }
    }

    public void ship() {
        switch (status) {
            case "PAID":
                System.out.println("Order shipped");
                status = "SHIPPED";
                break;
            default:
                throw new IllegalStateException("Cannot ship in " + status);
        }
    }
}
```

Рефакторинг с Strategy

```
// Интерфейс стратегии
public interface DiscountStrategy {
    double applyDiscount(double amount);
}
```

```
    // Конкретные стратегии
    public class VipDiscount implements DiscountStrategy {
        public double applyDiscount(double amount) { return amount * 0.8; }
    }
    public class RegularDiscount implements DiscountStrategy {
        public double applyDiscount(double amount) { return amount * 0.95; }
    }
    public class SummerDiscount implements DiscountStrategy {
        public double applyDiscount(double amount) { return amount * 0.9; }
    }
    public class PromoDiscount implements DiscountStrategy {
        private double percent;
        public PromoDiscount(double percent) { this.percent = percent; }
        public double applyDiscount(double amount) { return amount * (1 - percent); }
    }
    public class NoDiscount implements DiscountStrategy {
        public double applyDiscount(double amount) { return amount; }
    }

    // Контекст
    public class Order {
        private double amount;
        private DiscountStrategy discountStrategy;

        public Order(double amount, DiscountStrategy strategy) {
            this.amount = amount;
            this.discountStrategy = strategy;
        }
```

```
    public void setDiscountStrategy(DiscountStrategy strategy) {
        this.discountStrategy = strategy;
    }

    public double getTotal() {
        return discountStrategy.applyDiscount(amount);
    }
}

// Клиент
Order order = new Order(100, new VipDiscount());
System.out.println(order.getTotal()); // 80.0
order.setDiscountStrategy(new PromoDiscount(0.15));
System.out.println(order.getTotal()); // 85.0
```

Рефакторинг с State

```
// Интерфейс состояния
public interface OrderState {
    void pay(OrderContext order);
    void ship(OrderContext order);
    void deliver(OrderContext order);
    void cancel(OrderContext order);
}

// Конкретные состояния
public class NewState implements OrderState {
    public void pay(OrderContext order) {
        System.out.println("Payment accepted");
        order.setState(new PaidState());
```

```
    }
    public void ship(OrderContext order) {
        throw new IllegalStateException("Cannot ship unpaid order");
    }
    public void deliver(OrderContext order) {
        throw new IllegalStateException("Cannot deliver unpaid order");
    }
    public void cancel(OrderContext order) {
        System.out.println("Order cancelled");
        order.setState(new CancelledState());
    }
}

public class PaidState implements OrderState {
    public void pay(OrderContext order) {
        System.out.println("Already paid");
    }
    public void ship(OrderContext order) {
        System.out.println("Order shipped");
        order.setState(new ShippedState());
    }
    public void deliver(OrderContext order) {
        throw new IllegalStateException("Not shipped yet");
    }
    public void cancel(OrderContext order) {
        System.out.println("Refund processed, order cancelled");
        order.setState(new CancelledState());
    }
}
// ... аналогично ShippedState, DeliveredState, CancelledState
```

```
// Контекст
public class OrderContext {
    private OrderState state;
    private String id;

    public OrderContext(String id) {
        this.id = id;
        this.state = new NewState();
    }
    void setState(OrderState state) { this.state = state; }
    public void pay() { state.pay(this); }
    public void ship() { state.ship(this); }
    public void deliver() { state.deliver(this); }
    public void cancel() { state.cancel(this); }
}

// Использование
OrderContext order = new OrderContext("ORD-001");
order.pay();     // Payment accepted
order.ship();    // Order shipped
order.deliver(); // Order delivered
order.cancel();  // Exception: Cannot cancel delivered order
```

Тесты

```
@Test
void testStrategy() {
    Order order = new Order(100, new NoDiscount());
    assertEquals(100, order.getTotal(), 0.001);
    order.setDiscountStrategy(new VipDiscount());
```

```
    assertEquals(80, order.getTotal(), 0.001);
}

@Test
void testStateTransitions() {
    OrderContext order = new OrderContext("1");
    assertThrows(IllegalStateException.class, () -> order.ship());
    order.pay();
    order.ship();
    order.deliver();
    assertThrows(IllegalStateException.class, () -> order.cancel());
}
```

Компромиссы

- **До:** 1 класс Order с двумя switch-ами (~60 строк). **После:** 1 интерфейс + 5 стратегий + 1 класс Order (~80 строк) для Strategy; плюс 6 классов состояний + контекст (~120 строк) для State.
- Увеличение объёма кода в 2–3 раза, но каждый класс стал маленьким и понятным.
- Тестирование: теперь можно протестировать каждую стратегию и каждое состояние изолированно, без создания полного заказа.
- Добавление новой скидки – просто создать новый класс стратегии.
- Добавление нового состояния – новый класс и правка переходов в существующих состояниях.

Индивидуальные задания

Каждый вариант состоит из двух частей: **Система скидок (Strategy)** и **Статусы заказа/процесса (State)**.

Вариант 1. *Strategy:* скидки для интернет-магазина: без скидки, постоянный покупатель (5%), золотая карта (10%), сезонная (15%), промокод

WELCOME (20%). *State:* статусы документа: ЧЕРНОВИК → НА РАССМОТРЕНИИ → ОДОБРЕН → ОПУБЛИКОВАН; возможен отказ в рассмотрении (переход в ОТКЛОНЁН).

Вариант 2. *Strategy:* расчёт доставки: обычная (5% от суммы, мин 200 руб.), экспресс (15%, мин 500), самовывоз (0), международная (30%). *State:* статусы заявки на ремонт: НОВАЯ → ДИАГНОСТИКА → ОЖИДАНИЕ_ЗАПЧАСТЕЙ → РЕМОНТ → ВЫПОЛНЕН → ЗАКРЫТ.

Вариант 3. *Strategy:* расчёт налогов (для разных стран): Россия (20%), США (5% федеральный + 3% штата), Германия (19%), без налогов. *State:* статусы командировки: ЗАПЛАНИРОВАНА → ПОДТВЕРЖДЕНА → ОТПРАВЛЕН → ВОЗВРАЩЁН → ЗАКРЫТ; возможна отмена на любом этапе.

Вариант 4. *Strategy:* система кэширования: без кэша, LRU (Least Recently Used), LFU, TTL (время жизни) – реализовать имитацию алгоритмов (без реальной структуры). *State:* статусы сетевого соединения: DISCONNECTED → CONNECTING → CONNECTED → AUTHENTICATED → ERROR; переходы по таймауту.

Вариант 5. *Strategy:* валидация пароля: лёгкая (только длина>4), средняя (длина>6 + цифра), сильная (>8 + цифра + спецсимвол). *State:* статусы задачи в трекере: OPEN → IN_PROGRESS → REVIEW → TESTING → DONE → CLOSED; также REOPENED из DONE.

Вариант 6. *Strategy:* компрессия данных: без сжатия, GZIP, ZIP, LZ4 (имитация – просто разные постфиксы). *State:* статусы платежа: INITIATED → AUTHORIZED → CAPTURED → SETTLED → REFUNDED; возможен void (отмена) до захвата.

Вариант 7. *Strategy:* форматирование отчёта: JSON, XML, HTML, CSV. *State:* статусы выпуска ПО: PLAN → DEVELOP → TEST → RELEASE → DEPRECATED; возможен переход в MAINTENANCE.

Вариант 8. *Strategy:* расчёт бонусов: 1% от суммы, 5% при заказе >5000, 10% в день рождения, двойные бонусы для премиум. *State:* статусы

библиотечного абонемента: ACTIVE → OVERDUE → RENEWED → CLOSED → LOST.

Вариант 9. *Strategy:* алгоритмы сортировки (имитация): пузырёк, быстрая, слиянием, вставками. *State:* статусы сессии пользователя: GUEST → LOGGED_IN → IDLE → LOCKED → LOGGED_OUT; переход по таймауту.

Вариант 10. *Strategy:* определение приоритета заявки: низкий (7 дней), средний (3 дня), высокий (1 день), критический (1 час). *State:* статусы бронирования отеля: ЗАПРОШЕНО → ПОДТВЕРЖДЕНО → ЗАСЕЛЁН → ВЫСЕЛЕН → ОТМЕНЕНО.

Вариант 11. *Strategy:* способы оплаты: наличные, карта (2% комиссии), PayPal (3%), криптовалюта (1%). *State:* статусы заявки на кредит: ЗАПОЛНЕНА → ПРОВЕРКА → ОДОБРЕНА → ПОДПИСАНА → ВЫДАЧА СРЕДСТВ; возможен ОТКАЗ.

Вариант 12. *Strategy:* форматы экспорта данных: PDF, Excel, Word, Markdown. *State:* статусы мероприятия: ЗАПЛАНИРОВАНО → ИДЁТ РЕГИСТРАЦИЯ → ПРОВОДИТСЯ → ЗАВЕРШЕНО → ОТМЕНЕНО.

Вариант 13. *Strategy:* уведомления пользователя: email, sms, push, telegram. *State:* статусы пользователя в чате: OFFLINE → ONLINE → AWAY → BUSY → INVISIBLE.

Вариант 14. *Strategy:* конвертация валют (фиксированные курсы): USD→EUR, USD→RUB, EUR→USD, без конвертации. *State:* статусы модерации контента: PENDING → REVIEW → APPROVED → REJECTED → PUBLISHED.

Вариант 15. *Strategy:* реализация очереди задач: FIFO, LIFO (стек), приоритетная, round-robin. *State:* статусы жизненного цикла IoT-устройства: PROVISIONED → ACTIVATED → RUNNING → STANDBY → ERROR → DECOMMISSIONED.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттернов Strategy и State, их сравнение, диаграммы классов (можно в виде текста или изображений).
4. **Ход выполнения** (разделы для Strategy и State):
   - Исходный код с условными операторами (анализ проблем).
   - Реализация интерфейсов и конкретных классов.
   - Интеграция с контекстом.
   - Пример использования.
5. **Результаты тестирования** – код тестов и скриншоты выполнения.
6. **Сравнительный анализ** – таблица «было/стало» по метрикам (количество классов, строк кода, тестируемость).
7. **Оценка компромиссов** – обсуждение trade-offs для вашего варианта.
8. **Вывод** – достигнуты ли цели, сформированные навыки.
9. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Контрольные вопросы

1. В чём принципиальное различие между паттернами Strategy и State по способу смены алгоритма/поведения?

2. Какой принцип SOLID устраняют оба паттерна?

3. Приведите пример, когда лучше использовать Strategy, а когда – State.

4. Как реализовать выбор стратегии на основе внешних параметров (например, из конфигурации)?

5. Может ли State-автомат иметь циклы (переходы обратно)? Приведите пример.

6. Как в паттерне State обеспечить выполнение действия при входе в состояние или выходе из него?

7. В чём недостаток обоих паттернов с точки зрения количества классов? Когда это оправдано?

8. Как паттерны Strategy и State влияют на тестируемость по сравнению с if-else?

9. Можно ли комбинировать Strategy и State? (Например, стратегия внутри состояния)

10. Что такое «контекст» в этих паттернах и какова его ответственность?

Лабораторная работа №7. Command и Observer: очередь задач и событийная шина

Цель работы: очистить и агрегировать код, разделив отправителя и получателя команд (Command), организовав отмену/повтор действий и поддержку макрокоманд, а также создать событийную шину (Observer) для слабосвязанной подписки на события внутри приложения.

Задачи:

1. Проанализировать существующий код с прямыми вызовами методов разных сервисов (отправитель привязан к получателю).
2. Реализовать паттерн **Command**: инкапсулировать действия (например, создание заказа, обновление статуса) в отдельные классы с единым интерфейсом.
3. Реализовать механизм очереди команд (Command Queue) и поддержку отмены/повтора (undo/redo) через стек выполненных команд.
4. Реализовать паттерн **MacroCommand** (макрокоманда) для группового выполнения нескольких команд как одной.
5. Реализовать паттерн **Observer** в виде простой событийной шины (Event Bus) внутри приложения для слабосвязанного реагирования на изменения (например, изменение статуса заказа, добавление товара).
6. Связать Command и Observer: команды после выполнения публикуют события в шину, на которые подписаны различные слушатели (логирование, уведомления, обновление UI).
7. Написать модульные тесты для команд (отмена, выполнение, макрокоманды) и для событийной шины.
8. Оценить компромиссы: увеличение числа классов против гибкости, производительность очереди команд против прямых вызовов.

Теоретическая часть

Проблема прямого связывания

В приложениях часто присутствует жёсткая связь между инициатором действия (например, контроллером) и исполнителем (сервисом). Это затрудняет:

- Добавление отмены/повтора (undo/redo)
- Логирование всех действий
- Отложенное выполнение (очереди)
- Комбинирование действий в макросы
- Слабое связывание компонентов для нотификации

Паттерны Command и Observer решают эти проблемы, но с разных сторон: Command инкапсулирует сам запрос, Observer – распространение изменений.

Паттерн Command (Команда)

Назначение: превращает запросы в объекты, позволяя передавать их как аргументы, ставить в очередь, логировать, поддерживать отмену операций.

Структура:

- Command – интерфейс с методом execute() и опциональным undo().
- ConcreteCommand – реализует метод execute(), вызывая методы получателя (receiver).
- Receiver – класс, который знает, как выполнить операцию (необязателен, можно встроить логику в команду).
- Invoker – вызывает команду (например, кнопка, очередь).
- Client – создаёт конкретные команды и связывает их с получателем.

Дополнительные вариации:

- MacroCommand – команда, содержащая список других команд, выполняет их последовательно.
- Команды с поддержкой undo() и redo() (требуют хранения состояния или обратного действия).

Применение: панели инструментов, транзакции, планировщики, история действий.

Паттерн Observer (Наблюдатель) и Event Bus

Назначение: определяет механизм подписки, позволяющий одним объектам (наблюдателям) следить за изменениями других объектов (субъектов).

Классическая структура:

- Subject – хранит список наблюдателей, методы attach(), detach(), notify().
- Observer – интерфейс с методом update().
- ConcreteObserver – реагирует на изменения.

Event Bus (шина событий) – вариация Observer:

- Централизованная шина, управляющая подписками и рассылкой событий.
- События – это объекты с типом (классом) и данными.
- Наблюдатели подписываются на типы событий, шина рассылает события всем заинтересованным.
- Преимущество: полностью развязывает издателя и подписчика (издатель не знает о подписчиках, только о шине).

Отличие Command от Observer: Command инкапсулирует действие (издатель вызывает команду, команда выполняет работу). Observer уведомляет об уже произошедшем событии (издатель публикует событие, подписчики реагируют). Команды могут публиковать события после своего выполнения – так они связываются естественным образом.

Связка Command + Observer

Типичный архитектурный приём:

1. Пользователь инициирует действие → создаётся команда.
2. Команда помещается в очередь (Invoker) и выполняется.
3. Команда вызывает методы получателя (сервиса), меняя состояние.
4. После выполнения команда публикует событие (например, OrderCreatedEvent) в Event Bus.

5. Все подписчики (логгер, уведомления, статистика) реагируют на событие.

Это реализует **CQRS** на микроуровне: команды изменяют состояние, события информируют об изменениях.

Компромиссы

Аспект	**Прямой вызов**	**Command**	**Observer (Event Bus)**
Связность	Высокая	Средняя (клиент знает о командах)	Низкая (через шину)
Отмена/повтор	Нет	Да	Не применимо
Очередование	Нет	Да	Не применимо
Логирование действий	Ручное	Централизованное (Invoker)	Через события
Количество классов	Мало	Много (каждая команда – класс)	Умеренно
Сложность отладки	Низкая	Средняя (больше классов)	Высокая (асинхронность событий)

Trade-offs: Command окупается, когда нужна отмена/история/логирование действий. Event Bus – когда много независимых реакций на события. Вместе они обеспечивают гибкую архитектуру, но требуют дисциплины и увеличивают порог входа.

Методика выполнения работы

Работа выполняется в два этапа: сначала реализация Command (с очередью и undo/redo), затем Observer (Event Bus) и их связывание.

Часть 1. Паттерн Command (команды над заказом)

Шаг 1.1. Исходный код (проблемный)

Допустим, есть класс OrderService с методами:

- void createOrder(OrderData data)
- void updateStatus(String orderId, Status newStatus)
- void cancelOrder(String orderId)

Контроллер вызывает их напрямую. Нет возможности отменить действие или поставить в очередь.

Шаг 1.2. Проектирование интерфейса Command

```
interface Command {
   void execute();
   void undo();
   String getName(); // для логирования
}
```

Шаг 1.3. Реализация конкретных команд

Каждая команда принимает в конструкторе получателя (OrderService) и необходимые параметры. В execute() вызывает метод получателя, в undo() – обратное действие (например, удалить заказ, вернуть статус).

Шаг 1.4. Инвокер с очередью и историей

Создайте класс CommandInvoker:

- Очередь команд (Queue<Command>).
- Стек выполненных команд для undo (Stack<Command>).
- Стек отменённых для redo (Stack<Command>).
- Метод addCommand(Command) – добавляет в очередь.
- Метод processQueue() – выполняет все команды из очереди поочерёдно, помещая их в undoStack.
- Методы undo() и redo() для отмены/повтора последней команды.

Шаг 1.5. Макрокоманда

Реализуйте MacroCommand implements Command, который содержит список команд. Его execute() выполняет все подкоманды, undo() отменяет в обратном порядке.

Часть 2. Observer и Event Bus

Шаг 2.1. События

Определите классы событий (можно один класс с полем type, но лучше иерархию):

- OrderCreatedEvent
- OrderStatusChangedEvent
- OrderCancelledEvent

Шаг 2.2. Реализация Event Bus

```
public interface EventBus {
    void publish(Object event);
    void subscribe(Class<?> eventType, EventHandler handler);
    void unsubscribe(Class<?> eventType, EventHandler handler);
}
```

Или используйте аннотации и рефлексию (упрощённо: Map<Class<?>, List<EventHandler>>).

Шаг 2.3. Подписчики

Реализуйте несколько подписчиков:

- LoggingHandler – логирует каждое событие.
- NotificationHandler – отправляет уведомление (имитация).
- AnalyticsHandler – собирает статистику.

Часть 3. Связывание Command и Event Bus

Модифицируйте команды: после успешного вызова receiver команда публикует соответствующее событие в Event Bus (который передаётся в команду через конструктор или внедряется).

Пример:

```
class CreateOrderCommand implements Command {
    private OrderService receiver;
    private OrderData data;
    private EventBus eventBus;
    private String createdOrderId;

    public void execute() {
        createdOrderId = receiver.createOrder(data);
        eventBus.publish(new OrderCreatedEvent(createdOrderId, data));
    }
    //...
}
```

Теперь любые подписчики Event Bus будут автоматически реагировать на выполнение команд.

Шаг 4. Тестирование

Напишите тесты:

- Выполнение одиночной команды и проверка вызова получателя.
- Отмена и повтор команды.
- Макрокоманда: выполнение нескольких команд и отмена всех.
- Очередь: порядок выполнения.
- Event Bus: подписка, рассылка, отписка.
- Интеграционный тест: после выполнения команды подписчик получил событие.

Шаг 5. Анализ компромиссов

Оцените, насколько увеличилась сложность кода, но какие возможности появились (отмена, событийная шина, слабая связанность).

Пример выполнения задания

Исходный код (проблемный)

```
// Простой сервис
public class OrderService {
    private Map<String, Order> orders = new HashMap<>();
    public String createOrder(OrderData data) {
        String id = UUID.randomUUID().toString();
        orders.put(id, new Order(id, data));
        return id;
    }
    public void updateStatus(String id, Status status) {
        orders.get(id).setStatus(status);
    }
    public void cancelOrder(String id) {
        orders.remove(id);
    }
}

// Вызов в контроллере
public class OrderController {
    private OrderService service = new OrderService();
    public void create(OrderData data) {
        String id = service.createOrder(data);
        System.out.println("Created: " + id);
        // логирование, уведомления  – всё здесь, жёсткая связь
```

```
    }
}
```

Шаг 1. Команды

```
public interface Command {
    void execute();
    void undo();
}

// Команда создания заказа
public class CreateOrderCommand implements Command {
    private OrderService receiver;
    private OrderData data;
    private String createdId;
    private EventBus eventBus;

    public CreateOrderCommand(OrderService receiver, OrderData data,
EventBus eventBus) {
        this.receiver = receiver;
        this.data = data;
        this.eventBus = eventBus;
    }

    @Override
    public void execute() {
        createdId = receiver.createOrder(data);
        eventBus.publish(new OrderCreatedEvent(createdId, data));
    }

    @Override
```

```
        public void undo() {
            if (createdId != null) {
                receiver.cancelOrder(createdId);
                eventBus.publish(new OrderCancelledEvent(createdId, "undo"));
            }
        }
    }

    // Команда изменения статуса (пример)
    public class ChangeStatusCommand implements Command {
        private OrderService receiver;
        private String orderId;
        private Status newStatus;
        private Status oldStatus;
        private EventBus eventBus;

        public ChangeStatusCommand(OrderService receiver, String orderId,
Status newStatus, EventBus eventBus) {
            this.receiver = receiver;
            this.orderId = orderId;
            this.newStatus = newStatus;
            this.eventBus = eventBus;
        }

        @Override
        public void execute() {
            oldStatus = receiver.getStatus(orderId);
            receiver.updateStatus(orderId, newStatus);
            eventBus.publish(new OrderStatusChangedEvent(orderId, oldStatus,
newStatus));
```

```
        }

        @Override
        public void undo() {
            receiver.updateStatus(orderId, oldStatus);
            eventBus.publish(new OrderStatusChangedEvent(orderId, newStatus,
oldStatus));
        }
    }
```

Шаг 2. Инвокер с очередью и undo/redo

```
    public class CommandInvoker {
        private Queue<Command> queue = new LinkedList<>();
        private Stack<Command> undoStack = new Stack<>();
        private Stack<Command> redoStack = new Stack<>();

        public void addCommand(Command cmd) {
            queue.add(cmd);
        }

        public void processQueue() {
            while (!queue.isEmpty()) {
                Command cmd = queue.poll();
                cmd.execute();
                undoStack.push(cmd);
                redoStack.clear(); // новая команда сбрасывает redo
            }
        }

        public void undo() {
```

```
        if (!undoStack.isEmpty()) {
            Command cmd = undoStack.pop();
            cmd.undo();
            redoStack.push(cmd);
        }
    }

    public void redo() {
        if (!redoStack.isEmpty()) {
            Command cmd = redoStack.pop();
            cmd.execute();
            undoStack.push(cmd);
        }
    }
}
```

Шаг 3. Макрокоманда

```
public class MacroCommand implements Command {
    private List<Command> commands = new ArrayList<>();

    public void addCommand(Command cmd) { commands.add(cmd); }

    @Override
    public void execute() {
        for (Command cmd : commands) cmd.execute();
    }

    @Override
    public void undo() {
        for (int i = commands.size()-1; i >= 0; i--) commands.get(i).undo();
```

```
    }
}
```

Шаг 4. Event Bus (простая реализация)

```
@FunctionalInterface
interface EventHandler {
    void handle(Object event);
}

public class SimpleEventBus implements EventBus {
    private final Map<Class<?>, List<EventHandler>> subscribers = new HashMap<>();

    @Override
    public void subscribe(Class<?> eventType, EventHandler handler) {
        subscribers.computeIfAbsent(eventType, k -> new ArrayList<>()).add(handler);
    }

    @Override
    public void unsubscribe(Class<?> eventType, EventHandler handler) {
        List<EventHandler> list = subscribers.get(eventType);
        if (list != null) list.remove(handler);
    }

    @Override
    public void publish(Object event) {
        List<EventHandler> handlers = subscribers.get(event.getClass());
        if (handlers != null) {
            for (EventHandler h : handlers) h.handle(event);
```

```
        }
    }
}
```

Шаг 5. Пример использования

```
// Инициализация
OrderService service = new OrderService();
SimpleEventBus eventBus = new SimpleEventBus();

// Подписчики
eventBus.subscribe(OrderCreatedEvent.class, e ->
    System.out.println("LOG: Order created " +
((OrderCreatedEvent)e).getOrderId()));
eventBus.subscribe(OrderStatusChangedEvent.class, e ->
    System.out.println("NOTIFY: Status changed"));

CommandInvoker invoker = new CommandInvoker();

// Создаём и выполняем команды
OrderData data = new OrderData("Laptop", 1);
Command create = new CreateOrderCommand(service, data, eventBus);
Command change = new ChangeStatusCommand(service, "order-123",
Status.PAID, eventBus);

invoker.addCommand(create);
invoker.addCommand(change);
invoker.processQueue(); // выполняет обе команды, публикует события

invoker.undo(); // отменяет последнюю (смена статуса)
invoker.undo(); // отменяет создание заказа
```

```
// Макрокоманда
MacroCommand macro = new MacroCommand();
macro.addCommand(create);
macro.addCommand(change);
macro.execute();
macro.undo(); // отменяет всё
```

Тесты

```
@Test
void testCommandUndo() {
    OrderService service = new OrderService();
    EventBus bus = new SimpleEventBus();
    CreateOrderCommand cmd = new CreateOrderCommand(service, new OrderData("test",1), bus);
    cmd.execute();
    String id = service.getOrderIds().iterator().next();
    assertNotNull(service.getOrder(id));
    cmd.undo();
    assertNull(service.getOrder(id));
}

