ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном программировании всё большую актуальность приобретает использование шаблонов проектирования (паттернов) — универсальных решений, проверенных практикой, для типовых задач разработки программного обеспечения. Применение паттернов позволяет значительно повысить гибкость, расширяемость и сопровождаемость кода, а также облегчить взаимодействие между компонентами системы.

Ключевая идея паттернов проектирования заключается в повторном использовании архитектурных решений, которые уже доказали свою эффективность. Они помогают разработчикам не «изобретать велосипед», а использовать структурированные подходы к решению задач, таких как управление состояниями объектов, взаимодействие между компонентами, обработка событий и делегирование обязанностей.

Цель данной курсовой работы — продемонстрировать практическое применение поведенческих и структурных паттернов проектирования в программной системе, а также проанализировать их влияние на архитектуру и поведение приложения.

Курсовая работа включает две лабораторные работы, в рамках которых были спроектированы и реализованы программные решения, основанные на различных шаблонах проектирования. В заключении приводится сравнительный анализ подходов и формулируются выводы по работе.

1 ОБЗОР ПАТТЕРНОВ ПРОЕКТИРОВАНИЯ

1.1 Классификация шаблонов проектирования

Шаблоны проектирования представляют собой описания типовых архитектурных решений, которые могут быть повторно использованы при проектировании программного обеспечения. Они помогают решать часто возникающие задачи проектирования и улучшать структуру и читаемость кода.

Наиболее известная и широко используемая классификация шаблонов проектирования была предложена в книге «Design Patterns: Elements of Reusable Object-Oriented Software» авторов Э. Гаммы, Р. Хелма, Р. Джонсона и Дж. Влиссидеса (так называемая «банда четырех»). В соответствии с этой классификацией все шаблоны делятся на три основные группы:

- порождающие шаблоны (Creational Patterns) — отвечают за процесс создания объектов, делая его более гибким и независимым от конкретных классов создаваемых объектов. К ним относятся: Factory Method, Abstract Factory, Builder, Prototype, Singleton;
- структурные шаблоны (Structural Patterns) — определяют способы построения связей между классами и объектами, позволяя формировать более крупные структуры. К ним относятся: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy (Заместитель);
- поведенческие шаблоны (Behavioral Patterns) — описывают способы взаимодействия между объектами, а также распределение обязанностей между ними. В эту группу входят: Chain of Responsibility (Цепочка обязанностей), Command, Interpreter, Iterator, Mediator (Посредник), Memento, Observer (Наблюдатель), State (Состояние), Strategy, Template Method, Visitor.

Каждая из групп шаблонов ориентирована на определенный аспект проектирования и применяется в зависимости от задач и требований разрабатываемой системы. В рамках данной курсовой работы основное внимание уделяется поведенческим и структурным шаблонам, поскольку они

наиболее применимы при реализации логики взаимодействия объектов и построении архитектурных компонентов.

1.2 Поведенческие паттерны проектирования

Поведенческие паттерны проектирования определяют способы взаимодействия между объектами в системе, распределяют обязанности между ними и способствуют упрощению коммуникации и управления поведением программных компонентов. Эти шаблоны особенно важны в случае сложных сценариев взаимодействия, где требуется обеспечить слабую связанность объектов и гибкость архитектуры.

К числу наиболее распространённых поведенческих паттернов относятся:

- observer (Наблюдатель) — устанавливает зависимость «один ко многим» между объектами, так что при изменении состояния одного объекта все зависящие от него оповещаются и обновляются автоматически. Часто используется в системах с графическим интерфейсом и в реализации событийных моделей;
- mediator (Посредник) — инкапсулирует способ взаимодействия множества объектов, предотвращая их прямые ссылки друг на друга. Это упрощает связи между компонентами и повышает модульность системы;
- state (Состояние) — позволяет объекту изменять своё поведение в зависимости от своего внутреннего состояния. Внешне объект может менять свой класс поведения на лету, что делает поведение более управляемым;
- chain of Responsibility (Цепочка обязанностей) — позволяет избежать жёсткой привязки отправителя запроса к получателю, передавая запрос по цепочке потенциальных обработчиков до тех пор, пока один из них не обработает его.

Каждый из этих паттернов решает специфическую задачу, связанную с логикой поведения компонентов, и может быть использован как по отдельности, так и в сочетании с другими шаблонами. Например, шаблон «Цепочка

обязанностей» может использоваться совместно с «Состоянием» для построения адаптивной логики обработки пользовательских запросов, а «Наблюдатель» — в связке с «Посредником» для реализации событийной системы с минимальной связанностью компонентов.

1.3 Структурные паттерны проектирования

Структурные паттерны проектирования описывают способы построения связей между классами и объектами, которые позволяют формировать гибкие и масштабируемые архитектуры. Эти шаблоны упрощают проектирование систем, облегчая расширение, изменение и повторное использование компонентов без изменения их исходного кода.

Цель структурных паттернов — упорядочить отношения между сущностями в системе, обеспечивая прозрачность, переиспользуемость и ослабленную связанность.

Наиболее известные структурные паттерны:

- adapter (Адаптер) — позволяет объектам с несовместимыми интерфейсами работать вместе, преобразуя интерфейс одного класса в интерфейс, ожидаемый клиентом;
- decorator (Декоратор) — динамически добавляет объекту новые обязанности, оборачивая его в другой объект. Это позволяет модифицировать поведение объектов без изменения их кода;
- composite (Компоновщик) — объединяет объекты в древовидную структуру для представления иерархий «часть-целое», позволяя клиентам работать с отдельными объектами и их композициями единообразно;
- proxy (Заместитель) — предоставляет суррогат или заместителя другому объекту для контроля доступа к нему, добавления логики к существующим методам или оптимизации работы (например, ленивой инициализации или кэширования);
- facade (Фасад) — предоставляет упрощённый интерфейс к сложной подсистеме, скрывая её внутреннюю реализацию.

