МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра инструментального и прикладного программного обеспечения (ИиППО)

КУРСОВАЯ РАБОТА

по дисциплине: Технологии разработки программного обеспечения
по профилю: Разработка программных продуктов и проектирование информационных систем
направления профессиональной подготовки: 09.03.04 «Программная инженерия»

Тема: Разработка десктоп-приложения для управления проектами Git

Студент: Мусатов Иван Алексеевич
Группа: ИКБО-11-23
Руководитель: Супрун Антон Павлович

Оценка по итогам защиты: _____

Подпись студента:_____ Подпись руководителя: _____

М. РТУ МИРЭА. 2025

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

GUI	–	Graphical User Interface (графический пользовательский интерфейс
JDK	–	Java Development Kit (комплект средств разработки на языке Java)
JAR	–	Java Archive (архивный формат скомпилированного Java приложения)
MVVM	–	Model-View-ViewModel (архитектурный паттерн Модель-Представление-МодельПредставление)
UI	–	User Interface (пользовательский интерфейс)
OS	–	Operating System (операционная система)
API	–	Application Programming Interface (прикладной программный интерфейс)
IT	–	Information Technology (информационные технологии)
ПО	–	Программное Обеспечение
ПП	–	Программный Продукт
ТЗ	–	Техническое Задание

ТЕРМИНЫ И ОПРЕДЕЛЕНИЯ

Git	–	распределенная система контроля версий
Git-репозиторий	–	каталог файлов, содержащий историю изменений проекта и метаданные Git
Задача разработки	–	элемент планирования, описывающий работы, которые необходимо выполнить
Архитектура ПО	–	совокупность принципов и решений, определяющих структуру программного продукта и взаимодействие его компонентов
Модульное тестирование	–	процесс проверки отдельных компонентов программного продукта на корректность их работы
Кроссплатформенное приложение	–	программный продукт, способный функционировать на различных операционных системах без изменения исходного кода
Сборка проекта	–	процесс компиляции исходного кода и формирования исполняемого файла или установочного пакета
Установщик	–	программный компонент, предназначенный для установки приложения

ВВЕДЕНИЕ

В современном процессе разработки программного обеспечения значительную роль играет управление проектами, а также автоматизация рутинных операций, связанных с сопровождением исходного кода. С ростом количества проектов возникает потребность в централизованных средствах учета и планирования программных решений.

Большинство существующих инструментов ориентированы либо на удаленные репозитории и командную работу, либо на использование командной строки, что не всегда удобно для повседневной работы с локальными проектами. В связи с этим актуальной задачей является разработка программного продукта, предоставляющего удобный графический интерфейс для управления локальными Git-репозиториями, планирования задач разработки и автоматизации пользовательских сценариев.

Объектом данной курсовой работы является процесс управления локальными Git-проектами в ходе разработки программного обеспечения.

Предметом работы является совокупность архитектурных, программных и технологических решений, применяемых при разработке утилитарного программного продукта для разработчиков программного обеспечения.

Целью курсовой работы является разработка кроссплатформенного программного продукта, обеспечивающего централизованное управление локальными Git-репозиториями, задачами разработки и сценариями автоматизации.

Для достижения поставленной цели в рамках курсовой работы необходимо решить следующие задачи:

- сформировать концепцию продукта;

- определить требования и формализовать их в виде технического задания;
- спроектировать архитектуру программного продукта и базу данных;
- реализовать программное обеспечение в соответствии с проектом;
- применить паттерны проектирования для повышения сопровождаемости кода;
- реализовать модульные тесты и выполнить верификацию продукта;
- написать инструкцию по сборке и установке;
- составить руководство пользователя.

Ожидаемым результатом работы является программный продукт, соответствующий поставленной цели, а также пакет документации, состоящий из Технического задания, Инструкции по сборке и установке и Руководства пользователя.

1 АНАЛИЗ И ПРОЕКТИРОВАНИЕ ПРОДУКТА

1.1 Концепция продукта

Разрабатываемый программный продукт представляет собой настольное (desktop) приложение, предоставляющее инструменты для комплексного взаимодействия с локальными Git репозиториями.