@Test
void testEventBus() {
    SimpleEventBus bus = new SimpleEventBus();
    List<String> logs = new ArrayList<>();
    bus.subscribe(OrderCreatedEvent.class, e -> logs.add("created"));
    bus.publish(new OrderCreatedEvent("1", null));
    assertEquals(1, logs.size());
}
```

Компромиссы

- **Добавлено классов:** 1 интерфейс Command, 2+ команд, 1 Invoker, MacroCommand, SimpleEventBus, 3+ событий, подписчики. Объём кода вырос в 3–4 раза.
- **Гибкость:** можно легко добавить отмену, любой новый тип команды, новый подписчик без изменения существующего кода.
- **Производительность:** очередь команд добавляет небольшие накладные расходы (вызовы методов, стек). В высоконагруженной системе команды лучше выполнять асинхронно.
- **Отладка:** сложнее отследить поток выполнения, особенно с событиями.
- **Оправданность:** для приложений с поддержкой отмены/повтора, макросов и множества побочных реакций – очень полезно.

Индивидуальные задания

Каждый вариант требует реализовать **команды для работы с сущностью** (например, заказ, пользователь, документ), **макрокоманду**, **очередь с undo/redo** и **Event Bus** с 2–3 подписчиками.

Вариант 1. Управление библиотекой: команды BorrowBookCommand, ReturnBookCommand, ReserveBookCommand. События: BookBorrowed, BookReturned. Подписчики: логирование, обновление каталога, уведомление читателя.

Вариант 2. Система заказов в кафе: команды AddToOrderCommand, RemoveFromOrderCommand, CompleteOrderCommand. События: OrderItemAdded, OrderCompleted. Подписчики: печать чека, обновление кухни, уведомление официанта.

Вариант 3. Редактор текста (упрощённо): команды InsertTextCommand, DeleteTextCommand, ReplaceTextCommand. События: TextChanged. Подписчики: автосохранение, подсветка синтаксиса, проверка орфографии.

Вариант 4. Управление пользователями: команды CreateUserCommand, UpdateUserCommand, DeleteUserCommand. События: UserCreated, UserDeleted. Подписчики: отправка email, аудит, очистка кэша.

Вариант 5. Система билетов: команды BookTicketCommand, CancelTicketCommand, ChangeSeatCommand. События: TicketBooked, TicketCancelled. Подписчики: логирование, обновление схемы зала, отправка SMS.

Вариант 6. Интернет-магазин: команды AddToCartCommand, RemoveFromCartCommand, ApplyDiscountCommand. События: CartUpdated. Подписчики: пересчёт итога, сохранение в БД, аналитика.

Вариант 7. Управление задачами (Todo): команды AddTaskCommand, CompleteTaskCommand, MoveTaskCommand. События: TaskAdded, TaskCompleted. Подписчики: обновление UI, сохранение в файл, уведомление в чат.

Вариант 8. Банковские транзакции: команды DepositCommand, WithdrawCommand, TransferCommand. События: TransactionExecuted. Подписчики: выписка, антифрод, обновление баланса.

Вариант 9. Система контроля версий (упрощённо): команды CreateBranchCommand, CommitCommand, MergeCommand. События: BranchCreated, Committed. Подписчики: обновление графа, проверка конфликтов, уведомление команды.

Вариант 10. Бот для чата: команды SendMessageCommand, EditMessageCommand, DeleteMessageCommand. События: MessageSent. Подписчики: логирование, проверка цензуры, сохранение истории.

Вариант 11. Система бронирования отелей: команды BookRoomCommand, CancelBookingCommand, ChangeDatesCommand. События: BookingCreated, BookingCancelled. Подписчики: отправка подтверждения, блокировка комнаты, обновление календаря.

Вариант 12. Управление складом: команды ReceiveGoodsCommand, ShipGoodsCommand, TransferGoodsCommand. События: StockChanged.

Подписчики: проверка минимального остатка, уведомление закупщика, обновление отчётов.

Вариант 13. Система опросов: команды CreatePollCommand, VoteCommand, ClosePollCommand. События: PollCreated, VoteCast. Подписчики: подсчёт голосов, публикация результатов, анонсирование.

Вариант 14. Управление проектами: команды AddTaskCommand, AssignTaskCommand, ChangeDeadlineCommand. События: TaskAssigned, DeadlineChanged. Подписчики: обновление календаря, отправка уведомлений, расчёт критического пути.

Вариант 15. Онлайн-обучение: команды EnrollStudentCommand, SubmitHomeworkCommand, GradeAssignmentCommand. События: StudentEnrolled, HomeworkSubmitted. Подписчики: обновление рейтинга, уведомление преподавателя, генерация сертификата.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с указанием темы, ФИО, варианта.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттернов Command (включая MacroCommand) и Observer (Event Bus), диаграммы классов.
4. **Ход выполнения** (разделы Command и Observer):
   - Исходный код без паттернов.
   - Реализация команд, инвокера, очереди, undo/redo.
   - Реализация макрокоманды.
   - Реализация Event Bus и подписчиков.
   - Связывание команд с шиной.
5. **Результаты тестирования** – код тестов (JUnit) и скриншоты.
6. **Пример использования** (клиентский код, демонстрирующий все возможности).

7. **Оценка компромиссов** – анализ trade-offs для вашего варианта.
8. **Вывод** – достигнуты ли цели, сформированные навыки.
9. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Объём отчёта: 5–7 страниц.

Контрольные вопросы

1. В чём преимущество паттерна Command перед прямым вызовом методов?

2. Как реализовать поддержку отмены (undo) для команды, которая изменяет состояние объекта?

3. Что такое макрокоманда (MacroCommand) и зачем она нужна?

4. В каких случаях используется очередь команд (Command Queue) вместо немедленного выполнения?

5. Каким образом Event Bus уменьшает связанность компонентов по сравнению с классическим Observer (Subject → Observer)?

6. Как в вашей реализации команды публикуют события? Нужно ли команде знать о Event Bus?

7. Какие проблемы могут возникнуть при циклической публикации событий (событие порождает команду, которая порождает событие)?

8. Сравните сложность реализации undo/redo через Command и через сохранение полного состояния объекта (Memento).

9. Как тестировать команду, которая зависит от внешнего сервиса (например, OrderService)? Какие паттерны тестирования применяются?

10. Опишите сценарий, в котором совместное использование Command и Observer создаёт «конвейер» обработки. Как это влияет на производительность и отладку?

Лабораторная работа №8. Chain of Responsibility для валидации и фильтров

Цель работы: очистить и агрегировать код валидации и фильтрации данных путём внедрения паттерна Chain of Responsibility, позволяющего динамически строить цепочки обработчиков, где каждый обработчик либо обрабатывает запрос, либо передаёт его дальше, а также изолировать логику проверок друг от друга.

Задачи:

1. Проанализировать существующий код с последовательными вызовами валидаторов или множественными вложенными if-else для проверки данных.
2. Реализовать паттерн **Chain of Responsibility**: общий интерфейс обработчика, конкретные обработчики для разных правил, механизм связи в цепочку.
3. Обеспечить возможность прерывания цепочки при первой ошибке (fail-fast) или накопления всех ошибок (агрегация).
4. Реализовать динамическую сборку цепочки (порядок обработчиков может меняться в зависимости от конфигурации).
5. Создать обработчики для трёх типов валидации (например, проверка на пустоту → проверка формата → бизнес-правило).
6. Написать модульные тесты для отдельных обработчиков и цепочки в сборе.
7. Оценить компромиссы: гибкость против накладных расходов, лёгкость добавления нового правила против сложности отладки.

Теоретическая часть

Проблема последовательных проверок

В приложениях часто требуется выполнить ряд проверок над входными данными (валидация формы, фильтрация запроса, авторизация, логирование). Типичные подходы:

- **Длинный метод с** if-else**:** все проверки в одном методе. Нарушает OCP (открытости/закрытости), трудно добавлять новые проверки, изменять порядок.
- **Массив валидаторов:** цикл по списку валидаторов. Уже лучше, но вызывает валидаторы один за другим, не давая гибкого контроля передачи запроса (кто-то может обработать и остановить цепочку).

Паттерн Chain of Responsibility решает проблему, превращая каждую проверку в самостоятельный объект, который либо обрабатывает запрос, либо передаёт его следующему.

Паттерн Chain of Responsibility (Цепочка обязанностей)

Назначение: позволяет избежать жёсткой привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким объектам. Объекты связываются в цепочку, и запрос передаётся по цепочке, пока один из объектов не обработает его.

Структура:

- Handler – интерфейс, объявляющий метод handle(Request) и ссылку на следующий обработчик (setNext()).
- ConcreteHandler1, ConcreteHandler2 – реализуют проверку: если могут обработать – выполняют действие и возможно останавливают цепочку; если нет – передают дальше.
- Client – строит цепочку и передаёт запрос первому обработчику.

Варианты реализации:

- **Цепочка с остановкой (fail-fast):** при обнаружении ошибки обработчик возвращает результат с ошибкой и не вызывает следующий.

- **Цепочка-фильтр:** каждый обработчик может модифицировать запрос или данные, передавая их дальше (например, фильтры в веб-фреймворках).
- **Агрегирующая цепочка:** все обработчики выполняются, накапливая результаты или ошибки.

Применение: системы валидации, фильтры аутентификации, логирование, цепочки команд в GUI, пайплайны обработки данных.

Связь с другими паттернами

- **Decorator** – также строит цепочку, но декораторы оборачивают один компонент и добавляют поведение, а цепочка ответственности передаёт запрос по независимым обработчикам.
- **Command** – команда инкапсулирует действие, цепочка решает, кто его выполнит.
- **Composite** – оба работают с иерархией, но Composite строит деревья «часть-целое», а цепочка – линейную последовательность.

Компромиссы

Аспект	if-else или массив проверок	Chain of Responsibility
Гибкость добавления проверок	Низкая (правка существующего кода)	Высокая (новый обработчик + регистрация)
Изменение порядка проверок	Правка кода или порядка в массиве	Перестроение цепочки (динамически)
Переиспользование проверок	Низкое	Высокое (каждый обработчик независим)
Производительность	Хорошая (один цикл/условия)	Немного хуже (вызовы методов, объекты)
Сложность кода	Низкая (всё в одном месте)	Средняя (много мелких классов)

Trade-offs: цепочка окупается при наличии 4+ различных проверок, которые могут меняться динамически (например, разные наборы правил для разных типов форм). Для простого набора фиксированных проверок из 2–3 правил лучше оставить прямое выполнение.

Методика выполнения работы

Шаг 1. Изучение исходного кода

Получите индивидуальный вариант – класс с методом валидации, содержащим несколько последовательных if-else или вызовов валидаторов. Определите:

- Какие проверки выполняются? (например, обязательность, длина, формат email, бизнес-правило)
- В каком порядке они должны идти?
- Нужно ли останавливаться при первой ошибке или собирать все ошибки?

Шаг 2. Проектирование интерфейса обработчика

```
public interface Validator<T> {
   Validator<T> setNext(Validator<T> next);
   ValidationResult validate(T data);
}
```

Или, если нужна модификация данных:

```
public interface Filter<T> {
   T process(T input, FilterChain chain);
}
```

Для валидации удобно определить класс ValidationResult, содержащий флаг success и список ошибок.

Шаг 3. Реализация базового абстрактного обработчика (опционально)

Создайте AbstractValidator<T>, который хранит ссылку на следующий обработчик и реализует setNext(). Метод validate() будет проверять текущим обработчиком, и если ошибок нет, передавать дальше.

Шаг 4. Реализация конкретных обработчиков

Для каждого правила создайте класс:

- NotNullValidator – проверяет, что объект не null.
- NotEmptyValidator – для строк/коллекций.
- EmailFormatValidator – регулярное выражение.
- RangeValidator – числовой диапазон.
- BusinessRuleValidator – специфическое правило (например, сумма заказа не превышает лимит).

Каждый обработчик возвращает ValidationResult: если ошибка – добавляет сообщение и либо останавливает цепочку (fail-fast), либо продолжает (накопление).

Шаг 5. Динамическая сборка цепочки

Создайте класс ValidationChainBuilder, который позволяет добавлять обработчики в произвольном порядке и собирать цепочку:

```
ValidationChainBuilder<Order> builder = new ValidationChainBuilder<>();
Validator<Order> chain = builder
    .add(new NotNullValidator<>())
    .add(new OrderItemsNotEmptyValidator())
    .add(new TotalPriceLimitValidator(10000))
    .build();
```

Шаг 6. Клиентский код

Клиент получает цепочку и вызывает validate(data), работая с результатом (список ошибок или успех).

Шаг 7. Тестирование

Напишите тесты:

- Каждый обработчик тестируется изолированно.
- Цепочка: валидные данные – успех.
- Невалидные данные – ошибка на первом же нарушении (или все ошибки).
- Динамическая сборка: изменение порядка обработчиков влияет на результат.

Шаг 8. Анализ компромиссов

Оцените увеличение количества классов, сложность добавления нового правила, влияние на производительность.

Пример выполнения задания

Исходный код (проблемный)

```
public class OrderValidator {
    public List<String> validate(Order order) {
        List<String> errors = new ArrayList<>();

        // Проверка 1: заказ не null
        if (order == null) {
            errors.add("Order is null");
            return errors;
        }

        // Проверка 2: список товаров не пуст
```

```
        if (order.getItems() == null || order.getItems().isEmpty()) {
            errors.add("Order must have at least one item");
            return errors;
        }

        // Проверка 3: сумма заказа не превышает лимит
        double total = order.getItems().stream().mapToDouble(Item::getPrice).sum();
        if (total > 10000) {
            errors.add("Total price exceeds limit 10000");
            return errors;
        }

        // Проверка 4: email клиента валидный
        String email = order.getCustomerEmail();
        if (email == null || !email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
            errors.add("Invalid email format");
        }

        return errors;
    }
}
```

Проблемы: все проверки в одном методе, трудно переиспользовать, добавлять новые, менять порядок.

Реализация Chain of Responsibility

```
// Результат валидации
public class ValidationResult {
    private final List<String> errors = new ArrayList<>();
    private boolean stopped = false;
```

```
        public void addError(String error) { errors.add(error); }
        public void stop() { stopped = true; }
        public boolean isStopped() { return stopped; }
        public boolean isValid() { return errors.isEmpty() && !stopped; }
        public List<String> getErrors() { return
Collections.unmodifiableList(errors); }
    }

    // Интерфейс валидатора
    public interface Validator<T> {
        ValidationResult validate(T data);
        Validator<T> setNext(Validator<T> next);
    }

    // Абстрактный обработчик
    public abstract class AbstractValidator<T> implements Validator<T> {
        private Validator<T> next;

        @Override
        public Validator<T> setNext(Validator<T> next) {
            this.next = next;
            return next;
        }

        protected abstract void doValidate(T data, ValidationResult result);

        @Override
        public ValidationResult validate(T data) {
            ValidationResult result = new ValidationResult();
```

```
            doValidate(data, result);
            if (!result.isStopped() && next != null) {
                ValidationResult nextResult = next.validate(data);
                result.getErrors().addAll(nextResult.getErrors());
                if (nextResult.isStopped()) result.stop();
            }
            return result;
        }
    }

    // Конкретные обработчики
    public class NotNullValidator<T> extends AbstractValidator<T> {
        @Override
        protected void doValidate(T data, ValidationResult result) {
            if (data == null) {
                result.addError("Object is null");
                result.stop();
            }
        }
    }

    public class OrderItemsNotEmptyValidator extends AbstractValidator<Order> {
        @Override
        protected void doValidate(Order order, ValidationResult result) {
            if (order.getItems() == null || order.getItems().isEmpty()) {
                result.addError("Order must have at least one item");
                result.stop(); // критическая ошибка, дальше нет смысла проверять
            }
```

```
        }
    }

    public class TotalPriceLimitValidator extends AbstractValidator<Order> {
        private final double limit;
        public TotalPriceLimitValidator(double limit) { this.limit = limit; }

        @Override
        protected void doValidate(Order order, ValidationResult result) {
            double total = order.getItems().stream().mapToDouble(Item::getPrice).sum();
            if (total > limit) {
                result.addError("Total price exceeds limit " + limit);
                result.stop();
            }
        }
    }

    public class EmailFormatValidator extends AbstractValidator<Order> {
        @Override
        protected void doValidate(Order order, ValidationResult result) {
            String email = order.getCustomerEmail();
            if (email == null || !email.matches("^[A-Za-z0-9+_.-]+@(.+)$")) {
                result.addError("Invalid email format");
                // не останавливаем, так как это не критично для процесса? но можно и остановить
            }
        }
    }
```

```
// Строитель цепочки
public class ValidationChainBuilder<T> {
    private Validator<T> first;
    private Validator<T> last;

    public ValidationChainBuilder<T> add(Validator<T> validator) {
        if (first == null) {
            first = validator;
            last = validator;
        } else {
            last.setNext(validator);
            last = validator;
        }
        return this;
    }

    public Validator<T> build() {
        return first;
    }
}
```

Использование

```
Order order = new Order(/* данные */);

Validator<Order> chain = new ValidationChainBuilder<Order>()
    .add(new NotNullValidator<>())
    .add(new OrderItemsNotEmptyValidator())
    .add(new TotalPriceLimitValidator(10000))
    .add(new EmailFormatValidator())
    .build();
```

```
ValidationResult result = chain.validate(order);
if (!result.isValid()) {
    System.out.println("Validation failed: " + result.getErrors());
} else {
    System.out.println("Order is valid");
}
```

Вариант с модификацией данных (фильтры)

Если цепочка должна модифицировать данные (например, фильтры HTTP-запросов), интерфейс может быть другим:

```
public interface Filter<T> {
    T doFilter(T input, FilterChain chain);
}
public class LoggingFilter implements Filter<String> {
    public String doFilter(String input, FilterChain chain) {
        System.out.println("Before: " + input);
        String result = chain.doFilter(input);
        System.out.println("After: " + result);
        return result;
    }
}
```

Тестирование

```
@Test
void testNotNullValidator_stopsWhenNull() {
    Validator<Object> validator = new NotNullValidator<>();
    ValidationResult result = validator.validate(null);
    assertFalse(result.isValid());
    assertTrue(result.isStopped());
```

```
    assertEquals(1, result.getErrors().size());
}

@Test
void testChainWithMultipleErrors() {
    Order invalidOrder = new Order();
    invalidOrder.setItems(Collections.emptyList());
    invalidOrder.setCustomerEmail("bad");
    Validator<Order> chain = new ValidationChainBuilder<Order>()
        .add(new OrderItemsNotEmptyValidator())
        .add(new EmailFormatValidator())
        .build();
    ValidationResult result = chain.validate(invalidOrder);
    // Должна быть ошибка от первого (items empty), он остановил цепочку, так что EmailFormatValidator не вызван
    assertEquals(1, result.getErrors().size());
    assertTrue(result.getErrors().get(0).contains("item"));
}
```

Компромиссы в примере

- **До:** 1 класс валидатора (30 строк). **После:** 5 классов обработчиков, 2 вспомогательных класса (ValidationResult, AbstractValidator), строитель – около 120 строк (+300%).
- **Гибкость:** можно легко добавить новый, например, BlacklistedCustomerValidator, просто написав класс и добавив в цепочку.
- **Тестируемость:** каждый обработчик тестируется отдельно.
- **Производительность:** дополнительное создание объектов (ValidationResult) и вызовы методов. Для сотен тысяч валидаций может быть ощутимо.

- **Оправданность:** если валидация формы состоит из 3–4 правил, лучше остаться с простым методом. Если правила часто меняются, переиспользуются в разных формах, то Chain of Responsibility – хороший выбор.

Индивидуальные задания

Каждый вариант требует реализации цепочки валидации (или фильтров) для указанной сущности. Необходимо реализовать не менее 4–5 конкретных обработчиков, динамическую сборку цепочки, поддержку fail-fast (остановка при первой ошибке) или накопления (по требованию варианта).

Вариант 1. Валидация регистрации пользователя: обработчики – не пустое имя, длина пароля (≥8), наличие цифры в пароле, уникальность email (имитация проверки в «БД»), проверка номера телефона (формат). Собирать все ошибки.

Вариант 2. Валидация банковской транзакции: обработчики – сумма >0, достаточность средств (имитация), лимит на перевод (max 100000), проверка получателя (не тот же счёт), антифрод (сумма >50000 требует подтверждения). Останавливать при критической ошибке (недостаточно средств).

Вариант 3. Валидация заказа пиццы: размер (S/M/L), тесто (тонкое/пышное), начинки (минимум 1, максимум 5), адрес доставки (не пуст), телефон (формат). Останавливать при первой ошибке.

Вариант 4. Фильтры текстовых сообщений (цепочка модификации): удалить лишние пробелы, цензура (замена мата на ***), ограничение длины (обрезание до 140 символов), добавление подписи, экранирование HTML. Каждый фильтр модифицирует текст.

Вариант 5. Валидация продукта в каталоге: название не пусто, цена >0, артикул уникален (имитация БД), категория разрешена (список), изображение не слишком большое (имитация размера файла). Накопление всех ошибок.

Вариант 6. Авторизация запроса к API (цепочка проверок): проверка API-ключа (существует), проверка срока действия ключа, проверка IP (белый

список), проверка rate limit (не более 100 запросов/час), логирование. Останавливать при любой ошибке.

Вариант 7. Валидация отчёта перед генерацией: дата начала ≤ дата конца, период не более 31 дня, выбран хотя бы один формат, наличие прав на данные (роль ADMIN или MANAGER), размер данных не превышает 1M (имитация). Останавливать при первой ошибке.

Вариант 8. Фильтры для импорта CSV-строки: удаление BOM-символов, разбиение на колонки, проверка числа колонок (ожидается 5), преобразование типов (дата, число), пропуск пустых строк. Цепочка модифицирующая.

Вариант 9. Валидация комнаты в системе бронирования: название не пусто, вместимость (1–10), цена (≥0), наличие Wi-Fi (булево), описание не длиннее 500 символов. Накопление всех ошибок.

Вариант 10. Валидация заявки на кредит: возраст заёмщика (18–70), доход (≥ минимального), кредитная история (не плохая), сумма кредита (≤ 5×доход), цель кредита (из списка). Останавливать при критических (не прошел возраст).

Вариант 11. Фильтры для URL (перед отправкой запроса): encode пробелов, удаление дублирующихся слешей, добавление завершающего слеша (если путь пустой), нормализация параметров (сортировка), защита от XSS. Цепочка модифицирует URL.

Вариант 12. Валидация тестового задания (система проверки): компиляция (успех), длина кода не более 1000 строк, наличие комментариев, прохождение юнит-тестов (имитация), стиль кода (запрещены магические числа). Все проверки выполняются, результат – список ошибок.

Вариант 13. Валидация заказа в ресторане (доставка): время доставки (внутри часов работы), минимальная сумма заказа, наличие аллергенов (если отмечено), адрес в зоне доставки, способ оплаты разрешён. Останавливать при первой ошибке.

Вариант 14. Фильтры для логов (перед записью): маскировка паролей, обрезание слишком длинных строк (1000 символов), добавление временной метки, фильтрация уровня (только ERROR, если включен режим), перевод в нижний регистр. Цепочка модифицирует строку лога.

Вариант 15. Валидация конфигурации приложения (файл свойств): обязательные ключи присутствуют, числовые значения в диапазоне, enum-значения допустимы, пути к файлам существуют (имитация), нет циклических зависимостей. Накопление всех ошибок.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО студента, вариантом.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттерна Chain of Responsibility, его структуры, вариаций (fail-fast, агрегация, фильтры).
4. **Ход выполнения**:
   - Исходный код с проблемой (длинный метод валидации).
   - Интерфейсы и классы обработчиков.
   - Реализация цепочки и строителя.
   - Пример использования (сборка цепочки, вызов).
5. **Результаты тестирования** – код тестов (JUnit) для отдельных обработчиков и цепочки, скриншоты.
6. **Оценка компромиссов** – сравнение производительности, гибкости, сложности до и после.
7. **Вывод** – достигнуты ли цели, сформированные навыки.
8. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Контрольные вопросы

1. В чём основное отличие паттерна Chain of Responsibility от простого списка (массива) обработчиков?

2. Как в вашей реализации реализовано прерывание цепочки при ошибке (fail-fast)?

3. Что нужно изменить, чтобы цепочка не прерывалась, а собирала все ошибки?

4. Как можно динамически перестроить цепочку во время выполнения (например, сменить порядок проверок в зависимости от типа пользователя)?

5. Какие преимущества даёт использование строителя (ValidationChainBuilder) для сборки цепочки?

6. Как Chain of Responsibility связан с принципом единственной ответственности (SRP)?

7. В каких случаях использование данного паттерна может быть избыточным?

8. Как протестировать, что порядок обработчиков в цепочке влияет на результат? Приведите пример.

9. Как Chain of Responsibility взаимодействует с другими паттернами (например, Command или Decorator)?

10. Какие проблемы производительности могут возникнуть при очень длинной цепочке (десятки обработчиков)? Как их можно смягчить?

Лабораторная работа №9. Архитектурный рефакторинг к Clean Architecture (Ports & Adapters)

Цель работы: очистить и агрегировать код монолитного приложения путём отделения внешних зависимостей (базы данных, пользовательского интерфейса, внешних API) от бизнес-логики с использованием архитектурных паттернов Ports & Adapters (Hexagonal Architecture), реализовав чёткие границы через интерфейсы (порты) и перенеся бизнес-правила в изолированные use cases.

Задачи:

1. Проанализировать существующий «грязный» код, где бизнес-логика смешана с прямыми вызовами БД, UI-логикой и внешними сервисами.
2. Внедрить **порты (interfaces)** для основных операций (репозитории, отправка уведомлений, внешние API).
3. Перенести бизнес-логику в отдельные **use cases (интеракторы)**, которые зависят только от портов, а не от конкретных реализаций.
4. Реализовать **адаптеры** для БД (например, JDBC или in-memory) и других внешних зависимостей.
5. Настроить ручное **Dependency Injection** (внедрение зависимостей) для сборки приложения без использования контейнеров.
6. Продемонстрировать возможность замены адаптера (например, переключение с реальной БД на in-memory репозиторий) без изменения use case.
7. Написать модульные тесты для use cases с использованием **Test Doubles** для портов (репозиториев, сервисов).
8. Оценить компромиссы: увеличение количества классов против тестируемости и независимости от внешних систем.

Теоретическая часть

Проблема смешанной архитектуры

В типичных «простых» приложениях бизнес-логика часто оказывается жёстко связанной с деталями инфраструктуры:

- SQL-запросы прямо внутри методов сервисов.
- Валидация и расчёты перемешаны с парсингом JSON/HTTP-запросов.
- Отправка email или вызов внешнего API вызываются из середины бизнес-логики.

Это приводит к нарушению принципа единственной ответственности, сложности тестирования (нужны реальные БД/сеть), невозможности быстро заменить технологию (например, перейти с MySQL на MongoDB) и затруднению понимания логики.

Clean Architecture / Ports & Adapters (Hexagonal Architecture)

Основные принципы:

- **Бизнес-логика (domain)** не должна зависеть от внешних систем. Все зависимости направлены внутрь.
- **Порты (ports)** – интерфейсы, определяющие, как внешний мир взаимодействует с ядром (входящие порты – use cases) и как ядро требует от внешнего мира (исходящие порты, например, репозитории).
- **Адаптеры (adapters)** – реализации портов для конкретных технологий (JPA, REST, Kafka, CLI и т.д.).
- Ядро (entity + use cases) не знает об адаптерах.

Структура слоёв (от центра к периферии):

1. **Сущности (Entities)** – бизнес-объекты с правилами.
2. **Use cases (Interactors)** – сценарии использования, содержат бизнес-логику.
3. **Порты (Ports)** – интерфейсы для входа и выхода.
4. **Адаптеры (Adapters)** – реализации портов (контроллеры, репозитории, внешние API шлюзы, UI).

5. **Фреймворки и драйверы** – БД, веб-сервер, очереди сообщений.

Правило зависимостей: зависимости могут идти только извне внутрь. Внутренний слой не знает о внешнем.

Incoming и Outgoing порты

- **Incoming порт (Input Port)** – интерфейс use case, который вызывается адаптером (например, контроллером REST или командной строкой). Пример: CreateOrderUseCase.
- **Outgoing порт (Output Port)** – интерфейс, который use case вызывает для получения/сохранения данных или взаимодействия с внешними системами. Пример: OrderRepository, NotificationService.

Dependency Injection (ручное)

Вместо фреймворков (Spring, Guice) часто на начальном этапе можно сделать ручное внедрение зависимостей в точке входа (main или компоновщик). Это улучшает понимание архитектуры и упрощает тестирование.

Компромиссы

Аспект	**Смешанный код (без архитектуры)**	**Clean Architecture (Ports & Adapters)**
Тестируемость бизнес-логики	Низкая (требуется БД/сеть)	Высокая (use case тестируется с моками)
Скорость добавления фичи в начале	Высокая	Низкая (из-за предварительного проектирования)
Скорость замены технологии (БД)	Низкая (всюду SQL)	Высокая (только адаптер)
Количество классов	Мало	Много (интерфейсы + реализации + use cases)

Аспект	Смешанный код (без архитектуры)	Clean Architecture (Ports & Adapters)
Понятность бизнес-правил	Низкая (размазаны по фреймворкам)	Высокая (собраны в use cases)

Trade-off: Clean Architecture требует дисциплины и большего объёма кода, но она критична для средних и крупных систем, которые будут жить и развиваться несколько лет. Для прототипов или одноразовых скриптов она избыточна.

Методика выполнения работы

Шаг 1. Изучение исходного «грязного» кода

Получите индивидуальный вариант – монолитный класс/сервис, который:

- Содержит бизнес-логику и одновременно:
  - Делает прямые SQL-запросы (через JDBC или встроенную БД).
  - Работает с файловой системой.
  - Выводит сообщения в консоль (UI).
  - Вызывает внешние API (имитация).
- Определите, какие ответственности нужно разделить:
  - Сущности (Entity)
  - Use cases (сценарии)
  - Репозитории (порт + адаптер)
  - Внешние сервисы (уведомления, логирование)

Шаг 2. Выделение сущностей (Entity)

Создайте простые POJO-классы для бизнес-объектов (например, Order, Product, User). Они не должны содержать никакой логики доступа к данным или внешних вызовов.

Шаг 3. Определение портов (интерфейсов)

Создайте пакет ports:

- **Outgoing ports** (нужны use case'у):
  - OrderRepository – методы save(Order), findById(String), findAll() и т.п.
  - NotificationService – sendOrderConfirmation(Order).
- **Incoming port** (для внешних адаптеров):
  - CreateOrderUseCase – метод execute(OrderInput input) (возвращает результат).

Шаг 4. Реализация use case (Interactor)

Создайте класс CreateOrderInteractor, реализующий CreateOrderUseCase. Он зависит **только** от интерфейсов портов (например, OrderRepository, NotificationService). Содержит бизнес-логику: валидацию, расчёт суммы, сохранение через репозиторий, вызов уведомления.

Шаг 5. Реализация адаптеров

- **Адаптер репозитория:** InMemoryOrderRepository (для тестов и демонстрации) или JdbcOrderRepository (работа с реальной БД). Они реализуют интерфейс OrderRepository.
- **Адаптер уведомлений:** ConsoleNotificationService (выводит на консоль) или EmailNotificationService (имитация SMTP).
- **Адаптер входящего порта:** например, ConsoleController (читает команды из консоли и вызывает use case) или RestApiAdapter.

Шаг 6. Ручная сборка зависимостей (Dependency Injection)

Создайте класс Application или Main, где строится объектный граф:

OrderRepository repo = new InMemoryOrderRepository();

NotificationService notifier = new ConsoleNotificationService();

CreateOrderUseCase createOrder = new CreateOrderInteractor(repo, notifier);

ConsoleController controller = new ConsoleController(createOrder);

controller.start();

Шаг 7. Тестирование use case

Напишите JUnit-тест для CreateOrderInteractor, используя **Test Doubles**:

- MockOrderRepository (сохраняет заказы в список)
- SpyNotificationService (проверяет, что уведомление было вызвано)
- Проверка, что после выполнения use case репозиторий содержит новый заказ с правильными полями.

Шаг 8. Демонстрация замены адаптера

Покажите, что можно заменить InMemoryOrderRepository на JdbcOrderRepository (или другую реализацию) без изменения ни одной строки в CreateOrderInteractor и других use cases.

Шаг 9. Анализ компромиссов

В отчёте опишите, на сколько увеличилось количество классов, как изменилась тестируемость и гибкость, стоит ли такой рефакторинг для вашего конкретного варианта.

Пример выполнения задания

Исходный код (проблемный)

// OrderService – делает всё и сразу

public class OrderService {

private Connection dbConnection; *// прямое подключение к БД*

public void createOrder(String productName, int quantity, double price, String customerEmail) {