Применение структурных паттернов особенно важно в проектах, где необходимо обеспечить модульность, инкапсуляцию и возможность дальнейшего расширения без изменения уже работающего кода.

1.4 Применение паттернов в промышленной разработке

Паттерны проектирования находят широкое применение в промышленной разработке программного обеспечения, поскольку они способствуют созданию кода, обладающего высокой степенью повторного использования, сопровождаемости и масштабируемости. Использование проверенных архитектурных решений позволяет разработчикам избегать распространённых ошибок, упрощать коммуникацию в команде и ускорять процесс разработки.

В индустриальных проектах паттерны применяются на всех этапах жизненного цикла ПО:

- на этапе проектирования — паттерны помогают формализовать архитектурные решения. Например, выбор между стратегией (Strategy) и состоянием (State) влияет на структуру классов и поведение системы;
- на этапе реализации — паттерны позволяют переиспользовать логические конструкции. Так, шаблон наблюдатель (Observer) используется для построения событийных систем, а посредник (Mediator) — для координации взаимодействий между модулями;
- на этапе тестирования и сопровождения — применение паттернов делает код более модульным и удобным для тестирования. Например, шаблон заместитель (Proxy) может использоваться для создания заглушек и мок-объектов при юнит-тестировании.

Многие современные фреймворки и библиотеки изначально построены на основе паттернов. В частности:

- в Spring Framework (Java) активно используются паттерны Proxy, Singleton, Factory Method, Observer;
- в .NET паттерны Facade, Decorator и Command встроены в основу библиотек Windows Presentation Foundation (WPF);

– в React/Redux архитектуре используются идеи, схожие с Observer, Command и Mediator.

Кроме того, паттерны широко применяются при проектировании микросервисной архитектуры. Например, для организации взаимодействия сервисов через очередь сообщений может использоваться шаблон посредник, а для централизованного контроля — фасад или прокси.

Таким образом, знание и понимание паттернов проектирования является важной частью профессиональных компетенций программиста. Это позволяет эффективно взаимодействовать с существующими системами, разрабатывать масштабируемые архитектуры и обеспечивать высокое качество программного обеспечения.

2 ЛАБОРАТОРНАЯ РАБОТА №1. ШАБЛОН «АГРЕГАТОР СОБЫТИЙ»

2.1 Постановка задачи

Реализуйте шаблон «агрегатор событий» на базе паттернов проектирования «посредник» и «наблюдатель». Примените шаблон на примере отслеживания изменений данных в базе данных.

2.2 Использование паттерна «Посредник»

Паттерн «Посредник» (Mediator) относится к поведенческим шаблонам проектирования и применяется для упрощения взаимодействия между множеством объектов, обеспечивая их слабую связанность. Вместо того чтобы объекты напрямую ссылались друг на друга и обменивались сообщениями, они делегируют эту задачу посреднику, который управляет коммуникацией между ними.

В рамках лабораторной работы №1 задача состояла в реализации агрегатора событий, который отслеживает изменения, происходящие в базе данных, и уведомляет заинтересованные компоненты. Для решения этой задачи применён паттерн «Посредник», который выступает в роли центрального координирующего звена между источниками событий и подписчиками.

Роль посредника в системе:

- регистрация компонентов — посредник позволяет добавлять и удалять участников взаимодействия (наблюдателей);
- реакция на события — при возникновении события (например, изменение данных), источник уведомляет посредника, а тот, в свою очередь, рассылает уведомления всем подписанным объектам;
- минимизация связности — источники и наблюдатели не знают друг о друге напрямую, что упрощает расширение и модификацию системы.

Преимущества применения посредника в лабораторной работе:

- упрощена архитектура системы за счёт удаления прямых связей между объектами;

– повышена модульность — наблюдателей можно легко добавлять или удалять без изменения логики генерации событий;
– обеспечено централизованное управление событиями, что удобно для отладки и масштабирования.

В реализации лабораторной работы посредник представлен отдельным классом, реализующим интерфейс EventMediator, через который проходит вся маршрутизация событий. Источники событий (например, модули, симулирующие изменение записей в базе) уведомляют посредника, который затем вызывает методы соответствующих обработчиков (наблюдателей), реализующих интерфейс Observer.

Такой подход соответствует принципам слабой связанности и инверсии зависимостей, что делает его актуальным не только в учебных, но и в промышленных системах — особенно в контексте событийных шин, pub-sub систем и GUI-библиотек.

2.3 Использование паттерна «Наблюдатель»

Паттерн «Наблюдатель» (Observer) — один из ключевых поведенческих шаблонов проектирования, предназначенный для организации механизма подписки на события. Он обеспечивает автоматическое уведомление зависимых объектов (наблюдателей) об изменениях состояния другого объекта (наблюдаемого), без необходимости тесной связи между ними.

В лабораторной работе №1 паттерн «Наблюдатель» применяется совместно с паттерном «Посредник» для реализации агрегатора событий. Основная задача — создать систему, в которой множество компонентов (наблюдателей) могут отслеживать изменения, происходящие в источниках данных (наблюдаемых объектах), и реагировать соответствующим образом.

Основные элементы реализации:

– наблюдаемый объект (Subject) — компонент, в котором происходят события. В лабораторной работе это, например, имитатор изменений в базе данных;

- наблюдатели (Observers) — компоненты, подписывающиеся на события и реализующие определённую реакцию при их наступлении (например, вывод информации в лог, уведомление пользователя, отправка сообщения в топик Kafka и т.п.);
- интерфейс уведомления — определяет метод, который вызывается при возникновении события (например, onEvent()).

Принцип работы:

1. наблюдатели регистрируются у посредника или напрямую у наблюдаемого объекта;
2. при изменении состояния наблюдаемый объект (или посредник) вызывает метод уведомления о событии у каждого зарегистрированного наблюдателя, который подписан на данное событие;
3. каждый наблюдатель самостоятельно реализует логику реакции на событие.