Назначением данного продукта является автоматизация и упрощение процесса разработки программного обеспечения. Продукт решает задачи по учету и централизованному хранению сведений о локальных проектах Git, планированию и учету задач разработки, а также автоматизации рутинных пользовательских команд по сборке, запуску и тестированию программных решений.

Ключевой особенностью ПП является централизованное хранение данных обо всех локальных проектах, позволяющее вести учет их сведений и планирование разработки.

Новизной продукта является возможность настройки пользовательских сценариев автоматизации в рамках каждого локального проекта.

Целевой аудиторией ПП является сообщество разработчиков программного обеспечения, а также иные пользователи системы контроля версий Git. Продукт может быть применим как в частой практике разработки, так и в рамках команд разработки в IT-компаниях.

Данный продукт является масштабируемым инструментом с открытым исходным кодом. Это означает его открытость к расширению и возможность пользовательской настройки для своих нужд.

В целях формирования бренда продукта было принято решение об использовании названия «GitPM – Git Project Manager». Название отражает

назначение и предметную область продукта. А сокращение «GitPM» позволяет в удобной и лаконичной форме именовать продукт.

Для визуализации пользовательских сценариев взаимодействия с продуктом представим UML use-case диаграмму (см. рис. 1), сделанную при помощи инструмента Draw.io.

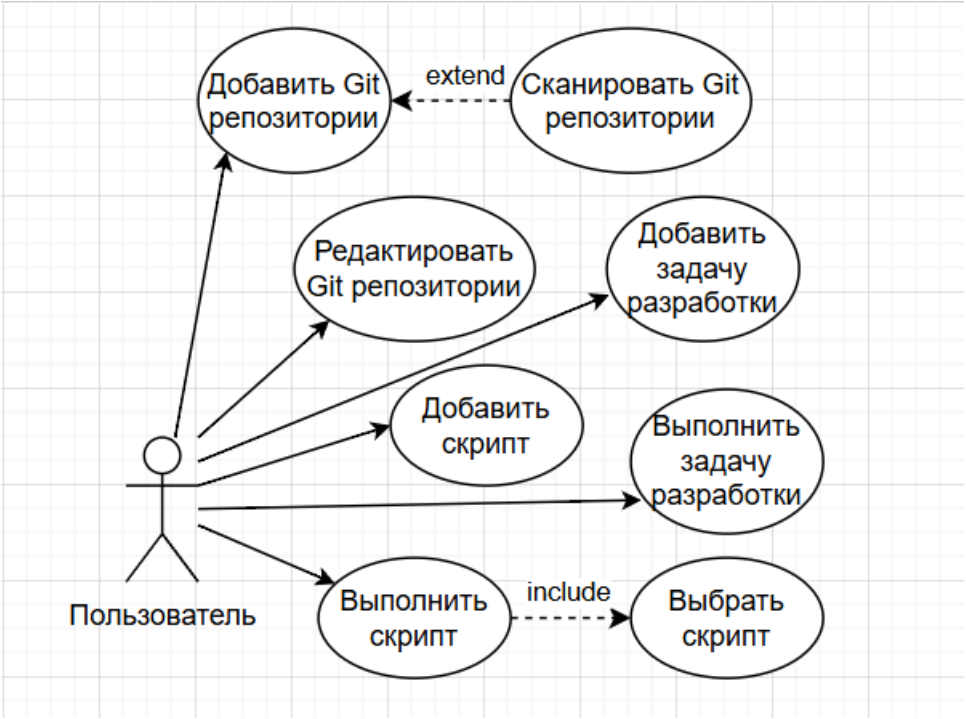

Рисунок 1 – UML диаграмма вариантов использования GitPM

1.2 Формирование требований к продукту

В соответствии с представленной концепцией и целью разработки программного продукта сформируем комплекс требований, охватывающих как функциональные, так и нефункциональные требования.

Полное и формальное описание требований в соответствии с ГОСТ 34.602-2020 [1] представлено в «Техническом задании на разработку программного продукта GitPM», текст которого представлен в Приложении А.

1.3 Обоснование технических решений и инструментария

Выбор технических решений для продукта «GitPM» обусловлен требованиями к созданию легковесной и кроссплатформенной утилиты. В связи с этим тип приложения – desktop, не зависящий от подключения к сети.