```
        // бизнес-логика
        if (productName == null || quantity <= 0) {
            throw new IllegalArgumentException("Invalid order data");
        }
        double total = quantity * price;

        // прямое сохранение в БД (смешано)
        try (PreparedStatement stmt = dbConnection.prepareStatement(
                "INSERT INTO orders (product, quantity, price, total, email) VALUES (?,?,?,?,?)")) {
            stmt.setString(1, productName);
            stmt.setInt(2, quantity);
            stmt.setDouble(3, price);
            stmt.setDouble(4, total);
            stmt.setString(5, customerEmail);
            stmt.executeUpdate();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

        // прямое уведомление
        System.out.println("Sending email to " + customerEmail + ": Order confirmed, total " + total);
    }
}
```

Рефакторинг к Clean Architecture

Шаг 2. Сущность

```
public class Order {
    private final String id;
```

```
    private final String productName;
    private final int quantity;
    private final double price;
    private final double total;
    private final String customerEmail;

    public Order(String id, String productName, int quantity, double price, double total, String customerEmail) {
        this.id = id;
        this.productName = productName;
        this.quantity = quantity;
        this.price = price;
        this.total = total;
        this.customerEmail = customerEmail;
    }
    // геттеры
}
```

Шаг 3. Порты

```
// Входящий порт (use case)
public interface CreateOrderUseCase {
    Order execute(CreateOrderCommand command);
}

// Исходящие порты
public interface OrderRepository {
    void save(Order order);
    Optional<Order> findById(String id);
}
```

```
public interface NotificationService {
    void sendOrderConfirmation(Order order);
}

// DTO для входных данных
public class CreateOrderCommand {
    public final String productName;
    public final int quantity;
    public final double price;
    public final String customerEmail;
    // конструктор
}
```

Шаг 4. Use case (интерактор)

```
public class CreateOrderInteractor implements CreateOrderUseCase {
    private final OrderRepository orderRepository;
    private final NotificationService notificationService;

    public CreateOrderInteractor(OrderRepository orderRepository,
NotificationService notificationService) {
        this.orderRepository = orderRepository;
        this.notificationService = notificationService;
    }

    @Override
    public Order execute(CreateOrderCommand command) {
        // бизнес-логика
        if (command.productName == null || command.productName.isBlank())
{
            throw new IllegalArgumentException("Product name is required");
```

```
        }
        if (command.quantity <= 0) {
            throw new IllegalArgumentException("Quantity must be positive");
        }
        if (command.price <= 0) {
            throw new IllegalArgumentException("Price must be positive");
        }
        double total = command.quantity * command.price;
        String orderId = UUID.randomUUID().toString();
        Order order = new Order(orderId, command.productName,
command.quantity, command.price, total, command.customerEmail);

        orderRepository.save(order);
        notificationService.sendOrderConfirmation(order);
        return order;
    }
}
```

Шаг 5. Адаптеры

```
// Адаптер исходящий: репозиторий in-memory
public class InMemoryOrderRepository implements OrderRepository {
    private final Map<String, Order> store = new HashMap<>();
    @Override
    public void save(Order order) { store.put(order.getId(), order); }
    @Override
    public Optional<Order> findById(String id) { return
Optional.ofNullable(store.get(id)); }
}

// Адаптер исходящий: уведомления в консоль
```

```
public class ConsoleNotificationService implements NotificationService {
    @Override
    public void sendOrderConfirmation(Order order) {
        System.out.println("Email to " + order.getCustomerEmail() + ": Order "
+ order.getId() + " total $" + order.getTotal());
    }
}

// Адаптер входящий: консольный контроллер
public class ConsoleController {
    private final CreateOrderUseCase createOrderUseCase;
    public ConsoleController(CreateOrderUseCase createOrderUseCase) {
this.createOrderUseCase = createOrderUseCase; }

    public void start() {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Product: "); String product = scanner.nextLine();
        System.out.print("Quantity: "); int qty = scanner.nextInt();
        System.out.print("Price: "); double price = scanner.nextDouble();
        scanner.nextLine();
        System.out.print("Email: "); String email = scanner.nextLine();
        CreateOrderCommand cmd = new CreateOrderCommand(product, qty,
price, email);
        Order order = createOrderUseCase.execute(cmd);
        System.out.println("Order created: " + order.getId());
    }
}
```

Шаг 6. Сборка (Dependency Injection)

```
public class Main {
```

```
        public static void main(String[] args) {
            OrderRepository repo = new InMemoryOrderRepository();  // можно заменить на JdbcOrderRepository
            NotificationService notifier = new ConsoleNotificationService();
            CreateOrderUseCase useCase = new CreateOrderInteractor(repo, notifier);
            ConsoleController controller = new ConsoleController(useCase);
            controller.start();
        }
    }
```

Шаг 7. Тест use case

```
    @Test
    void testCreateOrderSuccess() {
        OrderRepository repo = new InMemoryOrderRepository();
        NotificationService notifier = mock(NotificationService.class);
        CreateOrderUseCase useCase = new CreateOrderInteractor(repo, notifier);
        CreateOrderCommand cmd = new CreateOrderCommand("Laptop", 2, 1000.0, "test@example.com");

        Order order = useCase.execute(cmd);

        assertNotNull(order.getId());
        assertEquals(2000.0, order.getTotal());
        verify(notifier).sendOrderConfirmation(order);
        assertTrue(repo.findById(order.getId()).isPresent());
    }
```

Демонстрация замены адаптера

Замена InMemoryOrderRepository на JdbcOrderRepository не требует изменений в CreateOrderInteractor и ConsoleController. Достаточно изменить компоновку в Main.

Компромиссы в примере

- **До:** 1 класс (~40 строк). **После:** 1 сущность, 3 порта (интерфейса), 1 use case, 2 адаптера, 1 входной адаптер, точка сборки – около 8 классов (150+ строк).
- **Тестируемость:** теперь use case тестируется изолированно, без БД.
- **Гибкость:** легко заменить хранение на файл или реальную БД, уведомления – на email.
- **Начальная сложность:** больше времени на проектирование.
- **Оправданность:** если приложение имеет больше 2–3 сценариев и предполагает развитие, рефакторинг оправдан.

Индивидуальные задания

Каждый вариант содержит «грязный» сервис, который необходимо рефакторить к Clean Architecture. Требуется:

- Выделить сущности.
- Определить исходящие и входящие порты (интерфейсы).
- Перенести бизнес-логику в use case.
- Реализовать минимум два адаптера для исходящих портов (например, InMemory и другой).
- Настроить ручной DI.
- Написать тесты use case с моками.
- Продемонстрировать замену адаптера (например, с in-memory на файловое хранилище).

Вариант 1. Сервис управления задачами: методы createTask, completeTask, listTasks. Прямая работа с файлом JSON. Уведомления через System.out. Нет валидации.

Вариант 2. Сервис регистрации пользователей: register, findByEmail. Жёсткая связь с MySQL (JDBC-запросы). Email отправляется через SMTP (имитация). Валидация пароля внутри.

Вариант 3. Сервис заказа книг: placeOrder, getOrderStatus. Работа с PostgreSQL, вызов внешнего API для проверки наличия книги, отправка email через JavaMail.

Вариант 4. Сервис голосования: createPoll, vote, getResults. Хранит данные в HashMap внутри класса (нет отделения). Логирование в файл. Без тестов.

Вариант 5. Сервис перевода денег: transfer, getBalance. Прямые вызовы JDBC, логирование через log4j, внешний вызов курса валют (имитация). Нет интерфейсов.

Вариант 6. Система лояльности: addPoints, redeemPoints, getPoints. Хранение в SQLite, отправка SMS через внешний API. Бизнес-правила (минимальные баллы) перемешаны.

Вариант 7. Сервис бронирования конференц-залов: bookRoom, cancelBooking, listAvailable. Работа с MongoDB (имитация прямых запросов). Email-уведомления. Валидация времени внутри.

Вариант 8. Сервис подписки на новости: subscribe, unsubscribe, sendNewsletter. Хранение в CSV-файле. Отправка email через SMTP. Нет тестов.

Вариант 9. Сервис управления складом: addProduct, removeProduct, getStock. Прямые SQL-запросы к H2. Логирование в консоль. Валидация количества (неотрицательное).

Вариант 10. Сервис расчета зарплаты: calculateSalary, addEmployee, getEmployeeList. Хранение в Excel-файле (используя Apache POI). Вывод отчёта в PDF (имитация).

Вариант 11. Сервис сокращения ссылок: shortenUrl, getOriginalUrl, incrementClicks. Хранение в Redis (имитация). Валидация URL (регулярка). Ручное логирование.

Вариант 12. Сервис опросов/анкет: createSurvey, submitResponse, getStatistics. Прямое сохранение в текстовый файл. Отправка уведомлений в Telegram (имитация API).

Вариант 13. Сервис управления проектами: createProject, assignTask, getProjectStatus. Работа с MongoDB, вызов внешнего API для проверки доступности сотрудника. Логирование через slf4j.

Вариант 14. Сервис загрузки файлов: uploadFile, downloadFile, getFileList. Хранение на файловой системе (прямые манипуляции с java.io). Проверка лимита размера, антивирусная проверка (имитация).

Вариант 15. Сервис учёта рабочего времени: clockIn, clockOut, getHoursForEmployee. Прямая работа с PostgreSQL, отправка уведомлений через Slack API (имитация). Валидация пересечений.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО студента, вариантом.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание Clean Architecture (Ports & Adapters), правила зависимостей, отличие входных и выходных портов.
4. **Ход выполнения** (по шагам, с листингами):
   - Исходный «грязный» сервис (анализ проблем).
   - Выделенные сущности.
   - Определённые порты (интерфейсы).
   - Use case (интерактор) с бизнес-логикой.
   - Реализованные адаптеры (минимум 2 для исходящих портов).
   - Входной адаптер (контроллер).

- Ручная сборка в точке входа.

5. **Тестирование** – код тестов (JUnit) для use case с использованием Test Doubles, скриншоты выполнения.
6. **Демонстрация замены адаптера** – пример переключения с одного репозитория на другой без изменения use case.
7. **Оценка компромиссов** – таблица сравнения старого и нового кода (количество классов, строк, тестируемость, сложность).
8. **Вывод** – достигнуты ли цели, какие архитектурные навыки сформированы.
9. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Объём отчёта: 6–8 страниц.

Контрольные вопросы

1. В чём суть «правила зависимостей» в Clean Architecture? Почему зависимости должны направляться внутрь?

2. Что такое порт (port) и адаптер (adapter)? Приведите пример входного и выходного порта.

3. Какой слой отвечает за бизнес-логику в Clean Architecture: Entity, Use Case, Adapter или Framework?

4. Как внедрение портов и адаптеров улучшает тестируемость?

5. Почему в вашей реализации Use case не зависит напрямую от JDBC или файловой системы?

6. Что такое «инверсия зависимостей» (Dependency Inversion Principle) и как она реализована в данной работе?

7. Как бы вы перешли с InMemoryRepository на реальную базу данных без изменения use case?

8. Какие недостатки у подхода Clean Architecture для маленьких проектов?

9. В чём разница между входными (incoming) и выходными (outgoing) портами?

10. Как ручное Dependency Injection отличается от использования контейнера (Spring, Micronaut)? В каких случаях ручной DI предпочтительнее?

Лабораторная работа №10. Unit of Work + Repository (без ORM)

Цель работы: очистить и агрегировать код доступа к данным, реализовав паттерны Repository (представление коллекции объектов) и Unit of Work (отслеживание изменений и управление транзакциями) для работы с in-memory или файловым хранилищем без использования ORM, обеспечив транзакционную согласованность и упростив тестирование.

Задачи:

1. Проанализировать существующий код с прямыми запросами к хранилищу (например, прямые операции с коллекциями или файлами) без контроля целостности транзакций.
2. Реализовать паттерн **Repository** для каждой сущности (или обобщённый репозиторий), предоставляющий методы add, update, remove, findById, findAll и скрывающий детали хранения.
3. Реализовать паттерн **Unit of Work**, который отслеживает все изменения, сделанные с объектами в рамках одной бизнес-операции, и фиксирует их в хранилище атомарно (commit) либо откатывает (rollback).
4. Реализовать механизм отслеживания состояния сущностей (New, Clean, Dirty, Removed) с помощью карты отслеживания внутри Unit of Work.
5. Обеспечить возможность группировки операций над несколькими репозиториями в одну транзакцию через Unit of Work.
6. Написать модульные тесты, проверяющие транзакционность (успешный commit, rollback при ошибке, частичные изменения не фиксируются).
7. Оценить компромиссы: сложность реализации отслеживания изменений против гибкости транзакций, производительность против согласованности.

Теоретическая часть

Проблема работы с хранилищем в бизнес-логике

При отсутствии паттернов доступа к данным типичный код содержит:

- Прямые вызовы методов работы с файлами или коллекциями внутри сервисов.
- Отсутствие контроля за тем, что изменения нескольких объектов должны быть применены атомарно.
- Сложность тестирования (сервис привязан к конкретному хранилищу).
- Трудно поддерживать кэш и логику «грязных» объектов.

Паттерн Repository (Репозиторий)

Назначение: изолирует бизнес-логику от деталей хранения, предоставляя коллекционно-подобный интерфейс для доступа к агрегатам.

Структура:

- Repository<T, ID> – интерфейс с методами save(T entity), delete(T entity), findById(ID id), findAll() и т.п.
- InMemoryRepository<T> – реализация, хранящая данные в Map<ID, T>.
- FileRepository<T> – реализация с сериализацией в JSON/XML и т.д.

Репозиторий обычно работает с единственным типом агрегата (корнем).

Паттерн Unit of Work (Единица работы)

Назначение: поддерживает список объектов, изменённых в ходе бизнес-транзакции, и координирует запись этих изменений и разрешение проблем параллелизма.

Структура:

- UnitOfWork – интерфейс с методами registerNew(entity), registerDirty(entity), registerRemoved(entity), commit(), rollback().
- UnitOfWorkImpl – содержит ссылки на репозитории и карты отслеживания состояний (new, dirty, removed). При commit() применяет

все зарегистрированные изменения к репозиториям в правильном порядке.

- **Состояния сущностей:**
  - NEW – объект создан, но не сохранён.
  - CLEAN – объект загружен из БД и не изменялся.
  - DIRTY – объект изменён, требует сохранения.
  - REMOVED – объект помечен на удаление.

Взаимодействие Repository и Unit of Work

Обычно репозиторий делегирует регистрацию изменений Unit of Work. Например, при вызове repository.add(entity) репозиторий вызывает unitOfWork.registerNew(entity). При repository.update(entity) – registerDirty(entity). Фактическая запись в хранилище происходит только при unitOfWork.commit().

Это даёт возможность накапливать изменения и применять их пакетно, а также откатывать всю операцию при любой ошибке.

Компромиссы

Аспект	**Без паттернов (прямые операции)**	**Repository + Unit of Work**
Атомарность операций	Нет (требует ручного управления)	Да (commit/rollback)
Сложность реализации	Низкая	Средняя (требуется отслеживание состояний)
Производительность	Хорошая (каждая операция сразу)	Средняя (накопление изменений)
Тестируемость	Низкая (зависимость от хранилища)	Высокая (подмена хранилища)

Аспект	Без паттернов (прямые операции)	Repository + Unit of Work
Гибкость изменения хранилища	Низкая	Высокая (через репозиторий)

Trade-off: Unit of Work требует дополнительного кода для отслеживания состояния сущностей. Однако он даёт мощный механизм транзакций и упрощает бизнес-логику, особенно при сложных сценариях (много изменений за один запрос). Для очень простых операций (один insert) он избыточен.

Методика выполнения работы

Шаг 1. Исходный код (проблемный)

Получите индивидуальный вариант – класс сервиса, который напрямую работает с коллекцией (например, Map<String, Product>) или файлом, выполняя операции добавления, обновления и удаления без транзакционности. Нет возможности откатить изменения при ошибке.

Шаг 2. Создание сущности

Определите простую сущность (например, Product, Order, User) с несколькими полями и id (String или Long). Обеспечьте неизменяемость (immutability) или реализуйте методы equals()/hashCode() (важно для отслеживания изменений).

Шаг 3. Реализация Repository

Создайте интерфейс Repository<T, ID> и конкретную реализацию InMemoryRepository<T>, хранящую объекты в Map<ID, T>. Репозиторий не должен сам вызывать сохранение – только регистрировать изменения в Unit of Work (или, в простом варианте, сразу работать, но тогда транзакций не будет).

Для учебной цели сначала сделайте простой репозиторий без Unit of Work, затем интегрируйте его с UoW.

Шаг 4. Реализация Unit of Work

Создайте класс UnitOfWork:

- Поля: Map<Object, EntityState> states, где EntityState – enum.
- Ссылки на репозитории (или общий доступ к карте данных).
- Методы registerNew, registerClean, registerDirty, registerRemoved.
- Метод commit(): для каждого зарегистрированного объекта в зависимости от состояния выполняет действие через репозиторий (вставить, обновить, удалить), затем очищает состояния.
- Метод rollback(): просто очищает состояния (ничего не сохраняет).

Для упрощения можно сделать Unit of Work, который работает с одним репозиторием, или обобщённый, работающий со списком репозиториев.

Шаг 5. Интеграция репозитория и Unit of Work

Модифицируйте репозиторий, чтобы он принимал UnitOfWork в конструкторе (или через параметры методов). При вызове save(entity) репозиторий проверяет, существует ли entity: если нет – вызывает uow.registerNew(entity); если есть и изменён – registerDirty(entity). Метод delete вызывает registerRemoved(entity).

Фактическое сохранение происходит только в commit().

Шаг 6. Бизнес-сервис

Создайте сервисный класс, который использует репозиторий и Unit of Work. В методе бизнес-операции (например, updateProductPrice) сервис получает сущность через репозиторий, изменяет её, затем вызывает uow.commit(). При возникновении исключения вызывается rollback().

Шаг 7. Тестирование

Напишите JUnit-тесты:

- Успешная транзакция: создание/изменение нескольких сущностей, commit – данные сохранились.
- Транзакция с ошибкой: после регистрации Dirty возникает исключение, вызывается rollback – данные не изменились.
- Проверка, что повторный вызов commit не дублирует операции.
- Тест на изоляцию: две независимые единицы работы не мешают друг другу.

Шаг 8. Расширение для файлового хранилища (опционально)

Реализуйте FileRepository<T>, который сохраняет объекты в JSON-файл. Покажите, что Unit of Work работает одинаково с любой реализацией репозитория.

Шаг 9. Оценка компромиссов

Сравните «грязный» подход (прямые манипуляции с коллекцией без транзакций) и реализацию с Unit of Work + Repository.

Пример выполнения задания

Исходный код (проблемный)

```
public class ProductService {
    private Map<String, Product> storage = new HashMap<>();

    public void updatePrice(String productId, double newPrice) {
        Product product = storage.get(productId);
        if (product != null) {
            product.setPrice(newPrice);  // изменяет объект напрямую
        }
        // Нет отката, если позже что-то пойдёт не так
    }
```

```
    public void transferStock(String fromId, String toId, int quantity) {
        Product from = storage.get(fromId);
        Product to = storage.get(toId);
        from.setQuantity(from.getQuantity() - quantity);
        to.setQuantity(to.getQuantity() + quantity);
        // Если произойдёт ошибка после первого изменения – данные разъедутся
    }
}
```

Шаг 2. Сущность Product

```
public class Product {
    private final String id;
    private String name;
    private double price;
    private int quantity;

    public Product(String id, String name, double price, int quantity) {
        this.id = id;
        this.name = name;
        this.price = price;
        this.quantity = quantity;
    }
    // геттеры, сеттеры, equals/hashCode по id
}
```

Шаг 3. Интерфейс Repository

```
public interface Repository<T, ID> {
    T findById(ID id);
```

```
    List<T> findAll();
    void add(T entity);      // для новых
    void update(T entity);   // для грязных
    void remove(T entity);   // для удалённых
}
```

Шаг 4. Unit of Work (упрощённая версия)

```
public enum EntityState {
    NEW, DIRTY, REMOVED, CLEAN
}

public class UnitOfWork {
    private Map<Object, EntityState> states = new HashMap<>();
    private Repository repository;  // упрощённо – один репозиторий

    public UnitOfWork(Repository repository) {
        this.repository = repository;
    }

    public void registerNew(Object entity) {
        states.put(entity, EntityState.NEW);
    }

    public void registerDirty(Object entity) {
        if (states.get(entity) != EntityState.NEW) {
            states.put(entity, EntityState.DIRTY);
        }
    }

    public void registerRemoved(Object entity) {
```

```
        states.put(entity, EntityState.REMOVED);
    }

    public void commit() {
        for (Map.Entry<Object, EntityState> entry : states.entrySet()) {
            Object entity = entry.getKey();
            switch (entry.getValue()) {
                case NEW:
                    repository.add(entity);
                    break;
                case DIRTY:
                    repository.update(entity);
                    break;
                case REMOVED:
                    repository.remove(entity);
                    break;
                default:
                    // nothing
            }
        }
        states.clear();
    }

    public void rollback() {
        states.clear();
    }
}
```

Шаг 5. Репозиторий, интегрированный с UoW

```
public class ProductRepository implements Repository<Product, String> {
```

```
private Map<String, Product> storage = new HashMap<>();
private UnitOfWork uow;

public ProductRepository(UnitOfWork uow) { this.uow = uow; }

@Override
public Product findById(String id) {
    Product p = storage.get(id);
    if (p != null) uow.registerClean(p);
    return p;
}

@Override
public void add(Product p) {
    // не сохраняем сразу, только регистрируем
    uow.registerNew(p);
}

@Override
public void update(Product p) {
    uow.registerDirty(p);
}

@Override
public void remove(Product p) {
    uow.registerRemoved(p);
}

// Фактическое сохранение вызывается только из uow.commit()
void doAdd(Product p) { storage.put(p.getId(), p); }
```

```
    void doUpdate(Product p) { storage.put(p.getId(), p); }
    void doRemove(Product p) { storage.remove(p.getId()); }
}
```

Упрощённо: в commit() нужно вызывать методы doAdd, doUpdate, doRemove. Для этого либо репозиторий должен иметь методы для внутреннего использования, либо сделать их package-private.

Шаг 6. Сервис

```
public class ProductService {
    private ProductRepository repository;
    private UnitOfWork uow;

    public ProductService(ProductRepository repository, UnitOfWork uow) {
        this.repository = repository;
        this.uow = uow;
    }

    public void transferStock(String fromId, String toId, int qty) {
        try {
            Product from = repository.findById(fromId);
            Product to = repository.findById(toId);
            from.setQuantity(from.getQuantity() - qty);
            to.setQuantity(to.getQuantity() + qty);
            repository.update(from);
            repository.update(to);
            uow.commit();
        } catch (Exception e) {
            uow.rollback();
            throw e;
```

```
        }
    }
}
```

Шаг 7. Тесты

```
@Test
void testCommitPersistsChanges() {
    UnitOfWork uow = new UnitOfWork(repo);
    ProductRepository repo = new ProductRepository(uow);
    ProductService service = new ProductService(repo, uow);
    Product p = new Product("1", "Laptop", 1000, 10);
    repo.add(p);
    uow.commit();      // теперь p реально сохранён
    assertNotNull(repo.findById("1"));
}

@Test
void testRollbackOnError() {
    UnitOfWork uow = new UnitOfWork(repo);
    ProductRepository repo = new ProductRepository(uow);
    ProductService service = new ProductService(repo, uow);
    Product p1 = new Product("1", "A", 10, 5);
    Product p2 = new Product("2", "B", 20, 0);
    repo.add(p1);
    repo.add(p2);
    // симулируем ошибку при обновлении
    assertThrows(IllegalStateException.class, () -> {
        p2.setQuantity(-1); // недопустимо
        repo.update(p2);
        uow.commit();  // выбросит исключение валидации
```

```
    });
    // после rollback p1 не должен быть сохранён
    assertNull(repo.findById("1"));
  }
```

Шаг 8. Файловый адаптер

```
public class FileProductRepository implements Repository<Product, String>
{
    private final Path file;
    private Map<String, Product> cache = new HashMap<>();

    public FileProductRepository(Path file) { this.file = file; load(); }
    private void load() { /* чтение JSON из file */ }
    private void save() { /* запись cache в file */ }