Преимущества применения:

- слабая связанность компонентов — наблюдаемый объект ничего не знает о деталях реализации наблюдателей;
- гибкость — можно добавлять, удалять или изменять поведение наблюдателей без изменения логики генератора событий;
- расширяемость — система легко масштабируется при увеличении числа участников.

В лабораторной реализации наблюдатели могли выполнять различные действия: логирование событий, обновление пользовательского интерфейса, выполнение дополнительных проверок. Благодаря паттерну «Наблюдатель» вся эта логика была организована в виде независимых модулей, что повысило читаемость и удобство сопровождения кода.

Паттерн «Наблюдатель» особенно актуален в асинхронных и событийно-ориентированных системах, таких как GUI, системные уведомления, и взаимодействие между микросервисами. Его использование в учебной задаче

позволило продемонстрировать принципы реактивного программирования и событийной архитектуры.

2.4 Использование паттерна «Агрегатор событий»

Паттерн «Агрегатор событий» (Event Aggregator) не входит в классический перечень шаблонов «банды четырёх», но широко применяется на практике, особенно в системах с событийной архитектурой. Он представляет собой централизованный механизм подписки и обработки событий, выступающий в роли диспетчера между источниками событий и их обработчиками.

Цель данного паттерна — упростить взаимодействие между многочисленными компонентами системы, генерирующими и обрабатывающими события, устранить дублирование логики подписки и повышения масштабируемости.

Основные функции агрегатора событий:

- централизованная регистрация наблюдателей — обработчики событий регистрируются в одном месте — агрегаторе;
- приём и распределение событий — агрегатор принимает события от источников и передаёт их соответствующим обработчикам;
- снижение связности: источники событий и их обработчики не взаимодействуют напрямую, что упрощает поддержку и развитие системы.

В лабораторной работе №1 паттерн «Агрегатор событий» реализован как центральный компонент, управляющий подписками и маршрутизацией событий между модулями. Он включает следующие ключевые элементы:

- карта обработчиков событий, в которой по типу события хранится список соответствующих подписчиков;
- метод, вызываемый источниками для публикации нового события;
- метод подписки на событие для наблюдателя;
- механизм оповещения, вызывающий обработчики событий при поступлении нового события.

Паттерн «Агрегатор событий» является эффективным решением при разработке модульных, расширяемых и реактивных систем. Его использование в лабораторной работе позволило реализовать гибкую событийную модель, легко адаптируемую под новые сценарии.

2.5 Архитектура и реализация приложения

Приложение, разработанное в рамках лабораторной работы №1, представляет собой REST-сервис для управления сущностью Student, в который встроена система отслеживания событий с использованием паттернов «Посредник» и «Наблюдатель», а также паттерна «Агрегатор событий».

Архитектура приложения реализована по принципам многослойной модели и включает следующие ключевые компоненты:

- контроллер (Controller) — принимает HTTP-запросы от клиента и перенаправляет их в бизнес-логику;
- сервисный слой (Service) — реализует бизнес-логику работы с сущностью Student;
- репозиторий (Repository) — взаимодействует с базой данных через Spring Data JPA;
- DTO и мапперы — используются для передачи данных между слоями и преобразования сущностей;
- система событий — реализует паттерны наблюдатель, посредник и агрегатор событий для отслеживания операций CRUD над студентами.

Таблица, содержащая информацию о студентах, создана с помощью миграции liquibase:

Листинг 1 — миграция для таблицы student

```
databaseChangeLog:
  - changeSet:
      id: create-table-student
      author: Alexandr
      changes:
        - createTable:
            schemaName: public
            tableName: student
```

```
            columns:
              - column:
                  name: id
                  type: bigserial
                  constraints:
                    unique: true
                    primaryKey: true
              - column:
                  name: full_name
                  type: varchar(30)
                  constraints:
                    nullable: false
              - column:
                  name: group_name
                  type: varchar(30)
                  constraints:
                    nullable: false
```

Реализация паттернов:

1. EventMediator (посредник)

   Интерфейс EventMediator и его реализация EventMediatorImpl отвечают за регистрацию наблюдателей и рассылку уведомлений. Он инкапсулирует всю логику маршрутизации событий. Ниже приведен код интерфейса EventMediator:

   Листинг 2 — интерфейс EventMediator

```
public interface EventMediator {

  void subscribe(EventType event, Observer listener);

  void unsubscribe(EventType event, Observer listener);

  void publish(Event event);
}
```

2. Observer и реализация наблюдателей

   Интерфейс Observer и классы CreateObserver, ReadObserver, UpdateObserver, DeleteObserver реализуют реакцию на различные типы событий (создание, чтение, обновление, удаление). Ниже приведен код интерфейса Observer:

   Листинг 3 — интерфейс Observer

```
public interface Observer {
  void onEvent(Event event);
}
```

3. EventAggregator (агрегатор событий)

   Центральный компонент, объединяющий посредника и наблюдателей, представлен в виде конфигурационного класса EventMediatorConfig, где происходит связывание компонентов. Ниже приведен код для настройки агрегатора событий:

   Листинг 4 — конфигурация агрегатора событий

```
@Configuration
@RequiredArgsConstructor
public class EventMediatorConfig {

  private final KafkaSender kafkaSender;

  @Bean
  public EventMediator eventMediator() {
    EventMediator eventMediator = new EventMediatorImpl();

    eventMediator.subscribe(EventType.CREATE,
createEventListener());
    eventMediator.subscribe(EventType.READ,
readEventListener());
    eventMediator.subscribe(EventType.UPDATE,
updateEventListener());
    eventMediator.subscribe(EventType.DELETE,
deleteEventListener());

    return eventMediator;
  }

  @Bean
  public Observer createEventListener() {
    return new CreateObserver(kafkaSender);
  }

  @Bean
  public Observer readEventListener() {
    return new ReadObserver(kafkaSender);
  }

  @Bean
  public Observer updateEventListener() {
    return new UpdateObserver(kafkaSender);
  }

  @Bean
  public Observer deleteEventListener() {
    return new DeleteObserver(kafkaSender);
  }
}
```