Для хранения локальных данных проекта выбрана встраиваемая реляционная СУБД SQLite [2]. Особенностью SQLite является отсутствие необходимости в запуске отдельного сервера БД. База данных представляет собой единый файл на диске, что полноценно соответствует концепциям легковесности и простоты использования.

В целях обеспечения кроссплатформенности приложения для разработки был выбран язык Java. Использование JVM позволяет запускать скомпилированное приложение на любых операционных системах без модификации исходного кода. Помимо кроссплатформенности язык обладает мощной экосистемой для решения практически любых задач, а строгая статическая типизация позволяет минимизировать Runtime ошибки. Java в полной мере поддерживает ООП парадигму, что позволяет выстраивать сложные и масштабируемые архитектурные паттерны.

В качестве инструмента для выстраивания графического интерфейса был выбран фреймворк JavaFX [3]. В экосистеме Java это один из современных и многофункциональных фреймворков. JavaFX обладает широким набором элементов и способов стилизации для создания визуально приятного и функционального интерфейса. Поддержка привязки свойств (Property Binding) и разметки FXML для декларативного описания интерфейса позволяют четко разделить логику представления и логику обработки, обеспечивая возможность использования архитектур MVC и MVVM для слоя представления.

Для автоматизации сборки и управления зависимостями был выбран фреймворк Apache Maven [4]. Сборщик позволяет организовать

централизованное управление зависимостями (в нашем проекте их немало: Javafx-controls, Javafx-fxml, Sqlite-jdbc, Jgit, Slf4j-api, Junit-jupiter-api, Junit-jupiter-engine). Maven позволяет автоматизировать типовые задачи, предоставляя готовые цели (goals) для компиляции, запуска тестов и сборки JAR-файлов.

1.4 Проектирование архитектуры

Для обеспечения гибкости и сопровождаемости программного продукта была выбрана многослойная (onion) архитектура, также известная как «Чистая архитектура» (Clean Architecture) [5]. Данный подход позволяет строго разделить ответственность между компонентами системы, минимизировать связанность и сделать ядро приложения независимым от деталей реализации (фреймворков, БД, UI).

Опишем некоторые принципы Чистой архитектуры. Архитектура организуется в виде концентрических слоев с направлением зависимостей внутрь, к центру. Внешние слои зависят от внутренних, что гарантирует независимость бизнес-правил (ядра) от изменения способа отображения данных или БД.

Проектируемая система включает следующие слои:

1. Доменный слой (Domain Layer) – Ядро системы. Содержит сущности (models) – объекты, отражающие ключевые концепции предметной области, и интерфейсы репозиториев (repositories). Слой не содержит ссылок на БД, GUI или внешние библиотеки. Только описание бизнес-логики и правил.

2. Слой приложения (Application Layer). Содержит сервисы приложения (services) и сценарии использования (use-cases). Слой организует поток данных к доменным сущностям и обратно, координируя их работу для выполнения конкретных сценариев пользователя. Этот слой зависит только от Доменного слоя.

3. Слой инфраструктуры (Infrastructure Layer). Содержит конкретные реализации контрактов, объявленных в Доменном слое. Сюда входят репозитории (repositories), работающие с БД, внешние службы (gateways / adapters) – реализации внешних утилитарных систем.

4. Слой представления (Presentation Layer). Включает в себя все, что связано с пользовательским интерфейсом. Т.е. файлы разметки, контроллеры (controllers), модели представления (viewmodels). Слой зависит от слоя Приложения и от GUI фреймворка.

В нашем ПО слой Представления сам по себе является также и реализацией архитектурного паттерна MVVM (Model-View-ViewModel), который позволяет отделить представление от логики обработки через посредника, реагирующего на изменение представления.

Для наглядного представления устройства архитектуры представим схему взаимодействия слоев (см. рис. 2).