    // Аналогичные методы, но фактическое сохранение в save() вызывается
    // либо при commit, либо сразу.
  }
```

При интеграции с Unit of Work вместо doAdd будет вызываться cache.put(), а save() вызывается после всех изменений.

Компромиссы в примере

- **До:** 1 сервисный класс (~30 строк), но без транзакций, данные могут разъезжаться.
- **После:** сущность, интерфейс репозитория, реализация in-memory, Unit of Work, сервис – около 5–6 классов (100–150 строк).
- **Гибкость:** теперь можно добавлять новые сущности и репозитории, использовать один Unit of Work на несколько агрегатов.

- **Транзакционность:** достигнута полностью, ошибка в любом месте – откат.
- **Производительность:** появляются накладные расходы на хранение состояний, но для большинства приложений не критично.

Индивидуальные задания

Каждый вариант требует реализовать репозиторий для указанной сущности, Unit of Work, и сервис, использующий их для транзакционных операций. Нужно реализовать in-memory хранилище (или файловое по желанию), продемонстрировать commit/rollback в тестах.

Вариант 1. Сущность Order (id, customerName, totalAmount, status). Репозиторий. Unit of Work. Сервис: метод updateOrderStatus и cancelOrder (удаление). Демонстрация: при отмене заказа также обнуляются резервы товаров (вторая сущность Reservation). Транзакция на оба изменения.

Вариант 2. Сущность User (id, email, passwordHash, role). Репозиторий. Unit of Work. Сервис регистрации: добавить пользователя, создать Profile (id, bio). Два репозитория в одной UoW.

Вариант 3. Сущность Task (id, title, description, status, assigneeId). Репозиторий. Unit of Work. Сервис: переназначить задачу и записать в историю TaskHistory (другая сущность). Транзакция.

Вариант 4. Сущность BankAccount (id, owner, balance). Репозиторий. Unit of Work. Сервис: перевод денег между счетами (снятие с одного, зачисление на другой). Ошибка (недостаток средств) вызывает rollback.

Вариант 5. Сущность Employee (id, name, departmentId, salary). и Department (id, name, budget). Сервис при перемещении сотрудника обновляет бюджет департаментов. Unit of Work.

Вариант 6. Сущность Product (id, name, price, stock). Репозиторий. Сервис: при оформлении заказа уменьшить stock и создать OrderItem. Ошибка (stock < 0) – rollback.

Вариант 7. Сущность Article (id, title, content, published). Репозиторий. Сервис: публикация статьи (изменение статуса и запись в PublicationLog). UoW.

Вариант 8. Сущность Ticket (id, eventName, price, isBooked). Репозиторий. Сервис: бронирование двух билетов одновременно. Если один уже занят – откат.

Вариант 9. Сущность Student (id, name, groupId) и Group (id, name, maxStudents). Сервис зачисления студента: увеличить count группы и добавить студента. При превышении лимита – rollback.

Вариант 10. Сущность Book (id, title, author, isBorrowed) и Reader (id, name, borrowedBookIds). Сервис: выдать книгу читателю (установить isBorrowed = true и добавить id в список читателя). Транзакция.

Вариант 11. Сущность Document (id, name, version, content). Репозиторий. Unit of Work. Сервис: обновление документа – инкремент версии и запись в DocumentHistory. Транзакция.

Вариант 12. Сущность Cart (id, userId) и CartItem (id, cartId, productId, quantity). Сервис: добавление товара в корзину – если товар уже есть, увеличить quantity, иначе создать новый CartItem. UoW.

Вариант 13. Сущность Meeting (id, title, time, roomId). Репозиторий. Сервис: перепланирование встречи (изменить время и освободить/занять комнату) – две операции в одной транзакции.

Вариант 14. Сущность Subscription (id, userId, planId, expiresAt). Репозиторий. Сервис: продление подписки – обновить expiresAt и создать PaymentRecord. UoW.

Вариант 15. Сущность WarehouseItem (id, productId, location, quantity). Репозиторий. Сервис: перемещение товара между складами – уменьшить quantity в одном, увеличить в другом. Транзакция.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО студента, номером варианта.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттернов Repository и Unit of Work, их роли в управлении транзакциями, диаграмма состояний сущностей.
4. **Ход выполнения**:
   - Исходный проблемный код (описание).
   - Реализация сущности.
   - Реализация интерфейса репозитория и конкретного in-memory репозитория.
   - Реализация Unit of Work (отслеживание состояний, commit, rollback).
   - Интеграция репозитория и UoW.
   - Сервисный слой с бизнес-операциями.
5. **Результаты тестирования** – код тестов (JUnit) и скриншоты, проверяющие commit, rollback, изоляцию.
6. **Оценка компромиссов** – таблица «было/стало» по количеству классов, транзакционной целостности, производительности.
7. **Вывод** – достигнуты ли цели, какие навыки приобретены (управление транзакциями, отслеживание состояния объектов).
8. **Ответы на контрольные вопросы** (письменно, развёрнуто).

Контрольные вопросы

1. Какую проблему решает паттерн Repository?

2. Какую дополнительную ценность даёт Unit of Work по сравнению с использованием одного репозитория?

3. Какие состояния объекта отслеживаются в Unit of Work? Опишите их.

4. Почему важно, чтобы Unit of Work регистрировал объекты, а не выполнял операции немедленно?

5. В каком порядке должны выполняться операции при commit (INSERT, UPDATE, DELETE)? Почему?

6. Какие могут быть проблемы при долгом существовании Unit of Work (например, на всю пользовательскую сессию)?

7. Как бы вы реализовали автоматическое отслеживание изменений (dirty checking) без явных вызовов registerDirty?

8. Как Unit of Work взаимодействует с несколькими репозиториями разных сущностей?

9. В чём отличие Unit of Work от паттерна Transaction Script?

10. Какие альтернативы Unit of Work существуют в Java (JTA, Spring @Transactional)? Каковы trade-offs между ручной реализацией и использованием готовых решений?

Лабораторная работа №11. Test Doubles: Stub, Mock, Spy, Fake

Цель работы: очистить и агрегировать подходы к тестированию модулей с внешними зависимостями путём внедрения различных видов тестовых двойников (Stub, Mock, Spy, Fake) и сравнения их применимости для разных сценариев изоляции.

Задачи:

1. Проанализировать существующий сервис с жёсткими зависимостями (например, от внешнего API, базы данных, генератора случайных чисел), который невозможно тестировать изолированно.
2. Реализовать ручные **Stub** для замены зависимостей, возвращающих предопределённые ответы без логики.
3. Реализовать ручной **Fake** (упрощённую рабочую реализацию зависимости, например, in-memory репозиторий).
4. Реализовать **Mock** с использованием библиотеки Mockito для проверки взаимодействий (вызван ли метод, с какими аргументами).
5. Реализовать **Spy** (частичный mock) для отслеживания вызовов реальных методов или частичного переопределения.
6. Написать тесты для одного и того же сервиса с использованием разных типов двойников, объяснив выбор в каждом случае.
7. Сравнить подходы: сложность реализации, гибкость, выразительность, надёжность тестов.

Теоретическая часть

Проблема тестирования модулей с зависимостями

Бизнес-логика часто зависит от внешних систем: баз данных, REST API, файловой системы, времени, генераторов случайных чисел. Прямое использование этих зависимостей в тестах приводит к:

- Медленным тестам (реальные сетевые вызовы, БД).
- Нестабильным тестам (зависимость от состояния внешней системы).

- Невозможности протестировать краевые случаи (например, ошибки сети, пустые ответы, тайм-ауты).
- Нарушению принципа изоляции: тест проверяет сразу несколько компонентов.

Паттерн Test Double (Тестовый двойник)

Test Double – общий термин для любых объектов, которые заменяют реальные зависимости в тестах. Разновидности:

Тип	Назначение	Характеристики
Dummy	Заглушка, которая нужна только для компиляции (передать параметр)	Не содержит логики, методы ничего не делают или возвращают null
Stub	Возвращает предопределённые ответы на вызовы	Содержит минимальную логику (например, всегда возвращать заданное значение)
Spy	Частично реальный объект, отслеживает вызовы и может переопределять отдельные методы	Сочетает реальную логику и регистрацию взаимодействий
Mock	Предварительно запрограммирован на ожидания; проверяет, как именно был вызван (с какими аргументами, сколько раз)	Используется для проверки взаимодействий, а не состояния
Fake	Упрощённая рабочая реализация зависимости (например, in-memory БД)	Содержит реальную логику, но не пригодна для production

Ручная реализация против библиотек (Mockito)

- **Ручная реализация**: полный контроль, понятный код, но много boilerplate для каждого теста.

- **Mockito**: динамическое создание моков/стабов, выразительный синтаксис (when().thenReturn(), verify()), но требует изучения API и может скрывать детали реализации.

Когда что использовать?

- **Stub** – когда нужно просто вернуть фиксированное значение (например, userRepository.findById(1) всегда возвращает тестового пользователя).
- **Fake** – когда нужно эмулировать поведение хранилища с несколькими объектами и операциями (например, in-memory репозиторий с поддержкой добавления/поиска).
- **Mock** – когда важно проверить, что сервис вызвал зависимость с правильными аргументами (например, отправил email или сохранил объект в БД).
- **Spy** – когда нужно частично переопределить поведение реального объекта, но при этом отследить или сохранить часть логики (редко, часто анти-паттерн).

Компромиссы

Тип двойника	**Сложность реализации**	**Проверка взаимодействий**	**Пригодность для complex логики**
Stub (ручной)	Низкая	Нет (только по состоянию)	Низкая
Fake (ручной)	Высокая (нужно реализовать логику)	Нет	Высокая (работает как БД)
Mock (Mockito)	Низкая (библиотека)	Да	Средняя
Spy (Mockito)	Низкая	Да (частично)	Низкая (опасен)

Trade-off: моки делают тесты хрупкими (рефакторинг реализации ломает тесты), но дают точную проверку вызовов. Фейки требуют поддержки

кода самого фейка, но тесты получаются более устойчивыми. Лучшая практика – использовать фейки для инфраструктурных зависимостей (репозитории) и моки/стабы для сервисов с побочными эффектами (уведомления, логирование).

Методика выполнения работы

Шаг 1. Исходный код сервиса

Получите индивидуальный вариант – класс сервиса, который зависит от внешних интерфейсов (например, UserRepository, EmailSender, RandomGenerator). Напишите бизнес-методы:

- registerUser – проверяет, что email уникален, создаёт пользователя, отправляет приветственное письмо.
- updatePassword – генерирует токен сброса, сохраняет, отправляет email.

Сервис должен быть спроектирован так, чтобы зависимости внедрялись через конструктор (Dependency Injection). В исходном коде (без тестов) зависимости могут быть реальными, но вы их замените в тестах.

Шаг 2. Тестирование с ручным Stub

Создайте класс-заглушку StubUserRepository, реализующий интерфейс UserRepository. Метод findByEmail всегда возвращает null (пользователя нет). Другой метод save ничего не делает или запоминает переданный объект в локальном поле для последующей проверки. Напишите тест регистрации с этим Stub.

Шаг 3. Тестирование с ручным Fake

Создайте InMemoryUserRepository (Fake), который хранит пользователей в Map<Long, User>, реализует все методы репозитория (сохранение, поиск по id, по email). Этот Fake можно использовать для нескольких тестов. Напишите тест регистрации, который проверяет, что пользователь действительно сохранился (через вызов repository.findByEmail).

Шаг 4. Тестирование с Mock (Mockito)

Подключите библиотеку Mockito. Создайте мок для EmailSender и UserRepository. Используйте when().thenReturn() для стаббинга (например, when(repository.findByEmail(any())).thenReturn(Optional.empty())). Проверьте, что emailSender.sendWelcomeEmail() был вызван ровно один раз с нужным email. Используйте verify().

Шаг 5. Тестирование с Spy (Mockito)

Создайте частичный мок (spy) реального объекта, например, EmailSender с реальной отправкой, но переопределите только метод, который реально отправляет письма, чтобы не спамить в тестах. Или создайте spy для репозитория, чтобы отследить вызов save. Продемонстрируйте разницу между mock и spy.

Шаг 6. Сравнение подходов

Напишите одни и те же тесты (например, успешная регистрация, ошибка дубликата email) с использованием Stub, Fake, Mock. Сравните:

- Количество кода.
- Читаемость теста.
- Надёжность (устойчивость к рефакторингу).
- Возможность проверки побочных эффектов.

Шаг 7. Оценка компромиссов

В отчёте сделайте выводы о том, какой тип двойника подходит для каких сценариев в вашем варианте.

Пример выполнения задания

Исходный код сервиса

```
public interface UserRepository {
    Optional<User> findByEmail(String email);
    void save(User user);
}

public interface EmailSender {
    void sendWelcomeEmail(String email);
    void sendPasswordResetToken(String email, String token);
}

public class UserService {
    private final UserRepository userRepository;
    private final EmailSender emailSender;

    public UserService(UserRepository userRepository, EmailSender emailSender) {
        this.userRepository = userRepository;
        this.emailSender = emailSender;
    }

    public void registerUser(String email, String name) {
        if (userRepository.findByEmail(email).isPresent()) {
            throw new IllegalStateException("Email already exists");
        }
        User user = new User(email, name);
        userRepository.save(user);
        emailSender.sendWelcomeEmail(email);
    }
```

```
    public String requestPasswordReset(String email) {
        User user = userRepository.findByEmail(email)
            .orElseThrow(() -> new IllegalArgumentException("User not found"));
        String token = "reset-token-" + System.currentTimeMillis();
        // обычно сохраняем токен, для упрощения опустим
        emailSender.sendPasswordResetToken(email, token);
        return token;
    }
}
```

Шаг 2. Ручной Stub

```
public class StubUserRepository implements UserRepository {
    private User savedUser;
    @Override
    public Optional<User> findByEmail(String email) {
        return Optional.empty(); // всегда считаем, что email не занят
    }
    @Override
    public void save(User user) {
        this.savedUser = user; // запоминаем для проверки
    }
    public User getSavedUser() { return savedUser; }
}

// Тест
@Test
void registrationWithStub_success() {
    StubUserRepository stubRepo = new StubUserRepository();
```

```
    EmailSender mockSender = mock(EmailSender.class); // для отправки используем Mock
    UserService service = new UserService(stubRepo, mockSender);
    service.registerUser("test@example.com", "John");
    assertNotNull(stubRepo.getSavedUser());
    assertEquals("test@example.com", stubRepo.getSavedUser().getEmail());
  }
```

Шаг 3. Ручной Fake

```
  public class FakeInMemoryUserRepository implements UserRepository {
    private final Map<String, User> storage = new HashMap<>();
    @Override
    public Optional<User> findByEmail(String email) {
      return Optional.ofNullable(storage.get(email));
    }
    @Override
    public void save(User user) {
      storage.put(user.getEmail(), user);
    }
  }

  @Test
  void registrationWithFake_success() {
    FakeInMemoryUserRepository fakeRepo = new FakeInMemoryUserRepository();
    EmailSender mockSender = mock(EmailSender.class);
    UserService service = new UserService(fakeRepo, mockSender);
    service.registerUser("test@example.com", "John");
    assertTrue(fakeRepo.findByEmail("test@example.com").isPresent());
  }
```

Шаг 4. Mock (Mockito)

```
@Test
void registrationWithMock_thenVerify() {
   UserRepository mockRepo = mock(UserRepository.class);
   EmailSender mockSender = mock(EmailSender.class);

when(mockRepo.findByEmail("test@example.com")).thenReturn(Optional.empty(
));

   UserService service = new UserService(mockRepo, mockSender);
   service.registerUser("test@example.com", "John");

   verify(mockRepo).save(any(User.class));
   verify(mockSender).sendWelcomeEmail("test@example.com");
   verify(mockRepo, never()).findByEmail("other@example.com");
}
```

Шаг 5. Spy

```
@Test
void requestPasswordReset_withSpy() {
   UserRepository realRepo = new FakeInMemoryUserRepository();
   realRepo.save(new User("test@example.com", "John"));
   UserRepository spyRepo = spy(realRepo);
   EmailSender mockSender = mock(EmailSender.class);

   UserService service = new UserService(spyRepo, mockSender);
   String token = service.requestPasswordReset("test@example.com");

   // проверяем, что findByEmail был вызван на репозитории
```

```
        verify(spyRepo).findByEmail("test@example.com");
        verify(mockSender).sendPasswordResetToken(eq("test@example.com"),
anyString());
        assertNotNull(token);
    }
```

Шаг 6. Сравнение

Критерий	Stub	Fake	Mock
Строк кода в тесте	~10	~8	~12
Нужно писать доп. класс	Да	Да	Нет
Проверка сохранения	Через getter	Через findByEmail	verify(save)
Устойчивость к рефакторингу	Средняя (если метод сохранения меняется)	Высокая (тест проверяет состояние)	Низкая (привязан к имени метода)
Скорость выполнения	Очень быстрая	Быстрая	Быстрая

Индивидуальные задания

Каждый вариант содержит интерфейсы зависимостей и сервис с бизнес-логикой. Требуется:

- Написать минимум 2 теста с ручным Stub, 2 теста с Fake, 2 теста с Mock (Mockito), 1 тест со Spy.
- Объяснить, почему в каждом конкретном случае выбран именно этот тип двойника.
- Сравнить подходы в отчёте.

Вариант 1. Сервис заказов: зависит от OrderRepository, InventoryService, PaymentGateway. Метод placeOrder проверяет наличие товара, резервирует, списывает оплату, сохраняет заказ. При ошибке оплаты – откат резерва.

Вариант 2. Сервис рекомендаций: зависит от UserHistoryRepository, ProductRecommender (внешний API), CacheService. Метод getRecommendations сначала проверяет кэш, если нет – вызывает API, сохраняет в кэш.

Вариант 3. Сервис уведомлений: зависит от UserPreferencesRepository, SmsSender, EmailSender, PushSender. Метод notifyUser выбирает канал на основе предпочтений и отправляет.

Вариант 4. Сервис аутентификации: зависит от UserRepository, PasswordEncoder, TokenGenerator, LoginAttemptRepository. Метод login проверяет пароль, учитывает количество неудачных попыток, генерирует токен.

Вариант 5. Сервис аналитики: зависит от EventRepository, StatisticsCalculator, ReportGenerator. Метод generateDailyReport собирает события за день, передаёт в калькулятор, генерирует отчёт и отправляет на email.

Вариант 6. Сервис перевода денег: зависит от AccountRepository, ExchangeRateProvider, NotificationService. Метод transfer конвертирует валюту, списывает/зачисляет, уведомляет.

Вариант 7. Сервис управления инвентарём: зависит от ProductRepository, WarehouseService, SupplierClient. Метод reorderIfNeeded проверяет остатки и при необходимости создаёт заказ поставщику.

Вариант 8. Сервис анкетирования: зависит от SurveyRepository, ResponseParser, ScoreCalculator. Метод submitResponse сохраняет ответ, вычисляет баллы и обновляет рейтинг пользователя.

Вариант 9. Сервис подписки на новости: зависит от SubscriptionRepository, NewsletterService, EmailValidator, CrmClient. Метод subscribe валидирует email, добавляет в рассылку, синхронизирует с CRM.

Вариант 10. Сервис загрузки файлов: зависит от FileStorage, VirusScanner, MetadataExtractor. Метод uploadFile сканирует на вирусы, извлекает метаданные, сохраняет файл.

Вариант 11. Сервис поиска: зависит от SearchIndex, QueryParser, ResultRanker, UserActivityLogger. Метод search парсит запрос, ищет, ранжирует, логирует.

Вариант 12. Сервис резюме (биржа труда): зависит от ResumeRepository, CandidateMatcher, EmployerNotifier. Метод postResume сохраняет резюме, ищет подходящие вакансии, уведомляет работодателей.

Вариант 13. Сервис календаря: зависит от EventRepository, CalendarApiClient, ReminderService. Метод createEvent проверяет конфликты, создаёт событие, ставит напоминания.

Вариант 14. Сервис сокращения ссылок: зависит от LinkRepository, UrlValidator, ClickAnalytics. Метод shorten валидирует URL, создаёт короткий код, сохраняет, возвращает.

Вариант 15. Сервис модерации контента: зависит от ContentRepository, ToxicWordFilter, ModerationQueue, AdminNotifier. Метод submitContent проверяет токсичность, при превышении порога отправляет в очередь модерации.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО студента, вариантом.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание видов Test Doubles, их назначение, сравнительная таблица, примеры использования.
4. **Ход выполнения**:
   - Исходный код сервиса и интерфейсов зависимостей.
   - Реализация ручного Stub (листинг + тест).

- Реализация ручного Fake (листинг + тест).
- Тесты с Mock (Mockito) – листинг.
- Тест со Spy (Mockito) – листинг.

5. **Сравнительный анализ** – таблица с оценкой каждого подхода для вашего варианта по метрикам (сложность, выразительность, устойчивость, удобство отладки).
6. **Вывод** – достигнута ли цель, какие навыки тестирования приобретены.
7. **Ответы на контрольные вопросы** (развёрнуто).

Контрольные вопросы

1. В чём основное различие между Stub и Mock?

2. Когда следует предпочесть Fake вместо Mock? Приведите пример из своей работы.

3. Почему Spy считается анти-паттерном в большинстве случаев? Когда он всё же полезен?

4. Как ручной Stub отличается от ручного Fake?

5. Как Mockito позволяет проверить, что метод был вызван определённое количество раз?

6. Какие недостатки у использования Mock для тестирования поведения (state verification vs behavior verification)?

7. Что такое «хрупкие тесты» и как тип двойника влияет на их появление?

8. Можно ли комбинировать Fake и Mock в одном тесте? Приведите пример.

9. Почему Dummy используется редко и обычно встроен в другие двойники?

10. Как паттерн Dependency Injection связан с использованием Test Doubles? Почему без него трудно тестировать?

Лабораторная работа №12. Object Mother и Test Data Builder

Цель работы: очистить и агрегировать код тестовых наборов данных путём внедрения паттернов Object Mother (стандартные предустановки) и Test Data Builder (кастомизируемое создание объектов) для сложных сущностей с множеством полей, обеспечив читаемость, переиспользование и упрощение сопровождения тестовых данных.

Задачи:

1. Проанализировать существующие тесты, в которых создание объектов (сущностей с 10+ полями) дублируется или приводит к непонятным и длинным конструкторам с большим количеством параметров.
2. Реализовать паттерн **Object Mother** – статический фабричный класс, предоставляющий методы с предопределёнными вариациями объектов (стандартный пользователь, пользователь с пустым email, заказ с большим количеством товаров и т.п.).
3. Реализовать паттерн **Test Data Builder** (через Fluent API) для гибкой кастомизации отдельных полей при создании тестового объекта, с поддержкой значений по умолчанию.
4. Обеспечить комбинирование Object Mother (базовый объект) и Builder (переопределение отдельных полей).
5. Переписать существующие тесты с использованием созданных фабрик, чтобы они стали короче и читаемее.
6. Написать модульные тесты для самих строителей и материнских объектов (проверка, что они создают ожидаемые объекты).
7. Оценить компромиссы: уменьшение дублирования против увеличения вспомогательного кода, гибкость против сложности.

Теоретическая часть

Проблема создания тестовых данных

В модульных тестах часто требуется создавать сложные объекты с большим количеством полей. Типичные проблемы:

- **Длинные конструкторы** – трудно читать, что за параметр что означает.
- **Дублирование** – один и тот же объект создаётся в десятках тестов.
- **Хрупкость** – при добавлении нового обязательного поля все тесты ломаются.
- **Нечитаемость** – тест загромождён деталями создания объекта, а не бизнес-логикой.

Паттерн Object Mother

Назначение: централизованное создание тестовых объектов с предопределёнными, часто используемыми конфигурациями.

Реализация:

- Класс с статическими методами, возвращающими готовые объекты.
- Имена методов отражают назначение: typicalUser(), userWithNoEmail(), orderWithMultipleItems().
- Может также включать методы для создания коллекций.

Недостатки:

- При большом количестве вариаций метод раздувается.
- Если нужно изменить одно поле, приходится создавать новый метод или использовать комбинацию с Builder.

Паттерн Test Data Builder

Назначение: предоставляет гибкий, кастомизируемый способ создания объектов для тестов, используя Fluent API и значения по умолчанию.

Реализация:

- Внутренний (или отдельный) класс Builder с полями-копиями полей сущности.

- Конструктор Builder задаёт разумные значения по умолчанию.
- Методы-сеттеры возвращают this (Fluent).
- Метод build() создаёт объект (вызывая конструктор сущности или через сеттеры).

Преимущества:

- Тесты становятся самодокументированными: userBuilder().withEmail("x@x.com").build().
- Не нужно знать все поля, только те, что важны для конкретного теста.
- Легко добавить новое поле (добавить значение по умолчанию в Builder, старые тесты не ломаются).

Object Mother vs Builder

Аспект	Object Mother	Test Data Builder
Простота использования	Очень простая (вызов одного метода)	Требуется цепочка вызовов
Гибкость	Низкая (только предустановленные варианты)	Высокая (любое поле можно переопределить)
Поддержка	Средняя (при росте числа методов)	Хорошая (один строитель на сущность)
Переиспользование	Высокое (одинаковый объект в 10 тестах)	Среднее (приходится повторять кастомизацию)

Лучшая практика: комбинировать оба подхода – Object Mother предоставляет стандартные объекты, а Builder позволяет их модифицировать. Например:

User typicalUser = UserObjectMother.typicalUser();

User modified = UserBuilder.from(typicalUser).withEmail("new@example.com").build();

Компромиссы

- **Плюсы:** тесты становятся короче, устойчивее к изменениям в конструкторах, проще читаются.
- **Минусы:** увеличивается количество вспомогательных классов/методов (builder + mother). Может создавать иллюзию, что тестовые объекты «реальные», а они содержат много захардкоженных значений.

Trade-off: для простых объектов (2–3 поля) не нужны ни Mother, ни Builder. Для сущностей с 6+ полями, используемых во многих тестах, этот подход окупается.

Методика выполнения работы

Шаг 1. Исходные тесты (проблемные)

Получите индивидуальный вариант – сложную сущность с 8–12 полями и набор тестов, где её создание повторяется с небольшими вариациями. Определите:

- Какие поля являются обязательными, а какие опциональными?
- Какие часто используемые комбинации значений встречаются?
- Как часто добавляются новые поля?

Шаг 2. Реализация Builder для сущности

Создайте класс XxxBuilder (где Xxx – имя сущности):

- Хранит те же поля, что и сущность, с дефолтными значениями (разумными для тестов).
- Методы withField1(value), withField2(value) возвращают this.
- Метод build() создаёт экземпляр сущности (через конструктор или через рефлексию/сеттеры, если их нет).
- (Опционально) метод clone() для копирования существующего билдера.

Шаг 3. Реализация Object Mother

Создайте класс XxxObjectMother со статическими методами:

- defaultXxx() – возвращает объект с типичными значениями для большинства тестов.
- xxxWithoutEmail(), xxxWithEmptyCart(), xxxWithLargeTotal() и т.п. – специфические вариации.

В реализации Object Mother часто используют Builder под капотом:

```
public static User defaultUser() {
   return UserBuilder.aUser().build();
}
public static User userWithNoEmail() {
   return UserBuilder.aUser().withEmail(null).build();
}
```

Шаг 4. Переписывание тестов

Замените все непосредственные вызовы конструкторов или сеттеров на использование:

- Object Mother для стандартных случаев.
- Builder для случаев, когда нужно сконфигурировать специфические поля (не более 2–3 переопределений).

Шаг 5. Тестирование билдера и матери

Напишите юнит-тесты для Builder и Object Mother, чтобы убедиться, что они создают объекты с корректными значениями по умолчанию.

Шаг 6. Анализ эффективности

Сравните старые и новые тесты по количеству строк, читаемости, времени поддержки.

Пример выполнения задания

Исходная сущность (проблемная)

```
public class Order {
```

```
    private String id;
    private String customerName;
    private String customerEmail;
    private List<LineItem> items;
    private LocalDate orderDate;
    private double discountPercent;
    private String promoCode;
    private String shippingAddress;
    private String billingAddress;
    private boolean giftWrap;
    private String deliveryInstructions;
    // конструктор с 11 параметрами!!! (не показываем полностью)
}
```

Исходный тест (плохой)

```
@Test
void testDiscountApplication() {
    List<LineItem> items = Arrays.asList(new LineItem("laptop", 1000, 1));
    Order order = new Order("ORD-123", "John Doe", "john@example.com",
items,
        LocalDate.now(), 10.0, "SAVE10", "123 Main St", "123 Main St", false,
"");
    // ... логика теста
}
```

Шаг 2. Реализация Builder

```
public class OrderBuilder {
    private String id = "TEST-ID-001";
    private String customerName = "Test Customer";
    private String customerEmail = "test@example.com";
```

```
        private List<LineItem> items = new ArrayList<>();
        private LocalDate orderDate = LocalDate.now();
        private double discountPercent = 0.0;
        private String promoCode = null;
        private String shippingAddress = "123 Test St";
        private String billingAddress = "123 Test St";
        private boolean giftWrap = false;
        private String deliveryInstructions = "";

        public OrderBuilder withId(String id) { this.id = id; return this; }
        public OrderBuilder withCustomerName(String name) { this.customerName = name; return this; }
        public OrderBuilder withCustomerEmail(String email) { this.customerEmail = email; return this; }
        public OrderBuilder withItems(List<LineItem> items) { this.items = items; return this; }
        public OrderBuilder withDiscountPercent(double discount) { this.discountPercent = discount; return this; }
        public OrderBuilder withPromoCode(String code) { this.promoCode = code; return this; }
        // ... остальные with-методы

        public Order build() {
            return new Order(id, customerName, customerEmail, items, orderDate,
                discountPercent, promoCode, shippingAddress, billingAddress, giftWrap, deliveryInstructions);
        }

        public static OrderBuilder anOrder() { return new OrderBuilder(); }
    }
```

Шаг 3. Object Mother