4. Аннотация @Observer и аспект

Пользовательская аннотация @Observer и аспект ObserverAspect реализуют перехват вызовов методов сервисного слоя с целью генерации событий. Это позволяет автоматически отслеживать бизнес-операции без изменения основной логики. Ниже приведена реализация класса, обрабатывающий аннотацию:

Листинг 5 — Обработка аннотации @Observer

```
@Aspect
@Component
@RequiredArgsConstructor
public class ObserverAspect {

  private final EventMediator eventMediator;


@Around("@annotation(ru.mai.lab1.aggregator.annotation.Observer)")
  public Object trackEntityChange(ProceedingJoinPoint joinPoint) throws Throwable {
    MethodSignature signature = (MethodSignature) joinPoint.getSignature();
    Method method = signature.getMethod();
    Observer annotation = method.getAnnotation(Observer.class);

    Object result = joinPoint.proceed();

    eventMediator.publish(
        Event.builder()
            .eventType(annotation.event())
            .target(annotation.table())
            .payload(result)
            .timestamp(LocalDateTime.now())
            .build()
    );

    return result;
  }
}
```

5. KafkaSender (опционально)

Класс KafkaSender демонстрирует возможность расширения событийной модели до внешних систем, например, через публикацию событий в брокер Kafka.

Конфигурационный файл приложения application.yml:

Листинг 6 — конфигурация приложения

```
server:
  port: ${APPLICATION_PORT}

spring:
  application:
    name: lab1
  datasource:
    username: ${DB_USERNAME}
    password: ${DB_PASSWORD}
    url: ${DB_DRIVER}://${DB_HOST}:${DB_PORT}/${DB_NAME}
    driver-class-name: org.postgresql.Driver
  liquibase:
    liquibase-schema: public
    default-schema: public
    change-log: classpath:db/changelog/db.changelog-
master.yaml
    enabled: true
  kafka:
    bootstrap-servers: localhost:9092
    producer:
      key-serializer:
org.apache.kafka.common.serialization.StringSerializer
      value-serializer:
org.springframework.kafka.support.serializer.JsonSerializer
    consumer:
      group-id: observer-group
      key-deserializer:
org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
      properties:
        spring.json.trusted.packages: "*"
  jpa:
    show-sql: true
    properties:
      hibernate:
        format_sql: true
        default_schema: public
```

2.6 Демонстрация работы приложения

В качестве демонстрации работы используются docker-образы базы данных PostgreSQL и KafkaConfluent.

При запуске приложения на адресе /swagger-ui/index.html доступна документация о возможных действиях в приложении

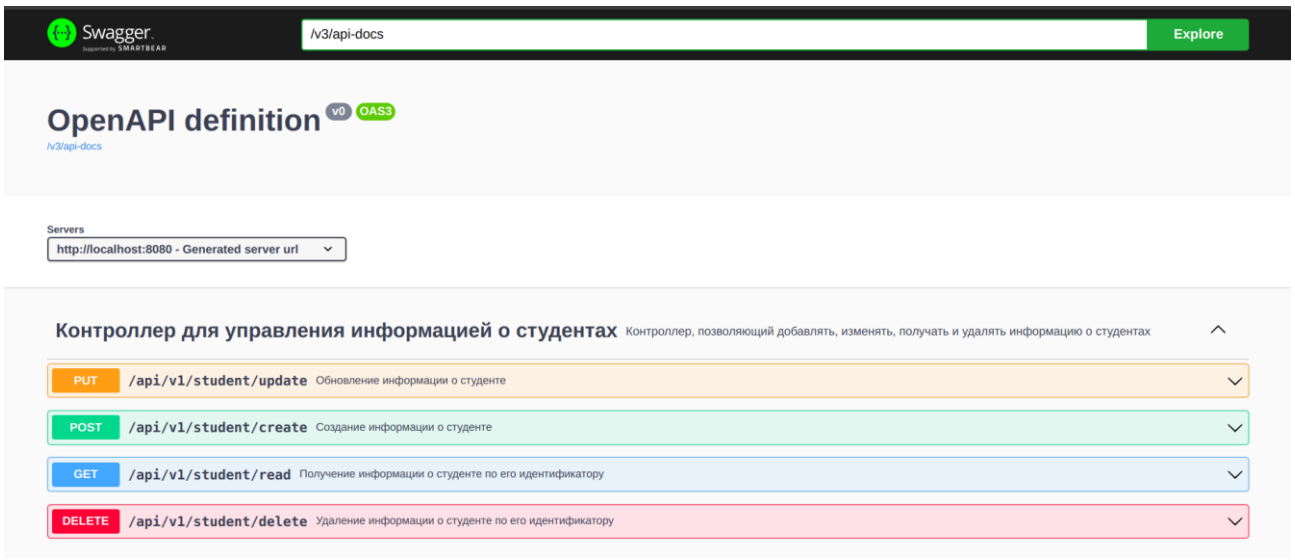

Рисунок 1 — OpenAPI документация

Для тестирования приложения нужно отправить http запрос на любой из перечисленных на рис. 1. В работе для этого используется Insomnia v11.1.0.

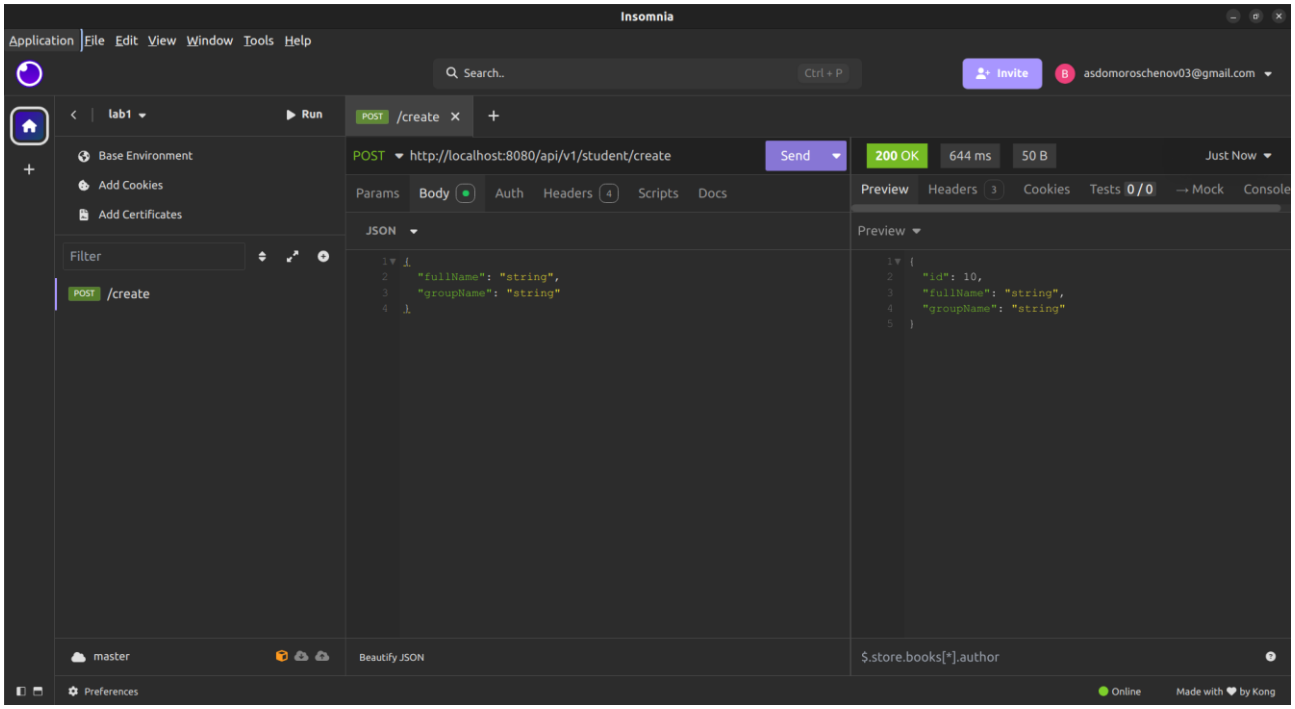

Рисунок 2 — Отправка http запроса

После отправки запроса — отслеживается какое событие произошло с помощью аннотации @Observer и целевой объект изменений. Вся эта информация используется для отправки сообщения в топик Kafka. Название топика соответствует названию события – create, read, update, delete.

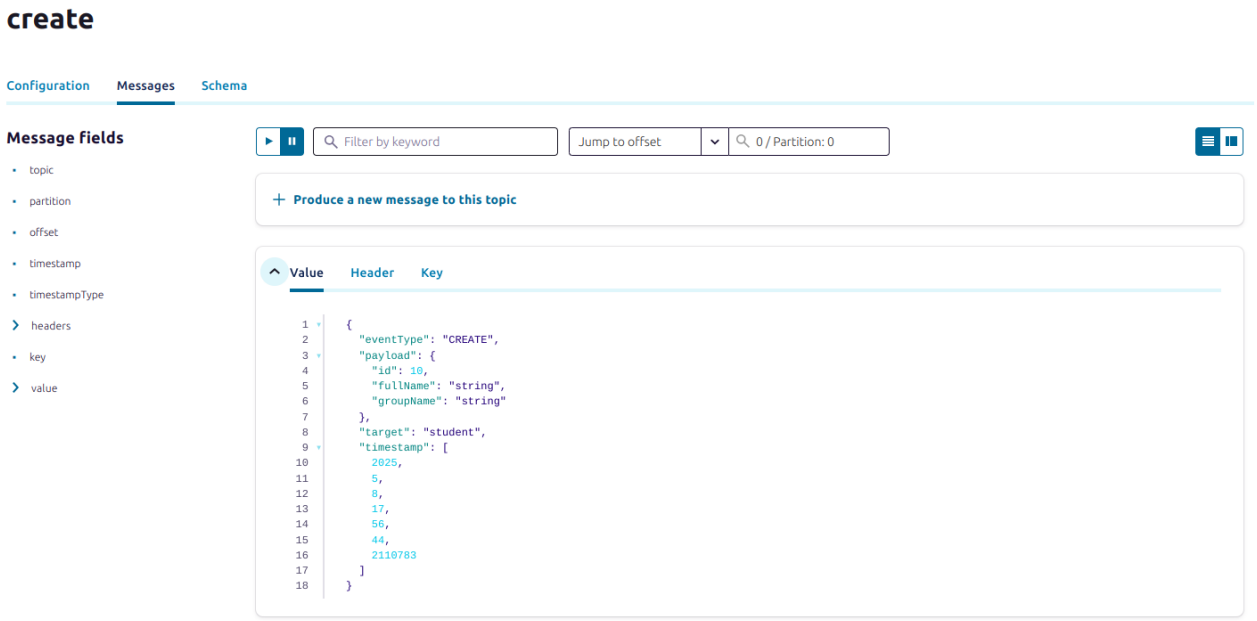

Рисунок 3 — сообщение о событии в топике Kafka

На рис. 3 видно, что в топик create пришло сообщение о событии create, которое отправил один из наблюдателей, подписанных на данное событие.

2.7 Выводы по лабораторной работе №1

В ходе выполнения лабораторной работы №1 была реализована система, демонстрирующая применение поведенческих паттернов проектирования — «Наблюдатель», «Посредник» и «Агрегатор событий» — в контексте событийно-ориентированной архитектуры.

Результатом работы стало REST-приложение, способное отслеживать операции над сущностью Student и передавать информацию об этих изменениях зарегистрированным обработчикам событий. Архитектура построена таким образом, чтобы обеспечить слабую связанность компонентов, что позволило легко добавлять и модифицировать логику обработки событий без вмешательства в основную бизнес-логику.