```
public class OrderObjectMother {
    public static Order standardOrder() {
        return OrderBuilder.anOrder()
            .withId("ORD-STD")
            .withCustomerName("Alice")
            .withCustomerEmail("alice@example.com")
            .withItems(List.of(new LineItem("book", 20, 2)))
            .build();
    }

    public static Order orderWithPromoCode(String promoCode) {
        return OrderBuilder.anOrder()
            .withPromoCode(promoCode)
            .withDiscountPercent(15.0)
            .build();
    }

    public static Order orderWithGiftWrap() {
        return OrderBuilder.anOrder()
            .withGiftWrap(true)
            .withDeliveryInstructions("Leave at the door")
            .build();
    }

    public static Order emptyOrder() {
        return OrderBuilder.anOrder()
            .withItems(Collections.emptyList())
            .build();
```

```
    }
}
```

Переписанный тест

```
@Test
void testDiscountApplication() {
    Order order = OrderObjectMother.standardOrder();
    // тест проверяет применение скидки к стандартному заказу
}

@Test
void testPromoCode() {
    Order order = OrderObjectMother.orderWithPromoCode("SUMMER20");
    // проверяем, что промокод применён
}

@Test
void testGiftWrapShipping() {
    Order order = OrderObjectMother.orderWithGiftWrap();
    // проверяем логику упаковки
}

// Кастомизация через Builder
@Test
void testHighValueOrder() {
    Order order = OrderBuilder.anOrder()
        .withItems(List.of(new LineItem("Laptop", 1500, 1)))
        .withDiscountPercent(0.0)
        .build();
    // тест для дорогого заказа без скидки
```

```
}
```

Тестирование строителя

```
@Test
void testBuilderDefaults() {
    Order order = OrderBuilder.anOrder().build();
    assertEquals("TEST-ID-001", order.getId());
    assertEquals("test@example.com", order.getCustomerEmail());
    assertTrue(order.getItems().isEmpty());
}
```

Компромиссы

- **До:** в каждом тесте 11 параметров конструктора, сложно читать, много дублирования.
- **После:** 2 вспомогательных класса (~120 строк). В тестах – 1–2 строки для создания объекта.
- **Поддержка:** добавление нового поля в Order требует добавления поля в Builder и установки дефолта. Старые тесты не ломаются (дефолт значения). Это лучше, чем правка 50 тестов.

Индивидуальные задания

Каждый вариант содержит сложную сущность с 8–12 полями и набор из 3–4 тестов, где эта сущность создаётся напрямую. Требуется:

- Реализовать Test Data Builder для сущности.
- Реализовать Object Mother минимум с 3 предустановками.
- Переписать тесты, используя Mother и Builder.
- Написать тесты для Builder.

Вариант 1. Сущность Invoice (поля: id, date, companyName, taxId, items (List), subtotal, taxRate, total, dueDate, status, notes, paid). Более 10 полей. Тесты: создание счета, расчёт налога, отметка об оплате.

Вариант 2. Сущность CustomerProfile (firstName, lastName, email, phone, address, city, zip, loyaltyPoints, birthDate, newsletterOptIn, preferences(Map)). Тесты: регистрация, обновление профиля, начисление бонусов.

Вариант 3. Сущность Product (id, name, description, price, category, tags (List), stockQuantity, weight, dimensions, isActive, createdDate, manufacturer). Тесты: создание продукта, обновление цены, проверка доступности.

Вариант 4. Сущность FlightBooking (bookingId, passengerName, passportNumber, flightNumber, departureDate, returnDate, seatClass, mealPreference, baggageWeight, specialAssistance, price, status). Тесты: бронирование, изменение класса, отмена.

Вариант 5. Сущность InsurancePolicy (policyNumber, customerId, type (enum), startDate, endDate, premiumAmount, coverageAmount, beneficiaries (List), deductible, autoRenew, termsAccepted). Тесты: создание полиса, расчёт премии, пролонгация.

Вариант 6. Сущность Meeting (title, organizer, startTime, endTime, participants (List), location, isRecurring, recurrenceRule, meetingRoom, videoConferenceLink, agenda, attachments (List)). Тесты: создание встречи, добавление участников, отправка приглашений.

Вариант 7. Сущность BlogPost (id, title, slug, content, author, tags (List), publishedDate, status (draft/published), viewCount, commentCount, allowComments, seoDescription). Тесты: создание черновика, публикация, увеличение счётчика просмотров.

Вариант 8. Сущность BankTransaction (transactionId, accountNumber, type (credit/debit), amount, currency, counterparty, description, transactionDate, valueDate, reference, fee, status). Тесты: проведение транзакции, расчёт комиссии, ошибочная отмена.

Вариант 9. Сущность HotelReservation (reservationId, guestName, guestEmail, roomType, checkIn, checkOut, numberOfGuests, breakfastIncluded, parkingIncluded, totalPrice, specialRequests, loyaltyNumber). Тесты: бронирование, изменение дат, отмена.

Вариант 10. Сущность ExamResult (studentId, examName, examDate, score, maxScore, percentage, grade, feedback, reattemptAllowed, proctorName, cheatingDetected, remarks). Тесты: публикация результата, расчёт процента, разрешение пересдачи.

Вариант 11. Сущность Employee (empId, fullName, department, position, salary, hireDate, bonusPercent, supervisorId, email, phone, address, active). Тесты: повышение зарплаты, перевод в другой отдел, увольнение.

Вариант 12. Сущность Course (courseCode, title, description, credits, instructor, schedule (List), capacity, enrolledStudents, prerequisites (List), syllabus, isActive, gradingPolicy). Тесты: регистрация студента, закрытие курса, обновление расписания.

Вариант 13. Сущность Shipment (trackingNumber, origin, destination, weight, dimensions, carrier, serviceType, estimatedDelivery, actualDelivery, status, insurance, recipientName). Тесты: создание отправления, обновление статуса, расчёт стоимости.

Вариант 14. Сущность Subscription (id, userId, planId, startDate, nextBillingDate, cancellationDate, autoRenew, paymentMethod, discountApplied, billingCycle, status, history (List)). Тесты: подписка, отмена, продление.

Вариант 15. Сущность Task (id, title, description, assigneeId, reporterId, priority (enum), status (enum), dueDate, estimatedHours, loggedHours, tags (List), attachments (List), parentTaskId). Тесты: создание задачи, назначение исполнителя, изменение статуса.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттернов Object Mother и Test Data Builder, их сравнение, примеры использования.
4. **Ход выполнения**:
   - Исходная сущность и проблемные тесты (фрагменты).
   - Реализация Builder (листинг).
   - Реализация Object Mother (листинг).
   - Переписанные тесты (сравнение до/после).
   - Тесты на Builder/Mother.
5. **Оценка компромиссов** – таблица сравнения подходов, анализ выигрыша в строках кода и читаемости.
6. **Вывод** – достигнуты ли цели, приобретённые навыки.
7. **Ответы на контрольные вопросы** (развёрнуто).

Контрольные вопросы

1. В чём основное различие между Object Mother и Test Data Builder?

2. В каких случаях использование Object Mother предпочтительнее Builder?

3. Как Builder решает проблему добавления нового обязательного поля в сущность?

4. Почему значения по умолчанию в Builder должны быть разумными для тестов, а не для production?

5. Как можно переиспользовать Builder для создания похожих объектов с небольшими отличиями?

6. В чём недостаток Object Mother при большом количестве вариаций объекта?

7. Как обеспечить неизменяемость (immutability) создаваемых объектов при использовании Builder?

8. Как паттерны Object Mother и Builder связаны с DRY (Don't Repeat Yourself) в тестах?

9. Какие альтернативы существуют для создания тестовых данных в Java (например, Instancio, JavaFaker)?

10. Почему не стоит использовать Builder из production-кода для тестов? (или наоборот, можно?)

Лабораторная работа №13. Domain Events и локальная Event Bus (без внешних брокеров)

Цель работы: очистить и агрегировать код реагирования на изменения доменных объектов путём внедрения слабо связанной событийной архитектуры через локальную шину событий (Event Bus) в памяти, обеспечив публикацию доменных событий (например, OrderPlaced, OrderShipped) и их асинхронную или синхронную обработку подписчиками без использования внешних брокеров сообщений.

Задачи:

1. Проанализировать существующий код, в котором побочные эффекты (логирование, отправка email, обновление статистики) жёстко зашиты внутри бизнес-логики сервиса.
2. Реализовать иерархию **доменных событий** (Domain Events) – простые иммутабельные классы-сущности, представляющие факт произошедшего в домене (например, OrderPlacedEvent, OrderStatusChangedEvent).
3. Реализовать локальную **Event Bus** (шину событий) внутри одного процесса, позволяющую подписываться на события определённого типа и публиковать их.
4. Обеспечить возможность синхронной и (опционально) асинхронной обработки событий (например, через ExecutorService).
5. Реорганизовать бизнес-сервис: после выполнения основных действий (изменения состояния) он публикует соответствующие доменное событие через Event Bus, не вызывая напрямую логирование, уведомления и т.д.
6. Создать несколько подписчиков (например, EmailSubscriber, StatisticsSubscriber, AuditLogSubscriber), которые реализуют побочные эффекты.

7. Написать интеграционные тесты, проверяющие, что при вызове бизнес-операции корректные события публикуются и все зарегистрированные подписчики их получают.
8. Оценить компромиссы: уменьшение связности против усложнения потока управления, синхронная vs асинхронная обработка, трассировка событий.

Теоретическая часть

Проблема смешанных побочных эффектов

В типичных приложениях бизнес-логика часто «обросла» вспомогательными действиями: после сохранения заказа отправляется письмо, обновляется статистика, пишется лог, вызывается внешний API. Эти действия:

- Нарушают принцип единственной ответственности (сервис знает о почтовике, логгере, аналитике).
- Делают тестирование сложным (каждый тест затрагивает множество инфраструктурных деталей).
- Затрудняют добавление нового побочного эффекта (нужно менять существующие методы сервиса).

Паттерн Domain Event (Доменное событие)

Назначение: инкапсулировать информацию о событии, произошедшем в домене, которое может быть интересно другим компонентам системы.

Характеристики:

- Неизменяемый объект (immutable) – только геттеры и конструктор.
- Содержит данные о том, что произошло (например, идентификатор заказа, время, старый и новый статус).
- Не содержит логики – только данные.
- Название отражает факт в прошлом (OrderPlaced, PaymentReceived).

Локальная Event Bus (шина событий в памяти)

Назначение: механизм подписки и публикации событий, работающий внутри одного процесса (JVM). Позволяет полностью развязать издателя и подписчика.

Структура:

- EventBus – интерфейс с методами publish(event) и subscribe(Class<?> eventType, EventHandler handler).
- простая реализация – хранит Map<Class<?>, List<EventHandler>>.
- При публикации события шина находит всех подписчиков на этот тип (и возможно на суперклассы, в простом варианте – точное совпадение) и вызывает их обработчики.

Варианты исполнения:

- **Синхронный** – подписчики вызываются в том же потоке, что и публикация. Просто, но можно замедлить основной поток.
- **Асинхронный** – подписчики выполняются в пуле потоков (или отдельно для каждого). Усложняет тестирование, требует аккуратности с транзакциями.

Связь с архитектурой

- **Event Sourcing** хранит все события как источник истины (более крупный паттерн).
- **CQRS** часто разделяет команды и запросы, события используются для синхронизации моделей.
- Данная лабораторная реализует только **локальное уведомление** о событиях в рамках одного процесса.

Компромиссы

Аспект	Прямые вызовы из сервиса	Domain Events + Event Bus
Связность	Высокая (сервис знает всё)	Низкая (сервис только публикует событие)
Тестируемость	Низкая (много моков)	Высокая (можно проверить публикацию события)
Добавление нового побочного эффекта	Менять сервис	Добавить подписчика
Отладка потока	Линейная	Распределённая (трудно увидеть все реакции)
Производительность	Высокая (один вызов)	Синхронная – такая же; асинхронная – накладные расходы на потоки

Trade-off: для небольших систем прямая связь может быть проще. Для развивающейся системы с большим количеством сквозных функций (уведомления, аудит, аналитика) Event Bus даёт гибкость.

Методика выполнения работы

Шаг 1. Исходный код (проблемный)

Получите индивидуальный вариант – сервис (например, OrderService), который после изменения доменного объекта вызывает несколько внешних сервисов: emailSender.sendOrderConfirmation(...), analytics.recordOrder(...), audit.log(...). В бизнес-методах эти вызовы жёстко закодированы.

Шаг 2. Создание доменных событий

Определите классы событий, соответствующие ключевым действиям в вашем варианте. Например:

- OrderPlacedEvent (содержит orderId, customerEmail, total).

- OrderStatusChangedEvent (содержит orderId, oldStatus, newStatus).

  Сделайте их иммутабельными (final поля, конструктор, геттеры).

Шаг 3. Реализация Event Bus (синхронной)

Создайте класс SimpleEventBus:

- приватное поле Map<Class<?>, List<EventHandler>> subscribers.
- метод subscribe(Class<?> eventType, EventHandler handler) добавляет в список.
- метод publish(Object event) находит список для event.getClass() и вызывает у каждого обработчика метод handle(event) (синхронно, в текущем потоке).

Определите @FunctionalInterface EventHandler { void handle(Object event); }.

Шаг 4. Изменение бизнес-сервиса

- Добавьте в сервис поле EventBus eventBus.
- После выполнения операции (сохранения заказа, изменения статуса и т.п.) создайте объект события и вызовите eventBus.publish(event).
- Удалите прямые вызовы emailSender, analytics и т.д. из сервиса.

Шаг 5. Реализация подписчиков

Создайте классы:

- OrderEmailSubscriber – при событии OrderPlacedEvent отправляет email (имитация).
- OrderStatisticsSubscriber – при событии обновляет счётчик заказов.
- AuditLogSubscriber – пишет в лог (например, в System.out).

Подпишите их в точке старта приложения (или в тесте) на соответствующие события:

eventBus.subscribe(OrderPlacedEvent.class, new OrderEmailSubscriber());

eventBus.subscribe(OrderPlacedEvent.class, new OrderStatisticsSubscriber());

Шаг 6. Опционально: асинхронная Event Bus

Создайте вторую реализацию AsyncEventBus, которая использует ExecutorService. При публикации задачи на обработку каждого подписчика отправляются в пул потоков. Сравните поведение.

Шаг 7. Тестирование

Напишите интеграционные тесты:

- Тест, проверяющий, что при вызове placeOrder публикуется событие OrderPlacedEvent.
- Тест с заглушкой подписчика (mock), который проверяет факт получения события.
- Тест (для асинхронной версии) с ожиданием завершения потоков.

Шаг 8. Оценка компромиссов

В отчёте сравните старый и новый код: количество изменённых классов, сложность понимания потока, гибкость добавления новых подписчиков.

Пример выполнения задания

Исходный код (проблемный)

```
public class OrderService {
    private EmailSender emailSender;
    private StatisticsService statistics;
    private AuditLogger audit;

    public void placeOrder(OrderData data) {
        Order order = new Order(data);
        orderRepository.save(order);
```

```
        // Побочные эффекты прямо здесь
        emailSender.sendOrderConfirmation(order.getCustomerEmail(),
order.getId());
        statistics.incrementOrderCount();
        audit.log("Order placed: " + order.getId());
    }
}
```

Шаг 2. События

```
public class OrderPlacedEvent {
    private final String orderId;
    private final String customerEmail;
    private final LocalDateTime occurredAt;

    public OrderPlacedEvent(String orderId, String customerEmail) {
        this.orderId = orderId;
        this.customerEmail = customerEmail;
        this.occurredAt = LocalDateTime.now();
    }
    // геттеры
}
```

Шаг 3. Event Bus

```
@FunctionalInterface
public interface EventHandler<T> {
    void handle(T event);
}

public class SimpleEventBus {
```

```
    private final Map<Class<?>, List<EventHandler<?>>> subscribers = new HashMap<>();

    public <T> void subscribe(Class<T> eventType, EventHandler<T> handler) {
        subscribers.computeIfAbsent(eventType, k -> new ArrayList<>()).add(handler);
    }

    @SuppressWarnings("unchecked")
    public void publish(Object event) {
        List<EventHandler<?>> handlers = subscribers.get(event.getClass());
        if (handlers != null) {
            for (EventHandler<?> handler : handlers) {
                ((EventHandler<Object>) handler).handle(event);
            }
        }
    }
}
```

Шаг 4. Модифицированный сервис

```
public class OrderService {
    private final OrderRepository repository;
    private final EventBus eventBus;

    public OrderService(OrderRepository repository, EventBus eventBus) {
        this.repository = repository;
        this.eventBus = eventBus;
    }
```

```
    public void placeOrder(OrderData data) {
        Order order = new Order(data);
        repository.save(order);
        eventBus.publish(new OrderPlacedEvent(order.getId(),
order.getCustomerEmail()));
    }
}
```

Шаг 5. Подписчики

```
public class EmailSubscriber implements EventHandler<OrderPlacedEvent>
{
    private final EmailSender emailSender;
    @Override
    public void handle(OrderPlacedEvent event) {
        emailSender.sendOrderConfirmation(event.getCustomerEmail(),
event.getOrderId());
    }
}

public class StatisticsSubscriber implements
EventHandler<OrderPlacedEvent> {
    private final StatisticsService stats;
    @Override
    public void handle(OrderPlacedEvent event) {
        stats.incrementOrderCount();
    }
}
```

Шаг 6. Настройка приложения (Main / тест)

```
EventBus eventBus = new SimpleEventBus();
```

```
eventBus.subscribe(OrderPlacedEvent.class, new EmailSubscriber(emailSender));
eventBus.subscribe(OrderPlacedEvent.class, new StatisticsSubscriber(statistics));

OrderService service = new OrderService(repository, eventBus);
service.placeOrder(data);  // при выполнении события дойдут до подписчиков
```

Шаг 7. Тест (с использованием mock)

```
@Test
void testEventPublished() {
    EventBus eventBus = new SimpleEventBus();
    EventHandler<OrderPlacedEvent> mockHandler = mock(EventHandler.class);
    eventBus.subscribe(OrderPlacedEvent.class, mockHandler);

    OrderService service = new OrderService(mockRepository, eventBus);
    service.placeOrder(someData);

    verify(mockHandler, times(1)).handle(any(OrderPlacedEvent.class));
}
```

Компромиссы

- **До:** один класс OrderService зависел от трёх внешних сервисов (email, статистика, аудит).
- **После:** OrderService знает только об EventBus. Email, статистика, аудит вынесены в независимые подписчики.

- Добавление нового побочного эффекта (например, SMS-уведомление) теперь просто: создаём подписчик и регистрируем его в EventBus – OrderService не меняется.
- Усложнение: трассировка цепочки «что произошло после размещения заказа» труднее: нужно знать, какие подписчики зарегистрированы на событие.

Индивидуальные задания

Каждый вариант – доменный агрегат (сущность) и набор побочных эффектов. Требуется:

- Выделить доменные события (минимум 2 разных события).
- Реализовать Event Bus (синхронный).
- Переписать сервис, чтобы он публиковал события вместо прямых вызовов.
- Реализовать подписчиков для всех исходных побочных эффектов.
- Написать интеграционные тесты.

Вариант 1. Заказ (Order): при создании (placeOrder) → уведомление покупателя, запись в лог аудита. При отмене (cancelOrder) → уведомление, возврат товара на склад (имитация). События: OrderPlaced, OrderCancelled.

Вариант 2. Пользователь (User): при регистрации → отправка приветственного письма, начисление 100 бонусов. При удалении аккаунта → очистка персональных данных (лог), отписка от рассылки. События: UserRegistered, UserDeleted.

Вариант 3. Товар на складе (InventoryItem): при пополнении запаса → обновление аналитики, проверка порога минимального остатка (если достигнут → событие LowStock с отдельным подписчиком). События: StockIncreased, LowStockDetected.

Вариант 4. Бронирование отеля (Reservation): при создании → отправка подтверждения, блокировка номера. При отмене → разблокировка номера, отправка уведомления. События: ReservationCreated, ReservationCancelled.

Вариант 5. Платеж (Payment): при успешном списании → запись в бухгалтерию, отправка чека. При возврате → запись возврата, уведомление пользователя. События: PaymentSuccess, PaymentRefunded.

Вариант 6. Документ в системе документооборота (Document): при утверждении → смена статуса, рассылка уведомления заинтересованным сторонам, запись в историю. События: DocumentApproved, DocumentRejected.

Вариант 7. Проектная задача (ProjectTask): при назначении исполнителя → отправка уведомления исполнителю, обновление доски проекта. При завершении задачи → уведомление постановщика, перерасчёт прогресса. События: TaskAssigned, TaskCompleted.

Вариант 8. Транзакция по счёту (BankTransaction): при зачислении → обновление баланса, отправка SMS-уведомления. При списании → проверка лимитов, уведомление (если превышен лимит). События: Deposited, Withdrawn.

Вариант 9. Событие в календаре (CalendarEvent): при создании → рассылка приглашений участникам. При изменении времени → отмена старых, рассылка обновлений. События: EventCreated, EventRescheduled.

Вариант 10. Заявка в техподдержку (SupportTicket): при открытии → отправка автоответа, заведение задачи в системе. При закрытии → отправка опроса удовлетворённости, запись времени решения. События: TicketOpened, TicketClosed.

Вариант 11. Комментарий к статье (Comment): при добавлении → уведомление автору статьи, проверка на спам. При удалении модератором → лог удаления, уведомление автору комментария. События: CommentAdded, CommentDeleted.

Вариант 12. Подписка в блоге (Subscription): при оформлении → отправка подтверждения, добавление в список рассылки. При отписке → удаление из рассылки, запрос причины. События: Subscribed, Unsubscribed.

Вариант 13. Встреча (Meeting): при создании → резервирование переговорной, добавление в календари участников. При отмене →

освобождение комнаты, уведомление участников. События: MeetingScheduled, MeetingCancelled.

Вариант 14. Ставка на аукционе (Bid): при поступлении новой ставки → проверка повышения минимальной, уведомление предыдущего лидера, лог аукциона. События: BidPlaced, AuctionClosed.

Вариант 15. Посылка (Shipment): при создании → печать наклейки, отправка трекинга клиенту. При доставке → обновление статуса, отправка уведомления. События: ShipmentCreated, ShipmentDelivered.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО студента, вариантом.
2. **Цель и задачи работы**.
3. **Теоретическая часть** – описание Domain Event, локальной Event Bus, синхронной и асинхронной обработки.
4. **Ход выполнения**:
   - Исходный сервис с прямыми вызовами.
   - Создание событий.
   - Реализация Event Bus.
   - Модификация сервиса (использование шины).
   - Реализация подписчиков.
   - Регистрация подписчиков (точка сборки).
5. **Результаты тестирования** – код интеграционных тестов, проверяющих публикацию событий.
6. **Оценка компромиссов** – сравнение старого и нового кода по метрикам связности, тестируемости, сложности отладки.
7. **Вывод** – достигнуты ли цели, приобретённые навыки.
8. **Ответы на контрольные вопросы**.

Контрольные вопросы

1. Чем доменное событие отличается от обычного сообщения в очереди (JMS, Kafka)?

2. Какие преимущества даёт использование Event Bus внутри одного процесса по сравнению с прямыми вызовами?

3. В чём недостатки синхронной публикации событий?

4. Как бы вы реализовали асинхронную Event Bus? Какие проблемы могут возникнуть при обработке событий в разных потоках?

5. Как гарантировать, что событие будет опубликовано только после успешного завершения транзакции (например, сохранения в БД)?

6. Как протестировать, что подписчик был вызван именно с ожидаемыми данными события?

7. Как зарегистрировать несколько подписчиков на одно событие и определить порядок их вызова?

8. Может ли один подписчик быть подписан на несколько типов событий? Как это реализовать?

9. Как Event Bus связан с паттерном Observer (классическим) и Mediator?

10. Почему в данной работе не используются внешние брокеры (RabbitMQ, Kafka)? Что нужно изменить, чтобы перейти на них?

Лабораторная работа №14. CQRS на монолите: разделение Command и Query

Цель работы: очистить и агрегировать код приложения, разделив модели записи (Command) и чтения (Query) в рамках монолитной системы, используя синхронизацию через доменные события (Event Bus из ЛР13), и количественно сравнить производительность запросов к нормализованной и денормализованной моделям.

Задачи:

1. Проанализировать существующее монолитное приложение, где одна модель данных (нормализованная БД) используется как для команд (изменения), так и для сложных запросов (отчёты, поиск), что приводит к низкой производительности чтения.
2. Реализовать **Command side** – нормализованную модель данных (стандартные сущности с внешними ключами) и репозитории для записи.
3. Реализовать **Query side** – денормализованное представление (например, одну плоскую таблицу или несколько материализованных представлений), оптимизированное для чтения.
4. Настроить синхронизацию между Command и Query моделями через доменные события (Event Bus), опубликованные после успешного выполнения команд.
5. Реализовать подписчиков (проекторы), которые обновляют денормализованное представление при получении событий (например, OrderPlacedEvent, OrderStatusChangedEvent).
6. Обеспечить, чтобы запросы выполнялись только к Query side, а команды – только к Command side (разделение интерфейсов).
7. Провести тесты производительности: сравнить время выполнения сложного запроса (например, поиск заказов с фильтрацией) к Command модели (с JOIN) и к Query модели (прямой поиск).

8. Написать тесты, проверяющие консистентность между Command и Query моделями (в пределах допустимой задержки синхронизации).
9. Оценить компромиссы: увеличение сложности (две модели, синхронизация) против прироста производительности чтения и упрощения запросов.

Теоретическая часть

Проблема одной модели для записи и чтения

В традиционных приложениях используется одна и та же реляционная модель для выполнения операций создания/обновления и для сложных запросов. Это приводит к:

- **Низкой производительности чтения** при сложных JOIN для отчётов и поиска.
- **Конфликтам:** оптимизация для запросов ухудшает производительность команд, и наоборот.
- **Сложности масштабирования:** невозможно независимо масштабировать нагрузку на чтение и запись.

Паттерн CQRS (Command Query Responsibility Segregation)

Назначение: разделяет ответственность за выполнение команд (изменение состояния) и запросов (чтение данных), позволяя использовать разные модели данных, разные хранилища и даже разные архитектуры для каждой стороны.

Структура CQRS на монолите:

- **Command Model (Write Model):** нормализованная, соблюдает бизнес-правила, гарантирует целостность. Команды (например, CreateOrderCommand) вызывают методы доменных сущностей и сохраняют через репозитории.
- **Query Model (Read Model):** денормализованная, оптимизированная для чтения. Не содержит бизнес-логики. Обычно одно или несколько

материализованных представлений (например, один объект на заказ со всеми необходимыми полями).

- **Синхронизация:** после выполнения команды публикуются доменные события. Подписчики (проекторы) обновляют Query модель. Синхронизация может быть синхронной (в той же транзакции) или асинхронной (через очередь – в рамках монолита можно сделать синхронной для простоты, но тогда теряется часть преимуществ).

Разделение интерфейсов:

- Клиент для изменения вызывает CommandService (например, placeOrder).
- Клиент для чтения вызывает QueryService (например, getOrderSummary, searchOrders).

Почему не полноценный CQRS? (в рамках монолита)

Полноценный CQRS подразумевает раздельные хранилища (разные БД), часто асинхронную синхронизацию через брокер сообщений. В данной лабораторной работе используется **упрощённый CQRS на монолите**:

- Оба хранилища могут находиться в одной БД (разные таблицы) или в памяти (in-memory для простоты).
- Синхронизация через синхронный Event Bus (из ЛР13) – консистентность в рамках одной транзакции или практически мгновенная.

Компромиссы

Аспект	Традиционный CRUD	CQRS на монолите
Сложность запроса (сложный JOIN)	Высокая, медленный	Низкая (плоская таблица), быстрый
Простота поддержки бизнес-правил	Хорошая	Хорошая (команды в одной модели)

Аспект	Традиционный CRUD	CQRS на монолите
Сложность кода	Низкая	Высокая (две модели, синхронизация)
Консистенция (чтение после записи)	Мгновенная	Может быть задержка (если асинхронно)
Гибкость масштабирования	Низкая	Средняя (можно разделить позже)

Trade-off: CQRS окупается, когда:

- Запросы к чтению сложные и частые.
- Нагрузка на чтение и запись сильно различается.
- Нужны разные представления для разных пользователей.

В простых CRUD-приложениях это избыточно.

Методика выполнения работы

Шаг 1. Исходное приложение (проблемное)

Получите вариант – монолитное приложение с одной моделью данных (например, заказы, клиенты, товары). Класс-репозиторий выполняет все запросы (и команды, и чтение) через сложные JOIN. Тесты показывают медленную работу чтения.

Шаг 2. Проектирование Command модели

Оставьте существующую нормализованную модель как есть. Обычно это классы сущностей: Order, Customer, Product, OrderItem. Репозиторий для записи (Save, Update, Delete) остаётся.

Шаг 3. Проектирование Query модели

Создайте денормализованное представление, например, класс OrderReadModel с полями, объединяющими информацию из заказа, заказчика

и товаров (например, orderId, customerName, total, status, productNames – список или агрегированная строка). Храните коллекцию Map<String, OrderReadModel> (in-memory) или таблицу в БД.

Шаг 4. Синхронизация через Event Bus (из ЛР13)

Используйте готовый Event Bus. Определите события, которые будут публиковаться после успешного выполнения команд (например, OrderPlacedEvent, OrderItemAddedEvent, OrderStatusChangedEvent).

Шаг 5. Реализация проекторов (подписчиков)

Создайте класс OrderReadModelProjector, который подписывается на события и обновляет Query модель:

- handle(OrderPlacedEvent) – создаёт запись в OrderReadModel.
- handle(OrderStatusChangedEvent) – обновляет статус.

Шаг 6. Реализация Command и Query сервисов

- **CommandService** – принимает команды, изменяет Command модель, публикует события.
- **QueryService** – работает только с Query моделью (например, findOrderById, searchByCustomerName). Не имеет права изменять состояние.

Шаг 7. Настройка и связывание

В точке входа (Main / тест):

- Создайте Command репозиторий.
- Создайте Event Bus.
- Создайте Projector и подпишите его на события.
- Создайте CommandService и QueryService.

Шаг 8. Тесты производительности

Напишите тест, который:

- Создаёт N заказов (например, 1000) через CommandService.
- Замеряет время выполнения сложного запроса (поиск заказов клиента с фильтром по сумме) на Command модели (с JOIN) и на Query модели (прямой поиск).
- Выводит соотношение.

Шаг 9. Тест консистенции

Проверьте, что после выполнения команды Query модель обновляется (синхронно или в течение допустимого времени).

Шаг 10. Оценка компромиссов

В отчёте сравните производительность и сложность поддержки обоих подходов.

Пример выполнения задания

Исходная Command модель (нормализованная)