3 ЛАБОРАТОРНАЯ РАБОТА №2. МОДЕЛИРОВАНИЕ КОПИРОВАЛЬНОГО СЕРВИСА

3.1 Постановка задачи

Смоделируйте систему “копировальный сервис”. Сервис умеет обрабатывать запросы пользователей (запросы могут поступать не последовательно) на печать (фотографии разных форматов, документы формата А4 (например, диплом) в цветном или чёрно-белом варианте и т.д.). Для печати по определённому запросу необходимо отправить запрос на определённый принтер (первоначально запрос поступает на ч/б принтер, далее на цветной по необходимости (в зависимости от запроса) — примените шаблон проектирования “цепочка обязанностей”), далее настроить целевой принтер под тип документа в запросе (примените паттерн “состояние”) и выполнить печать (имитируйте процесс печати ожиданием). При печати фотографий рассмотрите варианты, когда в запросе уже есть фотография или же когда её нет (когда нет, необходимо проделегировать запрос сервису фотографирования - примените шаблон проектирования “заместитель”). Время в системе дискретно. Начальные настройки частей системы должны быть псевдослучайными. Продемонстрируйте работу системы. Обеспечьте наглядный вывод информации о процессе работы и результатах работы системы.

3.2 Использование паттерна «Цепочка обязанностей»

Паттерн «Цепочка обязанностей» (Chain of Responsibility) позволяет передавать запрос по цепочке обработчиков, где каждый объект-обработчик решает, может ли он обработать запрос, или передаёт его следующему элементу цепи. Это позволяет отделить отправителя запроса от его получателя и реализовать гибкую маршрутизацию запросов без жёсткой связности компонентов.

В рамках лабораторной работы №2 паттерн «Цепочка обязанностей» применён для построения механизма обработки печатных запросов разными

типами принтеров. Система моделирует работу копировального сервиса, в котором печать может быть выполнена либо на цветном принтере, либо на чёрно-белом, в зависимости от содержимого и требуемого формата.

Реализация паттерна

1. Интерфейс обработчика

   Интерфейс PrinterHandler определяет метод handle(PrintRequest request), который реализуют конкретные обработчики. Также предусмотрен метод setNext(PrinterHandler next) для установки следующего обработчика в цепи.

2. Конкретные обработчики

   Классы ColorPrinterHandler и BlackWhitePrinterHandler реализуют конкретные типы принтеров. Каждый из них проверяет, может ли выполнить печать в заданном формате (цвет/чёрно-белый), и либо обрабатывает запрос, либо передаёт его дальше.

3. Конфигурация цепочки

   Класс PrinterChainConfiguration собирает цепочку обработчиков, устанавливая их порядок: сначала чёрно-белый, затем цветной принтер. Это обеспечивает приоритет цветной печати, если она доступна.

3.3 Использование паттерна «Состояние»

Паттерн «Состояние» (State) относится к поведенческим шаблонам проектирования и позволяет объекту изменять своё поведение в зависимости от текущего состояния. При этом объект делегирует выполнение поведения объектам-состояниям, тем самым избегая множества условных операторов (if, switch) и обеспечивая более чистую, расширяемую архитектуру.

В лабораторной работе №2 паттерн «Состояние» используется для реализации поведения принтеров в зависимости от типа печатаемого содержимого — документа или фотографии. Для каждого типа принтера (цветного и чёрно-белого) реализуются различные состояния, определяющие особенности выполнения задания печати.

Реализация паттерна

1. Общий интерфейс состояний

   Интерфейс PrinterState определяет метод print(String content), реализуемый всеми состояниями. Это обеспечивает единый контракт для всех возможных вариантов поведения.

2. Конкретные состояния

   Для чёрно-белого принтера реализованы классы BlackWhiteDocumentState и BlackWhitePhotoState, а для цветного — ColorDocumentState и ColorPhotoState. Эти классы инкапсулируют поведение печати в зависимости от типа содержимого

3. Контекст-принтер

   Классы BlackWhitePrinter и ColorPrinter содержат ссылку на текущее состояние (PrinterState) и делегируют вызов метода печати соответствующему объекту-состоянию. Переход между состояниями осуществляется на основе данных запроса (ContentType.DOCUMENT или ContentType.PHOTO).

3.4 Использование паттерна «Заместитель»

Паттерн «Заместитель» (Proxy) относится к структурным шаблонам проектирования и применяется в тех случаях, когда необходимо контролировать доступ к объекту или добавить дополнительную логику при его использовании, не изменяя сам объект. Заместитель реализует тот же интерфейс, что и основной объект, и может выступать в роли обёртки, делегируя вызовы реальному объекту.

В лабораторной работе №2 паттерн «Заместитель» использован для расширения логики работы с принтерами (цветным и чёрно-белым), без изменения их исходной реализации. Это позволило внедрить дополнительные действия, такие как логирование, симуляция задержек или подсчёт ресурсов, — не нарушая принцип единственной ответственности.

Реализация паттерна

1. Интерфейс принтера

   Все принтеры реализуют общий интерфейс Printer, содержащий метод print(PrintRequest request).

2. Основные реализации

   Классы ColorPrinter и BlackWhitePrinter реализуют основную логику печати, включая выбор состояния и выполнение печати в зависимости от типа содержимого.

3. Заместители (прокси)

   Классы ColorPrinterProxy и BlackWhitePrinterProxy реализуют тот же интерфейс Printer, но выступают в качестве посредников. Они:

   a. получают запрос от пользователя;
   b. выполняют вспомогательные действия (например, вывод в консоль, задержку);
   c. делегируют вызов основному принтеру.

3.5 Архитектура и реализация приложения

Разработанное в рамках лабораторной работы №2 приложение моделирует работу копировального сервиса, в котором печатные задания обрабатываются в зависимости от типа содержимого (документ или фотография) и доступных принтеров (цветной или чёрно-белый). Архитектура системы построена с использованием нескольких паттернов проектирования: «Цепочка обязанностей», «Состояние» и «Заместитель».

Общая структура

Приложение реализовано по многослойной архитектуре и включает следующие основные компоненты:

- контроллер (PrinterController) — точка входа в приложение, получающая запросы на печать;
- цепочка обработчиков (PrinterHandler, ColorPrinterHandler, BlackWhitePrinterHandler) — реализует маршрутизацию запросов между принтерами;

– прокси-принтеры (ColorPrinterProxy, BlackWhitePrinterProxy) — выполняют дополнительную логику перед делегированием печати;
– состояния принтеров (PrinterState и его реализации) — инкапсулируют поведение в зависимости от типа задания;
– конфигуратор (PrinterChainConfiguration) — собирает цепочку обязанностей и внедряет зависимости.

Ключевые элементы архитектуры

1. Паттерн «Цепочка обязанностей»

   Позволяет организовать последовательную проверку принтеров на возможность обработки запроса. Если первый обработчик (например, цветной принтер) не может выполнить задание, оно передаётся следующему.
2. Паттерн «Состояние»

   Реализует переключение поведения принтера в зависимости от типа содержимого. Состояния DocumentState и PhotoState определяют, как именно будет производиться печать.
3. Паттерн «Заместитель»

   Вводит дополнительную прослойку между контроллером и реальными принтерами. Прокси добавляют логирование и симуляцию задержек, не нарушая логику базового класса.

Принцип работы:

1. пользователь отправляет запрос на печать (например, фотографии);
2. контроллер передаёт запрос в цепочку обработчиков;
3. первый обработчик (черно-белый принтер) проверяет возможность обработки запроса через canHandle. Если может — обращается в класс PrinterConfigurer за настройкой принтера под тип целевого документа и получает настроенный прокси-принтер;
4. прокси-принтер логирует операцию и передаёт управление в реальный принтер;

5. принтер определяет состояние (PhotoState) и выполняет печать с соответствующей логикой;
6. если в запросе указана печать фото, но при этом фотография отсутствует, то имитируется процесс фотографирования через паттерн «Заместитель». После успешной проверки на наличие фото — обработка запроса завершается.

Особенности реализации

- классы чётко разделены по ответственности: логика маршрутизации, состояния, печати и конфигурации изолированы;
- приложение легко расширяется: можно добавить новые типы принтеров, состояний или дополнительных прокси без изменений существующего кода;
- используется Spring-контекст для внедрения зависимостей и конфигурации цепочки.

3.6 Демонстрация работы приложения

Для обработки запросов печати создан эндпоинт – POST /api/v1/printer/print. Запрос для печати содержит поле — contentType, которое может принимать 2 значения — PHOTO, DOCUMENT (печать фото или документа), поле color — принимает значения true или false (цвтеная или черно-белая печать) и поле content – само содержимое для печати. Далее приведен пример http запроса для печати:

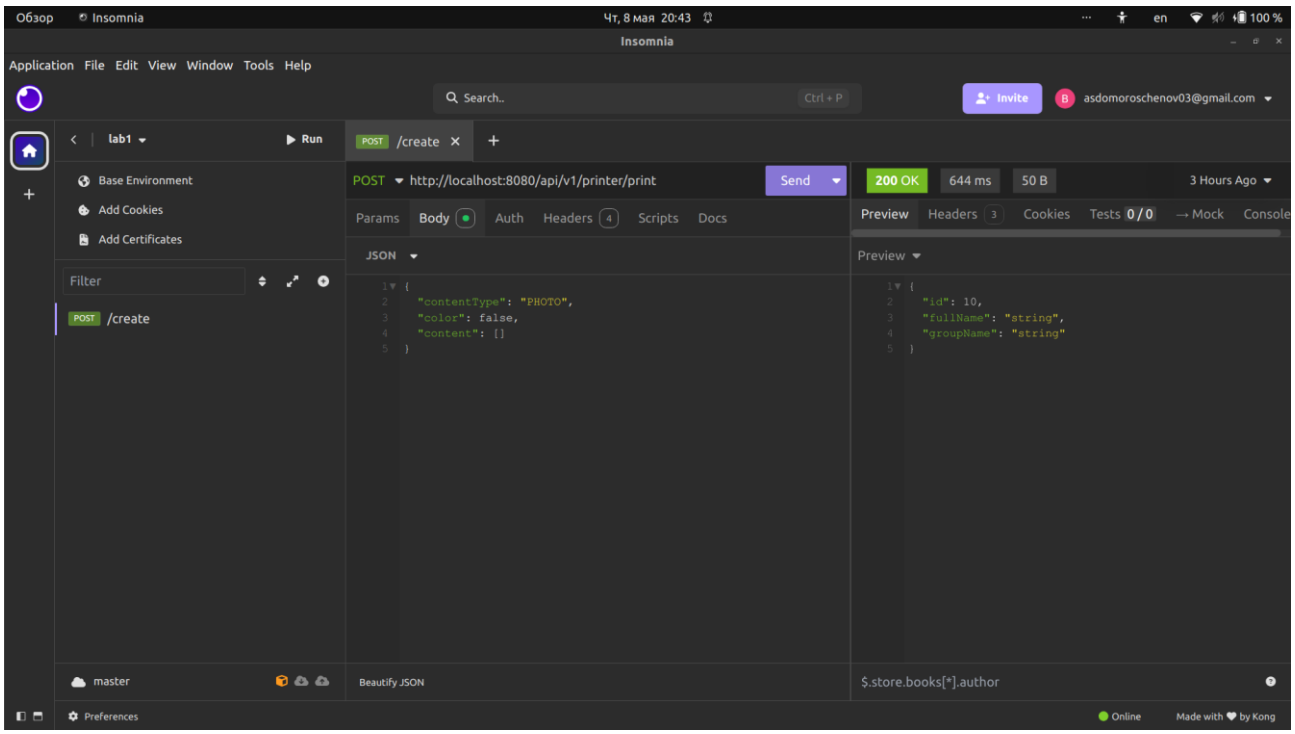

Рисунок 4 — пример запроса на печать

После отправки запроса — можно увидеть имитацию работы принтера с помощью логирования в консоль:

```
2025-05-08T20:46:42.211+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.mai.lab2.printer.chain.PrinterHandler  : Обработка запроса на печать...
2025-05-08T20:46:42.211+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.c.i.BlackWhitePrinterConfigurer  : Настраиваю черно-белый принтер под тип документа...
2025-05-08T20:46:42.211+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.c.i.BlackWhitePrinterConfigurer  : Настройка черно-белого принтера завершена!
2025-05-08T20:46:42.211+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.s.b.p.BlackWhitePrinterProxy     : Фото не предано в запросе! Фотографирую...
2025-05-08T20:46:43.212+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.s.b.p.BlackWhitePrinterProxy     : Фото сделано, работа продолжается...
2025-05-08T20:46:43.212+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.s.b.state.BlackWhitePhotoState   : Печатаю черно-белое фото...
2025-05-08T20:46:44.212+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.m.l.p.s.b.state.BlackWhitePhotoState   : Черно-белое фото распечатано!!!
2025-05-08T20:46:44.212+03:00  INFO 113656 --- [lab2] [nio-8080-exec-1] r.mai.lab2.printer.chain.PrinterHandler  : Запрос обработан
```

Рисунок 5 — результат обработки запроса

3.7 Выводы по лабораторной работе №2

В ходе выполнения лабораторной работы №2 было разработано программное приложение, моделирующее работу копировального сервиса с применением структурных и поведенческих паттернов проектирования. Основное внимание было уделено реализации шаблонов «Цепочка обязанностей», «Состояние» и «Заместитель».

Применение паттерна «Цепочка обязанностей» позволило организовать гибкую маршрутизацию печатных запросов между различными типами принтеров. Благодаря этому стало возможным реализовать обработку заданий в

порядке приоритета, без жёсткой привязки отправителя к конкретному исполнителю.

Шаблон «Состояние» обеспечил переключение логики печати в зависимости от типа содержимого (документ или фотография). Это повысило адаптивность системы и позволило избежать дублирования кода и громоздких условных конструкций.

Паттерн «Заместитель» применён для инкапсуляции дополнительной логики (логирование, симуляция задержек) при взаимодействии с реальными объектами принтеров. Это позволило расширить функциональность без изменения базовых классов, повысив модульность и тестируемость.

Архитектура приложения получилась масштабируемой, модульной и легко расширяемой. Использование шаблонов проектирования сделало систему гибкой, упростило добавление новых сценариев обработки запросов и позволило достоверно смоделировать поведение реального копировального оборудования.

Таким образом, лабораторная работа №2 продемонстрировала практическую ценность паттернов проектирования в задачах моделирования сложных сценариев взаимодействия объектов. Полученные знания и реализованные подходы могут быть успешно использованы в реальных проектах разработки программных систем.

ЗАКЛЮЧЕНИЕ

В данной курсовой работе была рассмотрена и реализована практика применения шаблонов проектирования при разработке программных систем. Целью исследования являлась демонстрация возможностей паттернов в обеспечении модульности, расширяемости и гибкости архитектуры приложений.

В теоретической части был выполнен обзор классификации шаблонов проектирования, а также подробно рассмотрены поведенческие и структурные паттерны: наблюдатель, посредник, состояние, цепочка обязанностей и заместитель. Эти шаблоны были выбраны как наиболее актуальные для построения событийных и логически адаптивных систем.

Практическая часть включала выполнение двух лабораторных работ:

- В первой лабораторной работе была реализована система отслеживания изменений данных с использованием шаблонов наблюдатель, посредник и агрегатор событий. Это позволило продемонстрировать преимущества слабой связанности и централизованной маршрутизации событий.
- Во второй лабораторной работе было смоделировано поведение копировального сервиса с применением шаблонов цепочка обязанностей, состояние и заместитель. Такое решение позволило реализовать динамическую маршрутизацию запросов, адаптивное поведение компонентов и изоляцию дополнительной логики от основной бизнес-функциональности.

В результате курсовой работы удалось продемонстрировать, что шаблоны проектирования являются мощным инструментом для создания надёжных и масштабируемых программных решений. Их применение позволяет повысить читаемость и переиспользуемость кода, а также облегчает дальнейшее сопровождение и развитие программных систем.

Полученные в ходе выполнения курсовой работы знания и навыки могут быть использованы в будущей профессиональной деятельности при проектировании и разработке сложных приложений.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Шаблоны проектирования. — СПб.: Питер, 2022. — 384 с.
2. Фримен Э., Фримен Э. Head First. Паттерны проектирования. — М.: Вильямс, 2021. — 656 с.
3. Шилдт Г. Java. Полное руководство. — 11-е изд. — М.: Вильямс, 2022. — 1344 с.
4. Блох Дж. Эффективное программирование на Java. — 3-е изд. — М.: Вильямс, 2019. — 416 с.
5. МакКи С. Паттерны архитектуры корпоративных приложений. — СПб.: Питер, 2020. — 560 с.
6. GoF Design Patterns Documentation. — [Электронный ресурс]. — Режим доступа: https://refactoring.guru/ru/design-patterns, свободный. — Дата обращения: 08.05.2025.
7. Spring Framework Documentation. — [Электронный ресурс]. — Режим доступа: https://docs.spring.io/spring-framework/docs/current/reference/html/, свободный. — Дата обращения: 08.05.2025.
8. Spring Boot Reference Documentation. — [Электронный ресурс]. — Режим доступа: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/, свободный. — Дата обращения: 08.05.2025.
9. Oracle. The Java™ Tutorials. Design Patterns. — [Электронный ресурс]. — Режим доступа: https://docs.oracle.com/javase/tutorial/, свободный. — Дата обращения: 08.05.2025.
10. Стивенс Р. Паттерны проектирования для профессиональных программистов. — М.: ДМК Пресс, 2020. — 512 с.

ПРИЛОЖЕНИЕ

Полные исходные тексты лабораторных работ, описанных в курсовой работе, размещены в открытом репозитории на GitHub. Ссылка на репозиторий: https://github.com/ASDomoroschenov/Software-Engineering