```
// Сущности
public class Order {
    private String id;
    private String customerId;
    private LocalDate orderDate;
    private String status;
    private List<OrderItem> items;
}

public class Customer {
    private String id;
    private String name;
```

```
    private String email;
}

public class OrderItem {
    private String productId;
    private int quantity;
    private double price;
}
```

Команда размещения заказа (упрощённо):

java

```
public class PlaceOrderCommand {
    private String customerId;
    private List<OrderItemData> items;
}

public class OrderCommandService {
    private OrderRepository orderRepo;
    private CustomerRepository customerRepo;
    private EventBus eventBus;

    public void placeOrder(PlaceOrderCommand cmd) {
        // Валидация, бизнес-логика
        Customer customer = customerRepo.findById(cmd.getCustomerId());
        Order order = new Order(... customer ... items ...);
        orderRepo.save(order);
        eventBus.publish(new OrderPlacedEvent(order.getId(),
customer.getName(), order.getTotal()));
    }
}
```

Query модель (денормализованная)

```
public class OrderReadModel {
    private String orderId;
    private String customerName;
    private double total;
    private String status;
    private String orderDate;
    private int itemCount;
    // геттеры/сеттеры
}

public class InMemoryOrderReadRepository {
    private Map<String, OrderReadModel> store = new ConcurrentHashMap<>();

    public void save(OrderReadModel model) { store.put(model.getOrderId(), model); }
    public void update(String orderId, String status) {
        OrderReadModel m = store.get(orderId);
        if (m != null) m.setStatus(status);
    }
    public OrderReadModel findById(String id) { return store.get(id); }
    public List<OrderReadModel> findByCustomerName(String name) {
        return store.values().stream()
            .filter(m -> m.getCustomerName().equalsIgnoreCase(name))
            .collect(Collectors.toList());
    }
}
```

Проектор (синхронизация)

```
public class OrderReadModelProjector {
    private InMemoryOrderReadRepository readRepo;

    public OrderReadModelProjector(InMemoryOrderReadRepository readRepo) {
        this.readRepo = readRepo;
    }

    public void on(OrderPlacedEvent event) {
        OrderReadModel model = new OrderReadModel();
        model.setOrderId(event.getOrderId());
        model.setCustomerName(event.getCustomerName());
        model.setTotal(event.getTotal());
        model.setStatus("NEW");
        model.setOrderDate(LocalDate.now().toString());
        readRepo.save(model);
    }

    public void on(OrderStatusChangedEvent event) {
        readRepo.update(event.getOrderId(), event.getNewStatus());
    }
}
```

Тест производительности (пример)

```
@Test
void performanceTest() {
    // Создаём 500 заказов через CommandService
    for (int i = 0; i < 500; i++) {
```

```
            commandService.placeOrder(new PlaceOrderCommand("cust" +
(i%10), ...));
        }

        // Запрос к Command модели (с JOIN) – симулируем через репозиторий
        long startCmd = System.nanoTime();
        List<Order> cmdResult =
orderCommandRepo.findByCustomerNameWithJoin("cust1");
        long timeCmd = System.nanoTime() - startCmd;

        // Запрос к Query модели (прямой поиск)
        long startQuery = System.nanoTime();
        List<OrderReadModel> queryResult =
orderReadRepo.findByCustomerName("cust1");
        long timeQuery = System.nanoTime() - startQuery;

        System.out.println("Command model query: " + timeCmd/1_000_000 + "
ms");
        System.out.println("Query model query: " + timeQuery/1_000_000 + "
ms");
        assertTrue(timeQuery < timeCmd); // ожидаем быстрее
    }
```

Компромиссы в примере

- **Прирост производительности:** запрос к плоской таблице без JOIN может быть в 10–100 раз быстрее (зависит от объёма).
- **Дополнительный код:** добавилось 3–4 класса (ReadModel, проектор, Query репозиторий).
- **Синхронизация:** при синхронной публикации событий команда не завершится, пока не обновится Query модель, что увеличивает время

выполнения команды. Для чистой CQRS нужно асинхронное обновление.

Индивидуальные задания

Каждый вариант – домен, в котором есть сложные запросы к нормализованной модели. Требуется:

- Выделить Command модель (существующую).
- Спроектировать денормализованную Query модель (1–2 представления).
- Реализовать синхронизацию через события (используя Event Bus из ЛР13).
- Написать Command и Query сервисы.
- Провести тесты производительности и сравнить.

Вариант 1. Система заказов: Command модель: Order, Customer, Product, OrderItem. Query модель: OrderSummary (поля: ID заказа, имя клиента, общая сумма, статус, количество товаров, дата). Сложный запрос: найти заказы клиента за последний месяц с суммой > 1000.

Вариант 2. Библиотека: Command: Book, Author, BorrowRecord. Query: BookBorrowingHistory (название книги, автор, дата выдачи, читатель, статус возврата). Запрос: список книг, которые сейчас на руках у конкретного читателя.

Вариант 3. Курсы и студенты: Command: Student, Course, Enrollment. Query: StudentCourseReport (имя студента, список курсов с оценками, средний балл). Запрос: успеваемость студента по всем курсам.

Вариант 4. Финансовые транзакции: Command: Account, Transaction. Query: AccountStatement (номер счёта, баланс, список последних транзакций с датами). Запрос: выписка по счёту за период.

Вариант 5. Управление проектами: Command: Project, Task, Employee. Query: ProjectProgress (название проекта, количество задач, завершённые задачи, ответственный). Запрос: проекты с просроченными задачами.

Вариант 6. Интернет-магазин: Command: Product, Category, Review. Query: ProductDetails (название, категория, средний рейтинг, количество отзывов, цена). Запрос: топ-10 популярных продуктов.

Вариант 7. Служба доставки: Command: Package, DeliveryRoute, Courier. Query: DeliveryStatus (ID посылки, статус, имя курьера, ожидаемая дата). Запрос: все посылки в доставке определённого курьера.

Вариант 8. Больничная система: Command: Patient, Appointment, Doctor. Query: AppointmentSchedule (пациент, доктор, дата и время, кабинет). Запрос: расписание доктора на день.

Вариант 9. Социальная сеть: Command: User, Post, Like. Query: UserActivityFeed (автор поста, текст, количество лайков, дата). Запрос: лента постов для пользователя (по друзьям).

Вариант 10. Система голосования: Command: Poll, Option, Vote. Query: PollResults (вопрос, варианты, количество голосов, победитель). Запрос: текущие результаты опроса.

Вариант 11. Логистика: Command: Shipment, Warehouse, Inventory. Query: StockLevel (товар, склад, текущее количество, минимальный запас). Запрос: товары, которые нужно заказать (ниже минимума).

Вариант 12. HR-система: Command: Employee, Department, SalaryHistory. Query: EmployeeOverview (ФИО, департамент, текущая зарплата, дата найма). Запрос: сотрудники с зарплатой выше средней по департаменту.

Вариант 13. Авиабилеты: Command: Flight, Booking, Passenger. Query: FlightManifest (рейс, список пассажиров, занятость мест). Запрос: список пассажиров на рейсе.

Вариант 14. Система обучения: Command: Course, Lesson, Progress. Query: CourseCompletion (курс, общее количество уроков, пройденные, процент). Запрос: курсы, которые пользователь прошёл более чем на 80%.

Вариант 15. Резюме и вакансии: Command: Resume, Vacancy, Match. Query: JobMatchScore (резюме, вакансия, процент совпадения, ключевые навыки). Запрос: топ-5 лучших резюме для заданной вакансии.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи работы**.
3. **Теоретическая часть** – объяснение CQRS, разделение Command и Query моделей, синхронизация через события, компромиссы.
4. **Ход выполнения**:
   - Описание исходной Command модели (сущности, их связи).
   - Денормализованная Query модель (структура).
   - Реализация событий и проектора (код).
   - Реализация Command и Query сервисов.
   - Настройка шины событий.
   - Тесты производительности (код и результаты).
5. **Результаты тестирования** – таблица с замерами времени выполнения запросов на Command и Query моделях.
6. **Оценка компромиссов** – обсуждение, оправдано ли CQRS в вашем варианте, анализ сложности.
7. **Вывод** – достигнуты ли цели, приобретённые навыки.
8. **Ответы на контрольные вопросы**.

Контрольные вопросы

1. В чём суть разделения Command и Query в CQRS?
2. Почему одну и ту же модель данных неэффективно использовать для сложных запросов?
3. Как в вашей реализации синхронизируются Command и Query модели? Какая консистенция обеспечивается?
4. Какие проблемы могут возникнуть при асинхронной синхронизации (отсутствует в данной работе, но можно обсудить)?
5. Как тестировалась производительность? Какие метрики сравнивались?
6. Какие изменения потребовались в клиентском коде для работы с разделёнными интерфейсами?
7. Как вы обеспечили, что Query модель не содержит бизнес-логики?
8. Можно ли в рамках CQRS использовать разные СУБД для Command и Query (например, PostgreSQL и Elasticsearch)? Что для этого нужно?
9. В каких случаях CQRS избыточен? Приведите пример из реальной практики.
10. Как CQRS связан с паттернами Event Sourcing и Domain Events?

Лабораторная работа №15. Event Sourcing и восстановление состояния

Цель работы: очистить и агрегировать код управления состоянием агрегата путём перехода от CRUD-подхода к хранению потока доменных событий (Event Store), обеспечив возможность полного восстановления текущего состояния через воспроизведение (replay) событий, а также ускорить этот процесс с помощью механизма снапшотов.

Задачи:

1. Проанализировать существующий CRUD-код, который сохраняет только текущее состояние агрегата (например, заказ, счёт, корзину), теряя историю изменений и возможность аудита.
2. Реализовать набор **доменных событий** для всех изменений агрегата (создание, изменение полей, удаление).
3. Реализовать **Event Store** – хранилище событий (в памяти, в файле или в простой БД), которое сохраняет события в хронологическом порядке с возможностью чтения всех событий для агрегата.
4. Реализовать **агрегат**, который:
   - Содержит private конструктор (или фабричный метод) для создания «пустого» агрегата.
   - Имеет метод applyEvent(Event) для изменения состояния (без проверок бизнес-правил, только модификация).
   - Имеет публичные методы-команды, которые проверяют бизнес-правила и порождают события (но не изменяют состояние напрямую).
   - Позволяет загрузить историю через loadFromHistory(List<Event>) (replay).
5. Реализовать механизм **снапшотов (snapshots)** – сохранение полного состояния агрегата после определённого количества событий (например, каждые 100 событий) для ускорения восстановления.

6. Написать модульные тесты, проверяющие:
   - Корректное создание агрегата через события.
   - Восстановление состояния из событий (replay).
   - Использование снапшотов для ускорения.
7. Сравнить производительность восстановления состояния из событий без снапшота и со снапшотом при большом количестве событий (например, 10 000 событий).
8. Оценить компромиссы: полная история и аудит против сложности реализации и объёма хранения.

Теоретическая часть

Проблема CRUD-подхода

Традиционный CRUD (Create-Read-Update-Delete) хранит только текущее состояние агрегата. Недостатки:

- **Потеря истории**: нельзя узнать, как состояние менялось со временем.
- **Сложный аудит**: нужно проектировать отдельные таблицы логов.
- **Невозможность «отката» во времени**: трудно восстановить состояние на любой момент.
- **Ограниченная трассировка ошибок**: непонятно, какая последовательность операций привела к багу.

Паттерн Event Sourcing (Хранилище событий)

Назначение: сохраняет все изменения состояния приложения как последовательность неизменяемых событий. Текущее состояние агрегата вычисляется путём повторного применения (replay) всех событий с момента создания.

Структура:

- **Event** – неизменяемый объект, описывающий факт изменения (например, OrderCreated, ItemAdded, OrderShipped). Содержит данные изменения и метаданные (время, версия агрегата).
- **Aggregate** – сущность, которая:
  - Имеет методы-команды (placeOrder, addItem). Каждая команда проверяет бизнес-правила и возвращает (или сохраняет в списке) события.
  - Имеет метод apply(Event) (обычно приватный), который обновляет состояние агрегата на основе события.
  - Загружается через load(Iterable<Event> history) – последовательно применяет все события.
- **Event Store** – хранилище, которое сохраняет события по агрегату. Обычно поддерживает:
  - saveEvents(aggregateId, events, expectedVersion) – для конкурентности.
  - loadEvents(aggregateId) – чтение всех событий для восстановления.
- **Snapshot** (опционально) – сохранённое полное состояние агрегата на определённый момент, чтобы не переигрывать всю историю с начала.

Event Sourcing + CQRS

Event Sourcing часто сочетается с CQRS: команды порождают события, события обновляют read model. В данной лабораторной работе фокус на самом Event Sourcing и восстановлении состояния.

Компромиссы

Аспект	CRUD	Event Sourcing
История изменений	Нет	Полная (аудит forever)
Сложность реализации	Низкая	Высокая (много классов событий, механизм replay)

Аспект	CRUD	Event Sourcing
Восстановление состояния	Мгновенное (чтение из БД)	Линейное по числу событий (можно ускорить снапшотами)
Объём хранения	Малый (только текущее состояние)	Большой (все события)
Трассировка и отладка	Трудная	Лёгая (можно воспроизвести баг)
Модификация бизнес-правил	Мгновенная	Требует миграции событий (версионирование)

Trade-off: Event Sourcing идеален для систем, где критичен аудит, необходимость отката, сложная бизнес-логика с длинным жизненным циклом. Для простых справочников он избыточен.

Методика выполнения работы

Шаг 1. Исходный CRUD-код

Получите вариант – простой агрегат (например, BankAccount, ShoppingCart, Order), который сохраняет текущее состояние в репозитории. Нет истории.

Шаг 2. Определение событий

Выделите все возможные изменения состояния агрегата в виде событий с необходимыми данными. Обычно минимум три события: создание, изменение, удаление. Для каждого события создайте класс (immutable, с геттерами и полем occurredAt).

Шаг 3. Реализация агрегата с событиями

- Сделайте агрегат иммутабельным или с внутренним состоянием, изменяемым только через applyEvent.

- В публичных командных методах (например, deposit(amount)) сначала проверьте бизнес-правила (например, amount > 0), затем сгенерируйте событие (MoneyDeposited) и примените его к себе через applyEvent.
- Метод applyEvent – приватный, обновляет состояние (например, balance += event.amount). Не выполняет проверок.
- Метод loadFromHistory(events) последовательно вызывает applyEvent для каждого события.

Шаг 4. Реализация Event Store (in-memory или файл)

Создайте класс EventStore:

- Хранит Map<AggregateId, List<Event>> (in-memory) или дописывает события в файл.
- Метод save(aggregateId, events) – добавляет события в список.
- Метод load(aggregateId) – возвращает все события для агрегата.
- (Опционально) поддержка оптимистичной блокировки через expectedVersion.

Шаг 5. Реализация снапшотов

- Создайте класс Snapshot (поля: aggregateId, state, version, createdAt).
- В агрегате добавьте метод takeSnapshot() – возвращает текущее состояние в виде сериализуемого объекта.
- В Event Store добавьте методы saveSnapshot, loadLatestSnapshot.
- При восстановлении: сначала загружаем последний снапшот (если есть), затем загружаем и применяем события, произошедшие после версии снапшота.

Шаг 6. Сервис (репозиторий агрегатов)

Создайте класс AggregateRepository:

- Метод save(aggregate) – получает новые события из агрегата (например, через getUncommittedEvents()), сохраняет их в Event Store, затем

обновляет снапшот, если число событий с последнего снапшота превышает порог (например, 100).

- Метод load(id) – загружает снапшот (если есть), затем грузит события, начиная с версии снапшота, и восстанавливает агрегат.

Шаг 7. Тестирование

Напишите тесты:

- Создание агрегата, выполнение команд, сохранение и загрузка – состояние должно совпасть.
- Восстановление после 1 000 событий без снапшота – замерить время.
- Восстановление со снапшотом после 1 000 событий – время должно быть существенно меньше.
- Проверка, что после сохранения снапшот обновляется только при достижении порога.

Шаг 8. Оценка компромиссов

Сравните CRUD и Event Sourcing по объёму кода, производительности восстановления, возможностям аудита.

Пример выполнения задания

Исходный CRUD-агрегат (проблемный)

```
public class BankAccount {
   private String id;
   private double balance;
   private String owner;

   public void deposit(double amount) {
      balance += amount;
   }
   public void withdraw(double amount) {
```

```
        if (amount > balance) throw new IllegalStateException();
        balance -= amount;
    }
}
```

События

```
public abstract class Event {
    private final LocalDateTime occurredAt = LocalDateTime.now();
    // геттер
}

public class AccountOpened extends Event {
    private final String accountId;
    private final String owner;
    private final double initialBalance;
    // конструктор, геттеры
}

public class MoneyDeposited extends Event {
    private final double amount;
    // конструктор, геттер
}

public class MoneyWithdrawn extends Event {
    private final double amount;
    // конструктор, геттер
}
```

Агрегат с Event Sourcing

```
public class BankAccount {
```

```
    private String id;
    private double balance;
    private String owner;
    private int version = 0;
    private final List<Event> uncommittedEvents = new ArrayList<>();

    // Команда: открыть счёт
    public static BankAccount open(String id, String owner, double initialBalance) {
        BankAccount account = new BankAccount();
        account.applyChange(new AccountOpened(id, owner, initialBalance));
        return account;
    }

    public void deposit(double amount) {
        if (amount <= 0) throw new IllegalArgumentException("Amount must be positive");
        applyChange(new MoneyDeposited(amount));
    }

    public void withdraw(double amount) {
        if (amount <= 0) throw new IllegalArgumentException();
        if (balance < amount) throw new IllegalStateException("Insufficient funds");
        applyChange(new MoneyWithdrawn(amount));
    }

    private void applyChange(Event event) {
        apply(event);
        uncommittedEvents.add(event);
```

```
        version++;
    }

    private void apply(Event event) {
        if (event instanceof AccountOpened) {
            AccountOpened e = (AccountOpened) event;
            this.id = e.getAccountId();
            this.owner = e.getOwner();
            this.balance = e.getInitialBalance();
        } else if (event instanceof MoneyDeposited) {
            this.balance += ((MoneyDeposited) event).getAmount();
        } else if (event instanceof MoneyWithdrawn) {
            this.balance -= ((MoneyWithdrawn) event).getAmount();
        }
    }

    public void loadFromHistory(List<Event> history) {
        for (Event e : history) {
            apply(e);
            version++;
        }
    }

    public List<Event> getUncommittedEvents() { return uncommittedEvents;
}
    public void markEventsCommitted() { uncommittedEvents.clear(); }
    public int getVersion() { return version; }
    public double getBalance() { return balance; }
    // геттеры id, owner
}
```

Event Store (in-memory)

```
public class InMemoryEventStore {
    private final Map<String, List<Event>> store = new HashMap<>();
    private final Map<String, List<Event>> snapshotEvents = new HashMap<>(); // для простоты

    public void saveEvents(String aggregateId, List<Event> events, int expectedVersion) {
        List<Event> existing = store.computeIfAbsent(aggregateId, k -> new ArrayList<>());
        if (existing.size() != expectedVersion) {
            throw new ConcurrentModificationException("Version mismatch");
        }
        existing.addAll(events);
    }

    public List<Event> loadEvents(String aggregateId) {
        return store.getOrDefault(aggregateId, Collections.emptyList());
    }
}
```

Снапшот

```
public class BankAccountSnapshot {
    private final String aggregateId;
    private final double balance;
    private final String owner;
    private final int version;
    // конструктор, геттеры
}
```

```
    // Добавляем методы в Event Store
    public void saveSnapshot(BankAccountSnapshot snapshot) { /* храним map
*/ }
    public BankAccountSnapshot loadLatestSnapshot(String aggregateId) { /* ...
*/ }
```

Репозиторий со снапшотами

```
    public class BankAccountRepository {
        private final InMemoryEventStore eventStore;
        private final Map<String, BankAccountSnapshot> snapshots = new
HashMap<>();
        private final int snapshotThreshold = 100;

        public void save(BankAccount account) {
            List<Event> newEvents = account.getUncommittedEvents();
            if (newEvents.isEmpty()) return;
            eventStore.saveEvents(account.getId(), newEvents,
account.getVersion() - newEvents.size());
            account.markEventsCommitted();

            // проверяем, нужно ли сделать снапшот
            int totalEvents = account.getVersion();
            int lastSnapshotVersion = getSnapshotVersion(account.getId());
            if (totalEvents - lastSnapshotVersion >= snapshotThreshold) {
                BankAccountSnapshot snapshot = new
BankAccountSnapshot(account.getId(), account.getBalance(), account.getOwner(),
account.getVersion());
                snapshots.put(account.getId(), snapshot);
            }
```

```
    }

    public BankAccount load(String id) {
        BankAccountSnapshot snapshot = snapshots.get(id);
        List<Event> events;
        int startVersion = 0;
        if (snapshot != null) {
            startVersion = snapshot.getVersion();
            events = eventStore.loadEvents(id).stream()
                    .skip(startVersion).collect(Collectors.toList());
        } else {
            events = eventStore.loadEvents(id);
        }
        BankAccount account = new BankAccount();
        account.loadFromHistory(events);
        return account;
    }
}
```

Тест производительности

```
@Test
void testPerformanceWithSnapshot() {
    BankAccount account = BankAccount.open("acc1", "John", 1000);
    // симулируем 10 000 событий
    for (int i = 0; i < 10_000; i++) {
        account.deposit(1);
        repo.save(account);
        account = repo.load("acc1");
    }
    // Замеряем загрузку без снапшота (принудительно очистив снапшот)
```

```
        long startNoSnap = System.nanoTime();
        BankAccount loadedNoSnap = repo.loadNoSnapshot("acc1");
        long timeNoSnap = System.nanoTime() - startNoSnap;

        // Замеряем загрузку со снапшотом
        long startWithSnap = System.nanoTime();
        BankAccount loadedWithSnap = repo.load("acc1");
        long timeWithSnap = System.nanoTime() - startWithSnap;

        System.out.println("Without snapshot: " + timeNoSnap / 1_000_000 + "
ms");
        System.out.println("With snapshot: " + timeWithSnap / 1_000_000 + "
ms");
        assertTrue(timeWithSnap < timeNoSnap);
    }
```

Компромиссы

- **Добавлено:** 5+ классов событий, агрегат с event sourcing, Event Store, репозиторий со снапшотами. Объём кода вырос в 3–4 раза.
- **Возможности:** полный аудит, возможность восстановления на любой момент, отладка через replay.
- **Производительность:** со снапшотами восстановление O(k) где k – число событий после снапшота (например, 100), без снапшотов – O(N) для N событий (10 000 может быть медленно).
- **Объём хранения:** все события хранятся вечно (можно архивировать старые).

Индивидуальные задания

Каждый вариант – доменный агрегат, который необходимо перевести на Event Sourcing, включая минимум 3 типа событий, Event Store и снапшоты.

Требуется реализовать команды, восстановление состояния, тесты производительности.

Вариант 1. ShoppingCart (корзина покупок): события – CartCreated, ItemAdded, ItemRemoved, CartCheckedOut. Команды: добавить товар, удалить, оформить. Бизнес-правила: нельзя добавить товар с отрицательным количеством, нельзя оформить пустую корзину.

Вариант 2. Task (задача в трекере): события – TaskCreated, StatusChanged, AssigneeChanged, TaskDeleted. Команды: создать, изменить статус (новый → в работе → выполнена → закрыта), назначить исполнителя.

Вариант 3. Subscription (подписка): события – SubscriptionStarted, PaymentRecorded, SubscriptionCancelled, TierChanged. Команды: начать подписку, внести оплату, отменить, изменить тариф.

Вариант 4. Document (документ с версиями): события – DocumentCreated, ContentUpdated, DocumentPublished, Archived. Команды: создать, обновить, опубликовать, архивировать. Хранить версию текста.

Вариант 5. FlightBooking (бронирование авиабилета): события – BookingCreated, SeatAssigned, PassengerDetailsUpdated, BookingCancelled. Команды: создать бронь, назначить место, обновить пассажира, отменить.

Вариант 6. BankAccount (банковский счёт) – расширенный: события – AccountOpened, MoneyDeposited, MoneyWithdrawn, InterestApplied, AccountFrozen, AccountUnfrozen. Команды: открыть, внести, снять, начислить проценты, заморозить/разморозить.

Вариант 7. VotingPoll (опрос с вариантами): события – PollCreated, VoteCast, PollClosed. Команды: создать опрос, проголосовать, закрыть. Хранить список вариантов и подсчёт голосов.

Вариант 8. InventoryItem (складской товар): события – ItemCreated, StockAdded, StockRemoved, Reserved, Released. Команды: создать, пополнить, отгрузить, зарезервировать, снять резерв.

Вариант 9. Employee (сотрудник): события – EmployeeHired, SalaryChanged, DepartmentTransfered, EmployeeFired. Команды: нанять, изменить зарплату, перевести в отдел, уволить.

Вариант 10. Reservation (бронирование столика): события – ReservationMade, ReservationConfirmed, ReservationCancelled, TableAssigned. Команды: забронировать, подтвердить, отменить, назначить столик.

Вариант 11. GameCharacter (персонаж игры): события – CharacterCreated, LevelUp, ItemEquipped, ItemUnequipped. Команды: создать, повысить уровень, экипировать/снять предмет.

Вариант 12. SupportTicket (обращение в поддержку): события – TicketCreated, AssignedToAgent, StatusChanged, TicketClosed. Команды: открыть тикет, назначить агента, изменить статус, закрыть.

Вариант 13. Contract (договор): события – ContractDrafted, ContractSigned, ContractAmended, ContractTerminated. Команды: подготовить, подписать, изменить, расторгнуть.

Вариант 14. LoyaltyAccount (бонусный счёт): события – AccountActivated, PointsEarned, PointsSpent, PointsExpired. Команды: активировать счет, начислить бонусы, списать бонусы, начислить сгоревшие.

Вариант 15. Meeting (встреча): события – MeetingScheduled, AttendeeAdded, AttendeeRemoved, TimeChanged, MeetingCancelled. Команды: создать встречу, добавить участника, удалить, изменить время, отменить.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи работы**.
3. **Теоретическая часть** – описание Event Sourcing, Event Store, снапшотов, компромиссов.
4. **Ход выполнения**:
   - Исходный CRUD-агрегат.
   - События и их поля.
   - Реализация агрегата с командами и apply.
   - Реализация Event Store (in-memory или файл).
   - Реализация снапшотов и репозитория.
   - Демонстрация сохранения и загрузки.
5. **Результаты тестирования**:
   - Тесты бизнес-логики (команды, проверки).
   - Тест восстановления из истории.
   - Тест производительности (без снапшота и со снапшотом) с графиком или таблицей.
6. **Оценка компромиссов** – обсуждение применимости Event Sourcing для данного агрегата.
7. **Вывод** – достигнуты ли цели, приобретённые навыки.
8. **Ответы на контрольные вопросы**.

Контрольные вопросы

1. Чем Event Sourcing отличается от традиционного CRUD с точки зрения хранения состояния?

2. Как в вашей реализации агрегат восстанавливается из событий? Что такое replay?

3. Какие преимущества даёт хранение событий вместо текущего состояния для аудита и отладки?

4. Что такое снапшот и зачем он нужен? Как часто в вашей работе создаются снапшоты?

5. Как решается проблема конкурентного доступа при сохранении событий (optimistic concurrency)?

6. Можно ли модифицировать бизнес-правила после того, как уже накопилось много событий? Какие проблемы это создаёт?

7. Как бы вы реализовали удаление агрегата (например, по GDPR) при Event Sourcing?

8. Какие типичные проблемы возникают при версионировании событий (изменение структуры)?

9. Как Event Sourcing связан с паттерном CQRS? Можно ли использовать один без другого?

10. Оцените накладные расходы на хранение событий по сравнению с CRUD для вашего варианта (размер хранилища, скорость записи).

Лабораторная работа №16. Saga (хореография) для компенсирующих транзакций

Цель работы: очистить и агрегировать код, реализовав паттерн Saga в стиле хореографии для управления распределёнными транзакциями внутри одного процесса, где несколько подсистем (например, биллинг и логистика) взаимодействуют через Event Bus, выполняя локальные транзакции и публикуя компенсирующие события при сбоях, без использования внешних брокеров и микросервисной архитектуры.

Задачи:

1. Проанализировать существующий код, в котором бизнес-процесс (например, оформление заказа) выполняет последовательные вызовы разных подсистем (биллинг, логистика) без возможности отката при сбоях.
2. Определить шаги саги (локальные транзакции) и соответствующие компенсирующие действия для каждого шага.
3. Реализовать набор доменных событий для каждого шага (успех и отказ), а также компенсирующих событий.
4. Реализовать локальные Event Bus (синхронный или асинхронный) для обмена событиями между подсистемами-модулями (без внешних брокеров).
5. Реализовать отдельные модули/сервисы (например, BillingService, LogisticsService), каждый из которых:
   - Подписывается на определённые события.
   - Выполняет свою локальную транзакцию (в памяти или с имитацией БД).
   - При успехе публикует событие успеха.
   - При ошибке публикует компенсирующее событие (или событие отказа) для отката предыдущих шагов.

6. Реализовать механизм компенсаций: при получении события-отказа или компенсирующего события каждый заинтересованный сервис выполняет обратное действие.
7. Написать интеграционные тесты, проверяющие:
   - Успешный сценарий (все шаги выполнены, компенсаций нет).
   - Сценарий с отказом на одном из шагов (происходит компенсация предыдущих).
8. Сравнить подходы: прямые вызовы, Saga (хореография), Saga (оркестрация) по сложности и надёжности.
9. Оценить компромиссы: асинхронность, сложность отладки, устойчивость к сбоям.

Теоретическая часть

Проблема распределённых транзакций

В приложениях, разделённых на модули или микросервисы, часто требуется выполнить операцию, затрагивающую несколько независимых подсистем (например, списать деньги и зарезервировать товар). Традиционные ACID-транзакции с двухфазной фиксацией (2PC) недоступны или нежелательны из-за блокировок, снижения производительности и высокой связанности.

Паттерн Saga (Сага)

Назначение: управляет распределённой транзакцией как последовательностью локальных транзакций, каждая из которых может быть скомпенсирована при сбое. Гарантирует, что либо все шаги выполнены, либо выполнены компенсации.

Два стиля реализации:

- **Хореография (Choreography):** каждая локальная транзакция публикует события, на которые подписываются другие сервисы. Нет центрального координатора. Сервисы сами знают, когда и что делать.
- **Оркестрация (Orchestration):** центральный координатор (оркестратор) управляет шагами Saga, вызывает сервисы и решает, когда выполнять компенсацию.

В данной работе реализуется хореография (как более децентрализованная и «событийная»). Все подсистемы общаются через Event Bus.

Структура Saga в хореографии (на примере заказа)

1. *Инициатор* (например, OrderService) публикует событие OrderPlaced.
2. BillingService слушает OrderPlaced, списывает деньги. Успех → публикует PaymentProcessed. Ошибка → публикует PaymentFailed.
3. LogisticsService слушает PaymentProcessed, резервирует товар. Успех → публикует DeliveryScheduled. Ошибка → публикует DeliveryFailed.
4. BillingService слушает DeliveryFailed (компенсирующее событие) и возвращает деньги (публикует PaymentRefunded).
5. Опционально: слушатель OrderService может отметить заказ как завершённый или отменённый.

Компенсирующие транзакции

Для каждого шага, который изменяет состояние, должно быть определено компенсирующее действие (обратная операция). Компенсация должна быть идемпотентной и может выполняться несколько раз (на случай повторных событий).

События и Event Bus

Используется Event Bus из лабораторной работы №13. События публикуются синхронно или асинхронно. В синхронном варианте компенсации происходят в том же потоке, что упрощает тестирование; в асинхронном – более устойчиво к сбоям, но сложнее отладка.

Компромиссы

Аспект	Прямые вызовы (монолит)	Saga (хореография)
Связность	Высокая (сервисы знают друг о друге)	Низкая (через события)
Возможность отката	Нет (при ошибке – частичные изменения)	Да (компенсации)
Сложность реализации	Низкая	Высокая (события, компенсации, идемпотентность)
Отладка	Линейная, простая	Трудная (нужно отслеживать поток событий)
Устойчивость к сбоям	Низкая	Средняя (можно перевыпускать события)

Trade-off: Saga (хореография) даёт слабую связанность и автоматическую компенсацию ценой увеличения сложности и необходимости проектировать события и компенсации. Для критичных бизнес-процессов (финансы, логистика) это оправдано; для простых сквозных операций – избыточно.

Методика выполнения работы

Шаг 1. Исходный код (проблемный)

Получите вариант – монолитный класс (например, OrderFacade), который последовательно вызывает методы BillingService.charge() и LogisticsService.reserve(). В случае ошибки второго шага деньги уже списаны, отката нет.

Шаг 2. Определение шагов саги и событий

Для вашего бизнес-сценария выделите:

- **Локальные транзакции** (какое действие выполняет каждый сервис).
- **События успеха** (например, PaymentSucceeded, InventoryReserved).
- **События отказа** (например, PaymentFailed, ReservationFailed).
- **Компенсирующие события** (могут совпадать с событиями отказа, если подписчик знает, что нужно сделать; или отдельные, например, RefundRequested).

Шаг 3. Реализация Event Bus (или использование готового)

Возьмите Event Bus из ЛР13 (синхронный). Для усложнения можно реализовать асинхронную версию с пулом потоков.

Шаг 4. Создание отдельных сервисов-подписчиков

Реализуйте каждый сервис как класс, который:

- Получает Event Bus через конструктор.
- В методе init() подписывается на интересующие события.
- Для каждого события имеет метод-обработчик, который выполняет локальную операцию и публикует следующее событие (или компенсацию).

Важно: обработчики должны быть идемпотентными (например, проверять, не была ли операция уже выполнена).

Шаг 5. Реализация механизма компенсаций

Для каждого шага, который имеет побочный эффект, добавьте обработку компенсирующего события. Например, BillingService слушает не только OrderPlaced, но и ReservationFailed, чтобы вернуть деньги.

Шаг 6. Инициирование Saga

Создайте инициатора (например, OrderController), который публикует первое событие (OrderPlaced) в Event Bus.

Шаг 7. Тестирование

Напишите тесты:

- **Happy path**: публикуем OrderPlaced → чек успеха, деньги списаны, товар зарезервирован.
- **Unhappy path 1**: ошибка на втором шаге (например, ReservationFailed). Проверяем, что возврат денег произошёл.
- **Unhappy path 2**: ошибка на первом шаге – компенсация не нужна (ничего не сделано).
- (Опционально) тест на идемпотентность: повторная отправка того же события не приводит к двойной компенсации.

Шаг 8. Оценка компромиссов

Сравните старый и новый код по метрикам: сложность, количество событий, надёжность при сбоях.

Пример выполнения задания

Исходный код (проблемный)

```
public class OrderProcessor {
    private BillingService billing = new BillingService();
    private LogisticsService logistics = new LogisticsService();

    public void processOrder(OrderData order) {
        billing.charge(order.getCustomerId(), order.getTotal());
        // Если дальше ошибка, деньги уже списаны!
        logistics.reserveStock(order.getItems());
        // заказ считается выполненным
    }
}
```

Определение событий

```
public class OrderPlaced { String orderId; double amount; }
public class PaymentProcessed { String orderId; String transactionId; }
public class PaymentFailed { String orderId; String reason; }
public class StockReserved { String orderId; }
public class StockReservationFailed { String orderId; String reason; }
public class PaymentRefunded { String orderId; }
```

Сервис биллинга (подписчик)

```
public class BillingService {
    private final EventBus eventBus;
    private final Map<String, String> payments = new HashMap<>(); // orderId -> transactionId

    public BillingService(EventBus eventBus) {
        this.eventBus = eventBus;
        eventBus.subscribe(OrderPlaced.class, this::handleOrderPlaced);
        eventBus.subscribe(StockReservationFailed.class, this::handleReservationFailed);
    }

    private void handleOrderPlaced(OrderPlaced event) {
        try {
            // имитация списания средств
            System.out.println("Charging " + event.getAmount() + " for order " + event.getOrderId());
            String txId = "TX-" + event.getOrderId();
            payments.put(event.getOrderId(), txId);
            eventBus.publish(new PaymentProcessed(event.getOrderId(), txId));
        } catch (Exception e) {
```

```
                eventBus.publish(new PaymentFailed(event.getOrderId(),
e.getMessage()));
            }
        }

        private void handleReservationFailed(StockReservationFailed event) {
            // возврат денег (компенсация)
            if (payments.containsKey(event.getOrderId())) {
                System.out.println("Refunding order " + event.getOrderId());
                payments.remove(event.getOrderId());
                eventBus.publish(new PaymentRefunded(event.getOrderId()));
            }
        }
    }
```

Сервис логистики

```
    public class LogisticsService {
        private final EventBus eventBus;
        private final Set<String> reserved = new HashSet<>();

        public LogisticsService(EventBus eventBus) {
            this.eventBus = eventBus;
            eventBus.subscribe(PaymentProcessed.class,
this::handlePaymentProcessed);
        }

        private void handlePaymentProcessed(PaymentProcessed event) {
            try {
                // имитация резервирования
```

```
            if (Math.random() < 0.2) throw new RuntimeException("Out of stock");
            System.out.println("Reserving stock for order " + event.getOrderId());
            reserved.add(event.getOrderId());
            eventBus.publish(new StockReserved(event.getOrderId()));
        } catch (Exception e) {
            eventBus.publish(new StockReservationFailed(event.getOrderId(), e.getMessage()));
        }
    }
}
```

Инициатор саги

```
public class OrderController {
    private final EventBus eventBus;
    public OrderController(EventBus eventBus) { this.eventBus = eventBus; }

    public void placeOrder(OrderData order) {
        eventBus.publish(new OrderPlaced(order.getId(), order.getTotal()));
    }
}
```

Тесты

```
@Test
void testHappyPath() {
    EventBus bus = new SimpleEventBus();
    BillingService billing = new BillingService(bus);
    LogisticsService logistics = new LogisticsService(bus);
    // подписываемся на финальные события для проверки
    List<String> finalEvents = new ArrayList<>();
```

```
        bus.subscribe(StockReserved.class, e -> finalEvents.add("SUCCESS"));
        bus.subscribe(StockReservationFailed.class, e -> finalEvents.add("FAILURE"));

        OrderController controller = new OrderController(bus);
        controller.placeOrder(new OrderData("ORD-1", 100.0));

        // Ждём (при синхронном bus всё выполнится сразу)
        assertEquals(1, finalEvents.size());
        assertEquals("SUCCESS", finalEvents.get(0));
    }

    @Test
    void testCompensation() {
        EventBus bus = new SimpleEventBus();
        BillingService billing = new BillingService(bus);
        LogisticsService logistics = new LogisticsService(bus);

        // форсируем ошибку логистики, например, через подмену логики
        // или просто ставим заглушку. Для теста можно модифицировать класс.
        // Далее проверяем, что произошёл возврат денег
        // ... (демонстрация, что PaymentRefunded было опубликовано)
    }
```

Компромиссы в примере

- **До:** 1 класс OrderProcessor (~20 строк) без компенсаций.
- **После:** 2 сервиса, 5+ событий, Event Bus, инициатор – около 150 строк.
- **Надёжность:** теперь при сбое резервирования деньги возвращаются.
- **Сложность:** необходимо проследить поток событий, чтобы отладить.

Индивидуальные задания

Каждый вариант содержит два или три модуля (условных подсистемы) и бизнес-операцию, затрагивающую их. Требуется реализовать Saga (хореографию) с использованием Event Bus, все шаги и компенсации. Необходимо покрыть успешный и неуспешный сценарии.

Вариант 1. Оформление заказа: шаг 1 – списание средств (биллинг), шаг 2 – резервирование товара (логистика). Компенсация – возврат денег при отказе резервирования.

Вариант 2. Регистрация пользователя: шаг 1 – создание учётной записи в системе аутентификации, шаг 2 – отправка приветственного письма, шаг 3 – начисление бонусов. Компенсация: удаление учётной записи, если письмо не отправлено.

Вариант 3. Бронирование отеля: шаг 1 – блокировка номера (гостиница), шаг 2 – списание предоплаты, шаг 3 – отправка подтверждения. Компенсация: отмена блокировки номера при отказе оплаты.

Вариант 4. Перемещение средств между счетами: шаг 1 – списание с источника, шаг 2 – зачисление на целевой счёт. Компенсация: возврат списания на исходный счёт, если зачисление не удалось.

Вариант 5. Бронирование тура: шаг 1 – резервирование авиабилета, шаг 2 – резервирование отеля, шаг 3 – оплата тура. Компенсация: отмена обоих резервов при отказе оплаты.

Вариант 6. Проведение вебинара: шаг 1 – регистрация участника (CRM), шаг 2 – отправка ссылки на почту, шаг 3 – запись в платформу вебинаров. Компенсация: удаление из CRM, если запись не удалась.

Вариант 7. Заказ такси: шаг 1 – поиск водителя (диспетчер), шаг 2 – блокировка средств (биллинг), шаг 3 – подтверждение поездки. Компенсация: возврат средств и освобождение водителя при отказе.

Вариант 8. Оформление подписки: шаг 1 – создание подписки (биллинг), шаг 2 – активация доступа (сервис доступа), шаг 3 – отправка приветствия. Компенсация: деактивация доступа при отказе биллинга.

Вариант 9. Публикация статьи в блоге: шаг 1 – сохранение статьи (CMS), шаг 2 – отправка уведомлений подписчикам, шаг 3 – индексация в поиск. Компенсация: удаление статьи, если уведомления не отправлены.

Вариант 10. Выдача книги в библиотеке: шаг 1 – проверка наличия книги (каталог), шаг 2 – списание штрафа (биллинг), шаг 3 – регистрация выдачи (электронный читательский билет). Компенсация: возврат штрафа и снятие брони.

Вариант 11. Обработка возврата товара: шаг 1 – одобрение возврата (логистика), шаг 2 – возврат денег на карту (биллинг), шаг 3 – обновление остатков на складе. Компенсация: если возврат денег не удался, отменить возврат и пометить как ошибку.

Вариант 12. Участие в конкурсе: шаг 1 – регистрация заявки (CRM), шаг 2 – зачисление баллов участнику, шаг 3 – отправка подтверждения. Компенсация: отмена заявки и списание баллов.

Вариант 13. Открытие депозита в банке: шаг 1 – создание депозитного счёта, шаг 2 – перевод суммы с текущего счёта, шаг 3 – начисление процентов. Компенсация: возврат суммы на текущий счёт и закрытие депозита.

Вариант 14. Размещение рекламной кампании: шаг 1 – резервирование бюджета (биллинг), шаг 2 – создание кампании в рекламной системе, шаг 3 – запуск модерации. Компенсация: отмена резервирования бюджета, если кампания не создана.

Вариант 15. Заказ еды в приложении: шаг 1 – списание стоимости заказа, шаг 2 – подтверждение рестораном, шаг 3 – передача курьеру. Компенсация: возврат денег и уведомление ресторана об отмене.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.

2. **Цель и задачи работы** (из начала документа).
3. **Теоретическая часть** – описание паттерна Saga (хореография и оркестрация), понятие компенсирующих транзакций, сравнение с двухфазной фиксацией.
4. **Ход выполнения**:
   - Исходный код с прямой связью и отсутствием отката.
   - Определённые события и их роль.
   - Реализация Event Bus (если не из ЛР13, то отдельный класс).
   - Код каждого сервиса-подписчика (листинги).
   - Инициатор саги.
5. **Результаты тестирования** – код тестов для успешного и неуспешного сценариев, скриншоты вывода.
6. **Оценка компромиссов** – анализ сложности, устойчивости, сравнение с альтернативами.
7. **Вывод** – достигнуты ли цели, какие навыки приобретены.
8. **Ответы на контрольные вопросы** (развёрнуто).

Контрольные вопросы

1. В чём разница между хореографией и оркестрацией в Saga?

2. Что такое компенсирующая транзакция? Почему она не должна быть просто отменой по смыслу?

3. Как в вашей реализации гарантируется, что при отказе на втором шаге первый шаг будет скомпенсирован?

4. Какие проблемы возникают при асинхронной обработке событий в Saga (например, потерянные события, дублирование)?

5. Как бы вы реализовали идемпотентность обработчиков для защиты от повторной доставки событий?

6. Почему в данной работе не используется внешний брокер (Kafka, RabbitMQ) и как бы вы переписали код для работы с ним?

7. Как можно отслеживать состояние саги (например, для мониторинга)?

8. Какие типичные анти-паттерны встречаются при реализации Saga (например, длительные блокировки, циклы компенсаций)?

9. Как тестировать сагу, если обработчики асинхронны?

10. Можно ли в одной Saga комбинировать синхронные вызовы и события? Какие риски?

Лабораторная работа №17. Паттерны устойчивости: Circuit Breaker и Retry

Цель работы: очистить и агрегировать код взаимодействия с внешними нестабильными подсистемами путём внедрения паттернов устойчивости Retry (с экспоненциальной задержкой) и Circuit Breaker (с состояниями Closed, Open, Half-Open), обеспечив защиту приложения от каскадных отказов и улучшив пользовательский опыт при временных сбоях.

Задачи:

1. Проанализировать существующий код, который напрямую вызывает внешний нестабильный API (симулируемый) без какой-либо защиты от сбоев.
2. Реализовать паттерн **Retry**: автоматическое повторение неудачных вызовов до заданного количества раз, с экспоненциальной задержкой между попытками.
3. Реализовать паттерн **Circuit Breaker** (Автоматический выключатель) с конечным автоматом состояний: Closed (все вызовы проходят), Open (вызовы немедленно возвращают ошибку, без обращения к API), Half-Open (пробный вызов для проверки восстановления).
4. Настроить параметы Circuit Breaker: порог ошибок для перехода Open → время ожидания в Open → количество успешных вызовов в Half-Open для возврата в Closed.
5. Реализовать имитацию нестабильного внешнего API с тремя режимами: стабильная работа, временный сбой, длительная недоступность.
6. Написать интеграционные тесты, проверяющие:
   - Retry: успешный вызов после нескольких сбоев.
   - Circuit Breaker: переход в Open после превышения порога ошибок, блокировка вызовов, переход в Half-Open, восстановление.
7. Оценить компромиссы: Retry (увеличение времени ответа) vs Circuit Breaker (защита от лавинной нагрузки) и их комбинация.

Теоретическая часть

Проблема взаимодействия с ненадёжными внешними системами

В распределённых системах вызовы внешних API могут завершаться ошибками по разным причинам: временные сбои сети, перегрузка сервера, тайм-ауты. Типичный код:

- Не обрабатывает временные ошибки → мгновенная ошибка пользователю.
- Пытается повторить вызов «вслепую» без пауз → может усугубить перегрузку.
- Не отслеживает состояние внешней системы → продолжает посылать запросы к уже недоступному сервису, тратя ресурсы и увеличивая время отклика.

Паттерн Retry (Повтор с задержкой)

Назначение: автоматически повторяет операцию при обнаружении временного сбоя, используя стратегию задержек, чтобы дать системе восстановиться.

Алгоритм:

- Заданное максимальное число попыток (например, 3).
- Задержка между попытками (например, фиксированная или экспоненциальная: delay = base * 2^(attempt-1) с optional максимальной задержкой).
- Определение, какие исключения считаются «временными» (например, TimeoutException, ServiceUnavailableException).

Компромиссы:

- Увеличивает общее время ответа.
- При неправильной настройке может перегрузить систему при тяжёлых сбоях.
- Не решает проблему длительной недоступности – все попытки всё равно выполняются.

Паттерн Circuit Breaker (Автоматический выключатель)

Назначение: предотвращает выполнение операций, которые, вероятнее всего, завершатся ошибкой, давая системе время на восстановление и предотвращая лавинные отказы.

Конечный автомат состояний:

- **Closed (Замкнут):** все вызовы проходят к внешней системе. Счётчик ошибок инкрементируется при каждой ошибке; если ошибок за период > порога → переход в Open.
- **Open (Разомкнут):** вызовы немедленно возвращают ошибку (без обращения к системе). Через заданный таймаут → переход в Half-Open.
- **Half-Open (Полуоткрыт):** пропускается ограниченное количество пробных вызовов. Если успешны → возврат в Closed. Если ошибка → возврат в Open.

Дополнительно:

- Скользящее окно для подсчёта ошибок.
- Период сброса счётчика в Closed.
- Настройка количества успешных вызовов для выхода из Half-Open.

Компромиссы:

- Защищает систему от каскадного отказа.
- Усложняет логику и тестирование.
- Может ложно переходить в Open при единичных всплесках, если порог низкий.

Комбинация Retry + Circuit Breaker

Лучшая практика: **Retry выполняется внутри Circuit Breaker**. При первом вызове (Closed) выполняются попытки с повторением. Если все попытки не удались, это считается одной неудачей для Circuit Breaker. Таким образом, серия временных сбоев не «выбивает» выключатель, пока они не станут устойчивыми.

Компромиссы

Аспект	Без защиты	Retry	Circuit Breaker	Retry + CB
Устойчивость к кратковременным сбоям	Низкая	Высокая	Средняя (не предназначен)	Высокая
Защита от лавинной нагрузки	Нет	Нет	Да	Да
Увеличение времени ответа при сбое	Нет	Да	Нет (быстрый отказ в Open)	Умеренное
Сложность реализации	Низкая	Средняя	Высокая	Высокая

Trade-off: избыточные Retry могут создать эффект «ударного трафика» при масштабном сбое. Circuit Breaker без Retry плохо справляется с временными проблемами. Их комбинация даёт оптимальную устойчивость, но требует тщательной настройки параметров.

Методика выполнения работы

Шаг 1. Исходный код (проблемный)

Получите вариант – класс сервиса, который напрямую вызывает методы ExternalPaymentGateway (или другого внешнего API) без обработки ошибок. Имитация нестабильного API: методы могут падать с определённой вероятностью или пропорционально числу вызовов.

Шаг 2. Реализация Retry без внешних библиотек

Создайте класс RetryTemplate с параметрами: maxAttempts, backoffInitialDelayMs, backoffMultiplier, retryableExceptions (список классов исключений). Метод execute(Supplier<T> action):

- Цикл от 1 до maxAttempts.

- В каждой итерации пытается выполнить action, если успех → возвращает результат.
- Если поймано исключение и оно подходит под retryable, вычисляется задержка (экспоненциально), ожидание, continue.
- Иначе (неретранслируемое исключение) – проброс.
- После всех попыток – проброс последнего исключения.

Шаг 3. Реализация Circuit Breaker

Создайте класс CircuitBreaker с состояниями (enum: CLOSED, OPEN, HALF_OPEN):

- failureThreshold (например, 3 ошибки/сек).
- timeout в OPEN состоянии (например, 5 секунд).
- halfOpenSuccessThreshold (например, 1 успешный вызов).
- Внутренние счётчики: failureCount, lastOpenTime, successCountInHalfOpen.
- Метод call(Supplier<T> action):
    - В состоянии OPEN: если текущее время > lastOpenTime + timeout → переход в HALF_OPEN. Иначе выбросить CircuitBreakerOpenException.
    - В HALF_OPEN: выполнить действие. Если успех → инкремент successCountInHalfOpen, при достижении порога → переход в CLOSED, сброс счётчиков. При ошибке → переход в OPEN, сброс таймера.
    - В CLOSED: выполнить действие. При успехе → сброс failureCount. При ошибке → инкремент failureCount; если достигнут порог → переход в OPEN.

Шаг 4. Комбинация Retry + Circuit Breaker

Создайте класс ResilientCaller, который принимает CircuitBreaker и RetryTemplate. В методе call:

- Внутри Circuit Breaker.call() используется RetryTemplate для выполнения действия. Таким образом, несколько попыток внутри Closed считаются как одна неудача для Circuit Breaker, если все они завершились ошибкой.

Шаг 5. Имитация нестабильного API

Создайте сервис UnreliablePaymentService с методами:

- charge(double amount) – возвращает Success при вызове, если количество вызовов меньше N; после N начинает падать; или падает с вероятностью P.
- Добавьте возможность программно сбросить состояние для тестов.

Шаг 6. Тестирование

Напишите тесты с использованием JUnit (без необходимости в реальной сети):

- **Retry:** вызов падает 2 раза, затем успех → тест должен завершиться успешно, количество попыток = 3.
- **Circuit Breaker:** 4 последовательных вызова (где 3 падают) → после 3-й ошибки переход в Open; следующий вызов должен сразу вернуть ошибку (без вызова API); после таймаута 1 сек (в тесте ускорить) переход в Half-Open; успешный вызов → возврат в Closed.
- **Комбинация:** Когда внутри Closed происходит 3 попытки Retry, которые все падают → это одна «неудача» для Circuit Breaker. Если задан порог 3 ошибки, то потребуется 3 таких «групп» вызовов для перехода в Open.

Шаг 7. Оценка компромиссов

В отчёте сравните поведение при разных настройках: малое число попыток, короткий таймаут Open, высокий порог ошибок.

Пример выполнения задания

Исходный код (проблемный)

```
public class OrderService {
    private ExternalPaymentGateway gateway = new ExternalPaymentGateway();

    public void payOrder(String orderId, double amount) {
        // Прямой вызов без защиты
        boolean success = gateway.charge(amount);
        if (!success) throw new RuntimeException("Payment failed");
    }
}

// Имитация нестабильного API
public class ExternalPaymentGateway {
    private int callCount = 0;
    public boolean charge(double amount) {
        callCount++;
        // первые 2 вызова успешны, следующие 3 падают, затем снова успех
        if (callCount <= 2) return true;
        if (callCount <= 5) return false;
        return true;
    }
}
```

Реализация Retry Template

```
public class RetryTemplate {
    private final int maxAttempts;
    private final long initialDelayMs;
```

```
    private final double multiplier;
    private final List<Class<? extends Throwable>> retryableExceptions;

    public RetryTemplate(int maxAttempts, long initialDelayMs, double multiplier,
                         List<Class<? extends Throwable>> retryableExceptions) {
        this.maxAttempts = maxAttempts;
        this.initialDelayMs = initialDelayMs;
        this.multiplier = multiplier;
        this.retryableExceptions = retryableExceptions;
    }

    public <T> T execute(Supplier<T> action) {
        long delay = initialDelayMs;
        Throwable lastException = null;
        for (int attempt = 1; attempt <= maxAttempts; attempt++) {
            try {
                return action.get();
            } catch (Throwable ex) {
                lastException = ex;
                if (!isRetryable(ex) || attempt == maxAttempts) {
                    throw new RuntimeException(ex);
                }
                try {
                    Thread.sleep(delay);
                    delay = (long) (delay * multiplier);
                } catch (InterruptedException ignored) {}
            }
        }
        throw new RuntimeException(lastException);
```

```
        }

        private boolean isRetryable(Throwable ex) {
            return retryableExceptions.stream().anyMatch(e -> e.isAssignableFrom(ex.getClass()));
        }
    }
```

Реализация Circuit Breaker

```
    public class CircuitBreaker {
        public enum State { CLOSED, OPEN, HALF_OPEN }

        private State state = State.CLOSED;
        private int failureCount = 0;
        private final int failureThreshold;
        private final long openTimeoutMs;
        private long lastOpenTime = 0;
        private int successCountInHalfOpen = 0;
        private final int successThreshold;

        public CircuitBreaker(int failureThreshold, long openTimeoutMs, int successThreshold) {
            this.failureThreshold = failureThreshold;
            this.openTimeoutMs = openTimeoutMs;
            this.successThreshold = successThreshold;
        }

        public <T> T call(Supplier<T> action) throws Exception {
            if (state == State.OPEN) {
                if (System.currentTimeMillis() - lastOpenTime > openTimeoutMs) {
```

```
            state = State.HALF_OPEN;
            successCountInHalfOpen = 0;
        } else {
            throw new Exception("CircuitBreaker is OPEN");
        }
    }

    try {
        T result = action.get();
        // Успех
        if (state == State.HALF_OPEN) {
            successCountInHalfOpen++;
            if (successCountInHalfOpen >= successThreshold) {
                state = State.CLOSED;
                failureCount = 0;
            }
        } else { // CLOSED
            failureCount = 0;
        }
        return result;
    } catch (Exception ex) {
        if (state == State.HALF_OPEN) {
            // Ошибка в Half-Open -> возвращаемся в OPEN
            state = State.OPEN;
            lastOpenTime = System.currentTimeMillis();
            successCountInHalfOpen = 0;
        } else { // CLOSED
            failureCount++;
            if (failureCount >= failureThreshold) {
                state = State.OPEN;
```

```
                lastOpenTime = System.currentTimeMillis();
            }
        }
        throw ex;
    }
  }
}
```

Комбинация

```
public class ResilientPaymentService {
  private final ExternalPaymentGateway gateway;
  private final RetryTemplate retry;
  private final CircuitBreaker breaker;

  public ResilientPaymentService(ExternalPaymentGateway gateway,
                                 RetryTemplate retry,
                                 CircuitBreaker breaker) {
    this.gateway = gateway;
    this.retry = retry;
    this.breaker = breaker;
  }

  public void charge(double amount) throws Exception {
    breaker.call(() -> retry.execute(() -> {
      if (!gateway.charge(amount)) throw new RuntimeException("Payment rejected");
      return true;
    }));
  }
}
```

Тесты

```
@Test
void testRetrySuccess() {
   UnreliableGateway gw = new UnreliableGateway(); // падает первые 2 раза
   RetryTemplate retry = new RetryTemplate(3, 10, 1.5,
      List.of(RuntimeException.class));
   // должен выполниться с 3-й попытки
   Boolean result = retry.execute(() -> gw.charge(100));
   assertTrue(result);
}

@Test
void testCircuitBreakerTrips() throws Exception {
   UnreliableGateway gw = new UnreliableGateway();
   // configure it to fail always after some calls
   CircuitBreaker cb = new CircuitBreaker(3, 1000, 1);
   RetryTemplate retry = new RetryTemplate(1, 0, 1, List.of()); // без повтора
   ResilientPaymentService service = new ResilientPaymentService(gw, retry, cb);

   // 3 вызова, каждый упадёт -> CB должен перейти в OPEN
   for (int i = 0; i < 3; i++) {
      try { service.charge(100); } catch (Exception ignored) {}
   }
   // следующий вызов должен выбросить CircuitBreakerOpenException
   assertThrows(Exception.class, () -> service.charge(100));
}
```

Компромиссы в примере

- **Добавлено:** 3 класса (Retry, CircuitBreaker, ResilientService) + тесты.
- **Улучшения:** защита от временных сбоев (Retry) и длительных сбоев (Circuit Breaker).
- **Цена:** увеличение задержки при повторных попытках, сложность тестирования конечного автомата.

Индивидуальные задания

Каждый вариант – внешний нестабильный сервис (имитация) и бизнес-логика, его вызывающая. Требуется реализовать Retry и Circuit Breaker, протестировать сценарии восстановления.

Вариант 1. PaymentGateway с вероятностью сбоя 30%, после 5 подряд сбоев переходит в критическое состояние. Retry 3 раза, задержка 100 мс (x2). Circuit Breaker: порог 3 ошибки, таймаут 2 с.

Вариант 2. InventoryService тайм-аут 50% запросов. Retry 4 раза, задержка 50 мс (x1.5). CB: порог 2 ошибки, таймаут 1 с.

Вариант 3. NotificationService падает с TooManyRequestsException при первой попытке, затем работает. Retry 2 раза, задержка 200 мс (фиксированная). CB: порог 3, таймаут 5 с.

Вариант 4. CurrencyConverter временно недоступен (ошибка 503) первые 2 секунды. Retry 5 раз, задержка 500 мс экспоненциально. CB: порог 4, таймаут 3 с.

Вариант 5. SmsProvider выдаёт 50% сбой с NetworkException, остальное – успех. Retry 3 раза, задержка 100 мс (x2). CB: порог 5, таймаут 10 с.

Вариант 6. FileStorageService: первые 3 вызова падают с IOException, затем работает стабильно. Retry 4 раза, задержка 50 мс (x1.5). CB: порог 2, таймаут 1 с.

Вариант 7. EmailService иногда отвечает InternalServerError (вероятность 40%). Retry 3 раза, экспонента 100мс. CB порог 3, таймаут 2 с.

Вариант 8. FraudDetectionService медленный: тайм-аут после 500 мс, но успевает только в 30% случаев. Retry 2 раза, задержка 200 мс. CB порог 25, таймаут 5 с.

Вариант 9. GeocodingApi стабилен, но иногда возвращает пустой ответ (считать ошибкой). Retry 4 раза (задержка 50мс). CB порог 10, таймаут 30 с.

Вариант 10. StockQuoteService: первые 4 вызова падают с TimeoutException, затем работает. Retry 5 раз, задержка 100мс *2. CB порог 5, таймаут 5 с.

Вариант 11. CrmClient работает только в 20% случаев, остальные ConnectionException. Retry 3 раза, от 100мс экспоненциально. CB порог 4, таймаут 3 с.

Вариант 12. TranslationApi возвращает ошибку TooManyRequests после 2 успешных вызовов, затем снова работает. Retry 3 раза, фикс. задержка 500мс. CB порог 2, таймаут 10 с.

Вариант 13. CloudStorage вызовы завершаются успешно в 1 раз из 4. Retry 3, задержка 200 мс *1.5. CB порог 5, таймаут 4 с.

Вариант 14. WeatherService недоступен в течение первых 3 секунд, затем стабилен. Retry 5 раз (экспонента от 100мс). CB порог 3, таймаут 3 с.

Вариант 15. Сервис токенов выдаёт ошибку аутентификации при каждом 3-м вызове. Retry 2 раза, задержка 50 мс. CB порог 4, таймаут 1 с.

Требования к отчёту

Отчёт должен содержать:

1. **Титульный лист** с темой работы, ФИО, вариантом.
2. **Цель и задачи работы**.
3. **Теоретическая часть** – описание паттернов Retry (с экспоненциальной задержкой) и Circuit Breaker (состояния, переходы), их цели и компромиссы.
4. **Ход выполнения**:
   - Исходный код без защиты.
   - Реализация RetryTemplate (листинг).
   - Реализация Circuit Breaker (листинг).
   - Комбинация в сервисе (листинг).
   - Имитация нестабильного API.
5. **Результаты тестирования** – код тестов для каждого сценария (Retry успех, CB переходы, комбинация) и скриншоты выполнения.
6. **Оценка компромиссов** – таблица сравнения с альтернативами, обсуждение подходящих настроек для своего варианта.
7. **Вывод** – достигнуты ли цели, навыки устойчивости систем.
8. **Ответы на контрольные вопросы**.

Контрольные вопросы

1. В чём разница между временным сбоем (например, сетевая проблема) и неисправимым сбоем (например, неверные данные)? Как к ним относятся Retry и Circuit Breaker?
2. Почему экспоненциальная задержка предпочтительнее фиксированной в Retry?
3. Что происходит в состоянии Half-Open? Зачем нужно несколько успешных вызовов для возврата в Closed?
4. Как можно избежать бесконечного циклирования (Circuit Breaker Open -> Half-Open -> Open при постоянном сбое)?
5. Как в вашей реализации гарантируется потокобезопасность Circuit Breaker? Какие проблемы при многопоточном доступе?
6. Как тестировать Circuit Breaker без реальных пауз (таймаутов)? Какие техники ускорения времени использовали?
7. Почему не рекомендуют использовать Retry для всех типов исключений (например, NullPointerException)?
8. Какой сценарий использования Circuit Breaker в монолитном приложении без микросервисов? (например, при вызове внешних API)
9. Какие метрики нужно собирать для мониторинга работы Circuit Breaker в production?
10. Как комбинирование Retry и Circuit Breaker защищает от «лавинной перегрузки» при восстановлении внешнего сервиса?

Лабораторная работа №18. Итоговый проект: рефакторинг полного приложения с композицией паттернов

Цель работы: очистить и агрегировать знания и навыки, полученные в ходе курса, путём комплексного рефакторинга legacy-приложения с применением не менее шести изученных паттернов проектирования, обосновав выбор каждого паттерна архитектурными компромиссами и защитив итоговое решение.

Задачи:

1. Проанализировать предоставленный legacy-код (монолитное приложение с типичными анти-паттернами: God Object, жесткие зависимости, смешение бизнес-логики с инфраструктурой, отсутствие тестов).
2. Определить ключевые архитектурные проблемы и сформулировать требования к целевой архитектуре.
3. Спроектировать целевую архитектуру с использованием архитектурного каркаса (например, Clean Architecture / Ports & Adapters).
4. Выбрать и обоснованно применить не менее шести паттернов проектирования (GoF, архитектурные, интеграционные, тестируемости) для решения выявленных проблем.
5. Реализовать рефакторинг с сохранением функциональности исходного приложения.
6. Покрыть критически важные компоненты модульными и интеграционными тестами (с использованием Test Doubles).
7. Подготовить защиту проекта: презентация архитектурных решений, демонстрация кода, количественная и качественная оценка trade-offs.
8. Написать итоговый отчёт, включающий диаграммы классов/последовательности, список применённых паттернов с обоснованием, сравнение метрик «до/после».

Теоретическая часть

Композиция паттернов в реальных системах

В промышленной разработке редко используется один паттерн изолированно. Архитектура средних и крупных систем строится как комбинация паттернов разных уровней:

- **Архитектурные паттерны** (Layered, Ports & Adapters, CQRS) задают общую структуру.
- **Паттерны GoF** (Factory, Command, Observer, Strategy, Composite, Decorator и др.) решают локальные задачи внутри слоёв.
- **Интеграционные паттерны** (Circuit Breaker, Retry, Saga) обеспечивают устойчивость и согласованность.
- **Паттерны тестируемости** (Test Double, Object Mother, Builder) делают код проверяемым.

Типичные проблемы legacy-кода и способы их решения

Проблема	Признаки	Применяемые паттерны
Жёсткие зависимости	new внутри методов, статические вызовы	Dependency Injection, Factory, Abstract Factory
God Object	Класс с 20+ полями и 30+ методами	Facade, Strategy, разделение на сервисы
Труднотестируемость	Логика смешана с БД/сетью	Ports & Adapters, Test Doubles
Отсутствие транзакционности	При ошибке данные разъезжаются	Unit of Work, Saga (хореография)
Сложные запросы	JOIN-ы, дублирование кода запросов	Repository, CQRS
Потеря истории изменений	Нет аудита, невозможно откатить	Event Sourcing, Domain Events

Стратегия рефакторинга к паттернам

1. **Выделение сущностей и Value Objects** (отделить бизнес-логику от инфраструктуры).
2. **Определение портов (интерфейсов)** для зависимостей (репозитории, внешние API).
3. **Перенос бизнес-логики в Use Cases (Interactors)** с использованием Command или простых методов.
4. **Реализация адаптеров** для конкретных технологий (файлы, БД, API).
5. **Введение событий и Event Bus** для слабой связанности.
6. **Добавление механизмов устойчивости** (Retry, Circuit Breaker) на границах.
7. **Написание тестов** каждого слоя изолированно.

Компромиссы при композиции паттернов

Паттерн	Когда оправдан	Издержки
Clean Architecture	Долгоживущий проект, частые изменения технологий	Много классов, порог входа
CQRS	Разные нагрузки на чтение и запись	Две модели, синхронизация
Event Sourcing	Требуется полный аудит, откат состояний	Огромное хранилище событий
Saga	Долгие бизнес-процессы, требующие компенсаций	Сложность отладки
Circuit Breaker	Вызовы внешних нестабильных сервисов	Задержки, параметризация
Observer/Event Bus	Много побочных реакций на события	Неявный поток управления

Ключевой trade-off: гибкость и устойчивость достигаются за счёт увеличения количества абстракций, классов и косвенных вызовов. Простое CRUD-приложение не требует большинства паттернов. Но для систем

масштаба, которые будут расти и эволюционировать, композиция паттернов – это инвестиция в будущее.

Методика выполнения работы

Шаг 1. Получение и анализ legacy-кода

Индивидуальный вариант – монолитное приложение (например, система управления заказами, библиотекой, складом). Код содержит минимум 3 из следующих проблем:

- God Object (один класс делает всё).
- Жёсткие зависимости от конкретных реализаций БД или API.
- Отсутствие тестов или тесты с реальными БД.
- Повторяющиеся фрагменты логики.
- Отсутствие обработки ошибок и компенсаций.
- Сложные запросы, смешанные с бизнес-правилами.

Задание: составить карту проблем (таблицу), выделить кандидатов на паттерны.

Шаг 2. Выбор целевой архитектуры и паттернов

Необходимо выбрать архитектурный каркас. Рекомендуется **Ports & Adapters (Hexagonal)** как наиболее гибкий. На его основе определить, какие паттерны будут использованы (минимум 6 из изученных).

Пример комбинации:

- Архитектурные: Ports & Adapters, CQRS.
- GoF: Factory Method (для создания репозиториев), Command (для бизнес-операций), Observer/Event Bus (для нотификаций).
- Интеграционные: Circuit Breaker (для внешних API).
- Тестируемости: Test Data Builder, Object Mother.

Шаг 3. Проектирование (диаграммы)

Нарисовать (в отчёте):

- Контекстную диаграмму (модули, порты, адаптеры).
- Диаграмму классов целевой архитектуры.
- Диаграмму последовательности для одного сценария (например, создание заказа с событиями).

Шаг 4. Поэтапный рефакторинг

Рекомендуемый порядок (можно выполнять в любой итерации):

1. **Выделение сущностей** из God Object.
2. **Определение портов** (репозитории, внешние сервисы).
3. **Реализация Use Cases** с использованием Command (или простых классов).
4. **Создание адаптеров** для БД (InMemory для тестов) и внешних API (заглушки).
5. **Внедрение Event Bus и Domain Events** для отделения побочных эффектов.
6. **Добавление Circuit Breaker** для внешних вызовов.
7. **Настройка ручного DI** (или простого контейнера).
8. **Написание тестов** для каждого Use Case и адаптера.

Шаг 5. Тестирование

Покрыть тестами:

- **Модульные тесты** для Use Cases с моками репозиториев.
- **Интеграционные тесты** для адаптеров (например, InMemoryRepository).
- **Тесты производительности** для сравнения производительности (опционально).

Шаг 6. Защита проекта

Подготовить презентацию (10–15 минут):

- Проблемы исходного кода (с примерами).

- Целевая архитектура и выбранные паттерны (почему именно они).
- Демонстрация ключевых фрагментов кода.
- Сравнение метрик (количество классов, строк кода, время выполнения тестов).
- Анализ компромиссов (что выиграли, чем пожертвовали).

Пример выполнения задания (упрощённый)

Исходный код (проблемный)

```
// God Object: OrderManager
public class OrderManager {
    private Connection db;
    private EmailSender email = new EmailSender();

    public void createOrder(Customer customer, List<Item> items) {
        // прямые SQL
        double total = items.stream().mapToDouble(Item::getPrice).sum();
        try (PreparedStatement stmt = db.prepareStatement("INSERT INTO orders...")) {
            stmt.setString(1, customer.getId());
            stmt.setDouble(2, total);
            stmt.executeUpdate();
        } catch (SQLException e) { ... }
        email.send(customer.getEmail(), "Order created");
    }

    public List<OrderReport> generateReport(Date start, Date end) {
        // сложный SQL-запрос с JOIN
        // 30 строк кода
    }
}
```

Целевая архитектура (Clean + CQRS + Event Bus + Circuit Breaker)

Применённые паттерны (7):

1. **Ports & Adapters** – разделение на порты (UseCase, Repository) и адаптеры.
2. **CQRS** – Command side: CreateOrderCommand, Query side: OrderReportQuery.
3. **Command** – инкапсуляция операции создания заказа.
4. **Domain Events + Event Bus** – публикация OrderCreatedEvent, подписчики: EmailNotifier, StatisticsUpdater.
5. **Repository** – интерфейс OrderRepository, адаптер JdbcOrderRepository.
6. **Circuit Breaker** – для внешнего email-сервиса (эмуляция сбоев).
7. **Test Data Builder** – для создания тестовых заказов.

Фрагмент реализации

```
// Command
public class CreateOrderCommand {
    private final OrderRepository orderRepo;
    private final EventBus eventBus;

    public void execute(CreateOrderRequest req) {
        Order order = new Order(req.getCustomerId(), req.getItems());
        orderRepo.save(order);
        eventBus.publish(new OrderCreatedEvent(order.getId(),
order.getTotal()));
    }
}

// Подписчик (Observer)
public class EmailNotifier implements EventHandler<OrderCreatedEvent> {
    private final CircuitBreaker emailBreaker;
    public void handle(OrderCreatedEvent event) {
```

```
            emailBreaker.call(() -> emailService.send(event.getCustomerEmail()));
        }
    }
```

Сравнение метрик

Метрика	До	После
Количество классов	3	15
Строк кода (основная логика)	150	250
Строк кода тестов	0	200
Время выполнения запроса отчёта (сложный запрос)	200 мс	15 мс (через CQRS)
Покрытие тестами	0%	85%
Затраты на поддержку новой фичи	1 день	2 часа (из-за тестов)

Компромиссы

- **Выиграли:** тестируемость, производительность сложных запросов, расширяемость, устойчивость к сбоям email.
- **Потеряли:** простота кода (больше классов), начальное время разработки выросло.

Индивидуальные задания

Каждый вариант – legacy-приложение с типичными проблемами. Требуется провести полный рефакторинг, применив минимум 6 паттернов (список указать в отчёте). Предусмотреть тесты, защиту.

Вариант 1. Система управления библиотекой: выдача и возврат книг, поиск по каталогу, отчёты по должникам. Код: один класс Library с 30+ методами, прямые SQL-запросы, отсутствие транзакций при выдаче/возврате.

Вариант 2. Интернет-магазин: корзина, оформление заказа, оплата через внешний шлюз (имитация). Проблемы: смешение логики, жёсткая привязка к шлюзу, нет тестов.

Вариант 3. Система бронирования билетов: поиск рейсов, бронирование, отмена, оплата. Legacy: God Object BookingEngine, копипаста, сложные SQL-запросы.

Вариант 4. CRM для небольшой компании: управление клиентами, заказами, историей взаимодействий. Проблемы: один класс CrmService на 2000 строк, статические методы, прямые вызовы JDBC.

Вариант 5. Система учёта рабочего времени: табели, расчёт зарплаты, отчёты. Код: смесь расчётов и SQL, длинные методы, отсутствие тестов.

Вариант 6. Система лояльности: начисление бонусов, списание, расчёт скидок. Проблемы: жёсткие зависимости от внешнего API (курсы валют), нет обработки ошибок.

Вариант 7. Управление задачами (аналог Trello): доски, колонки, задачи, перемещение. Legacy: один класс BoardManager, всё в одном файле, тесты отсутствуют.

Вариант 8. Система опросов: создание опросов, голосование, подсчёт результатов. Проблемы: смесь бизнес-логики и отображения, прямой доступ к БД в методах, дублирование кода.

Вариант 9. Онлайн-курсы: регистрация студентов, запись на курсы, выставление оценок. Legacy: God Object CourseService, длинные методы с if-else.

Вариант 10. Служба доставки: приём заказов, назначение курьера, трекинг. Проблемы: нет транзакционности при назначении, сложные JOIN-запросы.

Вариант 11. Финансовый портал: учёт доходов/расходов, категории, отчёты. Код: смесь логики и UI, нет разделения на слои.

Вариант 12. Медицинская карта: запись к врачу, история посещений, рецепты. Проблемы: жёсткая связь с базой данных, отсутствие событийной модели.

Вариант 13. Система голосования в компании: создание голосований, подсчёт, экспорт результатов. Legacy: всё в одном классе, нет тестов, при изменении схемы голосования – правка многих мест.

Вариант 14. Управление складом: приход/расход товаров, инвентаризация. Проблемы: отсутствие атомарности операций, сложные отчёты смешаны с логикой.

Вариант 15. Социальная сеть (упрощённо): посты, комментарии, лайки. Код: SocialNetwork с 40 методами, дублирование запросов, нет тестов.

Требования к отчёту (итоговый проект)

Отчёт должен содержать:

1. **Титульный лист** с темой, ФИО, вариантом.
2. **Цель и задачи работы**.
3. **Теоретическая часть** – обзор композиции паттернов, стратегия рефакторинга.
4. **Анализ исходного кода**:
   - Список выявленных анти-паттернов и проблем.
   - Таблица «проблема → кандидатный паттерн».
5. **Целевая архитектура**:
   - Обоснование выбора архитектурного каркаса.
   - Список применённых паттернов (минимум 6) с кратким обоснованием.
   - Диаграмма компонентов (или классов).
6. **Ход рефакторинга**:
   - Ключевые этапы (выделение сущностей, портов, use cases, событий и т.д.).

- Важные фрагменты кода (листинги).

7. **Тестирование**:
   - Перечень тестов (модульные, интеграционные).
   - Пример кода теста с Test Double.
   - Отчёт о покрытии (при возможности).
8. **Оценка компромиссов**:
   - Таблица метрик «было/стало».
   - Обсуждение trade-offs: увеличение количества классов vs гибкость, производительность vs поддерживаемость.
9. **Выводы** – достигнуты ли цели, чему научились.
10. **Ответы на контрольные вопросы** (письменно).

Контрольные вопросы

1. Какие архитектурные паттерны вы использовали и почему именно их?
2. Назовите все применённые паттерны GoF и объясните, какие проблемы они решили.
3. Как вы обеспечили тестируемость после рефакторинга? Какие Test Doubles использовали?
4. Какие компромиссы пришлось принять при внедрении CQRS/Event Sourcing (если использовали)?
5. Как изменилась производительность системы после рефакторинга? Приведите цифры.
6. Какой паттерн оказался самым сложным в реализации и почему?
7. Какие анти-паттерны были устранены? Приведите пример до и после.
8. Как вы управляли транзакционностью при наличии нескольких Use Cases?
9. Что бы вы сделали иначе, если бы проект был вдвое больше?
10. Как предложенная архитектура поможет в будущем переходе на микросервисы (если это потребуется)?

СПСИОК ЛИТЕРАТУРЫ

Основная литература

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2020. — 368 с. (GoF)

2. Фаулер М. Архитектура корпоративных программных приложений. — М.: Вильямс, 2022. — 544 с.

3. Фаулер М. Рефакторинг. Улучшение существующего кода. — СПб.: Питер, 2019. — 432 с.

4. Мартин Р. С. Чистая архитектура. Искусство разработки программного обеспечения. — СПб.: Питер, 2021. — 352 с.

5. Мартин Р. С. Чистый код. Создание, анализ и рефакторинг. — СПб.: Питер, 2020. — 464 с.

6. Фримен Э., Робсон Э., Бейтс Б., Сьерра К. Паттерны проектирования. — СПб.: Питер, 2021. — 656 с. (Head First)

7. Нокс М., Браун С., Лаговски П., Лепка Р. Разработка микросервисов. Паттерны и примеры. — М.: ДМК Пресс, 2021. — 450 с.

Дополнительная литература

8. Вернон В. Реализация стратегий предметно-ориентированного проектирования. — М.: Вильямс, 2021. — 688 с.

9. Шарда А., Захариев З. Event Sourcing и CQRS на практике. — М.: ДМК Пресс, 2023. — 320 с.

10. Ньюмен С. Создание микросервисов. — СПб.: Питер, 2022. — 304 с.

11. Хорстманн К. С. Java. Библиотека профессионала. Том 1. Основы. — М.: Вильямс, 2022. — 928 с. (главы о паттернах и лямбда-выражениях)

12. Мартин Р. С. Гибкая разработка программного обеспечения: принципы, паттерны и практики. — М.: Вильямс, 2021. — 768 с.

13. Фримен С., Прайс Н. Mockito для профессионалов. — М.: ДМК Пресс, 2020. — 256 с.

Интернет-ресурсы

14. Refactoring.Guru — каталог паттернов с примерами на Java. URL: https://refactoring.guru/ru/design-patterns

15. Martin Fowler's Blog — статьи о CQRS, Event Sourcing, Saga, Circuit Breaker. URL: https://martinfowler.com/

16. Java Design Patterns — коллекция паттернов с открытым исходным кодом. URL: https://java-design-patterns.com/

17. Microsoft Cloud Design Patterns — описание паттернов устойчивости и распределённых систем. URL: https://learn.microsoft.com/en-us/azure/architecture/patterns/

18. Baeldung Design Patterns Series — примеры реализации паттернов на Java. URL: https://www.baeldung.com/design-patterns-series