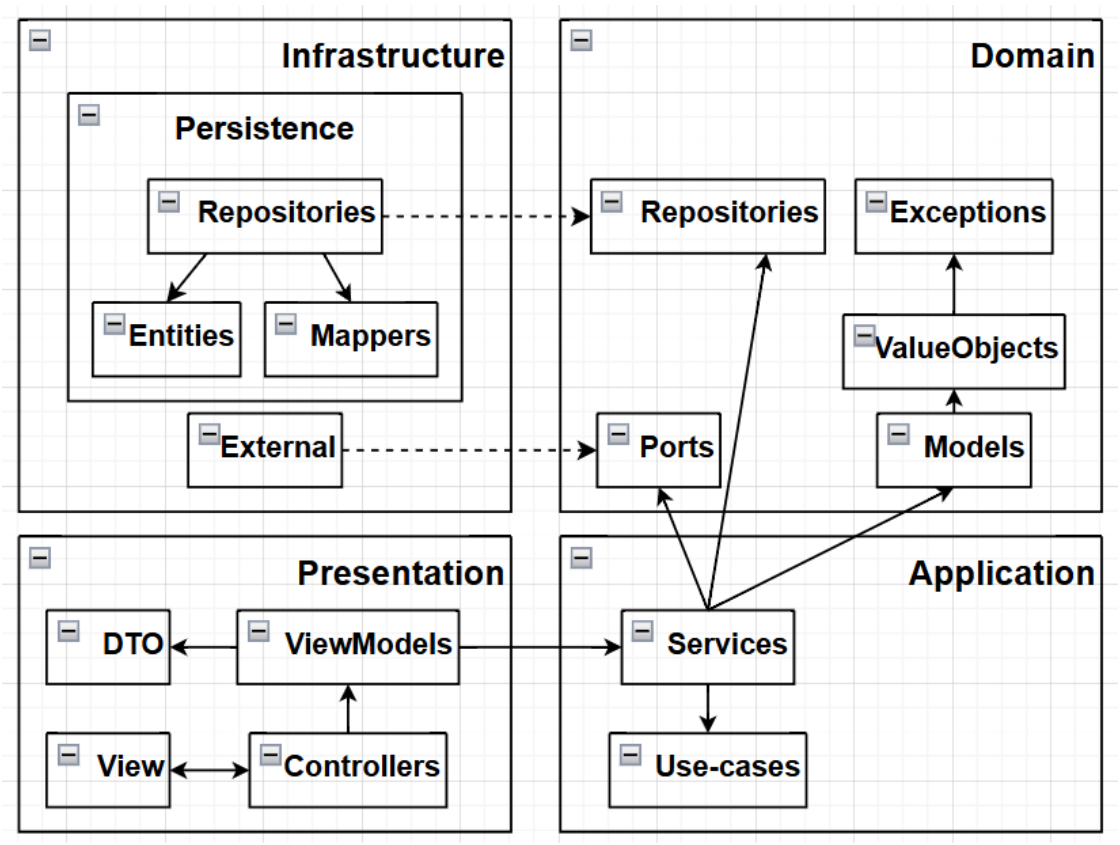

Рисунок 2 – Схема взаимодействия слоев в Чистой архитектуре

1.5 Проектирование базы данных

Для обеспечения постоянного хранения данных о сущностях нашего программного продукта создадим базу данных. Для этого сначала спроектируем ее схему.

Выделим ключевые сущности ПП и их свойства и определим для них таблицы БД. Для работы с Git проектами необходимо определить такую сущность, как Project. Помимо названия и описания обязательным атрибутом проекта является путь до папки с локальным репозиторием. При наличии связанного удаленного репозитория можно хранить и его адрес. Также в качестве технической информации можно хранить данные о времени добавления проекта в Менеджер.

Чтобы осуществлять планирование задач разработки, необходимо также хранить и данные о задачах. Для этого создадим таблицу Task. Кортежи таблицы будут ссылаться на проект, к которому они принадлежат. Задачи также имеют название и содержание, дату создания и флаг выполнения. Помимо этого, также определим опциональное поле с указанием даты дедлайна и поле с указанием приоритета выполнения задачи.

Для организации пользовательских скриптов автоматизации создадим две таблицы: Script и Command. По сути, Script – это именованный набор команд. Поэтому для него укажем только проект, к которому он относится, название и описание. А команды уже будут содержать непосредственно исполняемые команды. Для этого будем ссылаться на скрипт, к которому они относятся, укажем директорию выполнения команды, сам текст исполняемой команды и порядок ее выполнения в рамках скрипта.

На основании проведенного проектирования в инструменте ChartDB была сформирована физическая модель базы данных (см. рис. 3), отражающая таблицы, их связи и атрибуты.

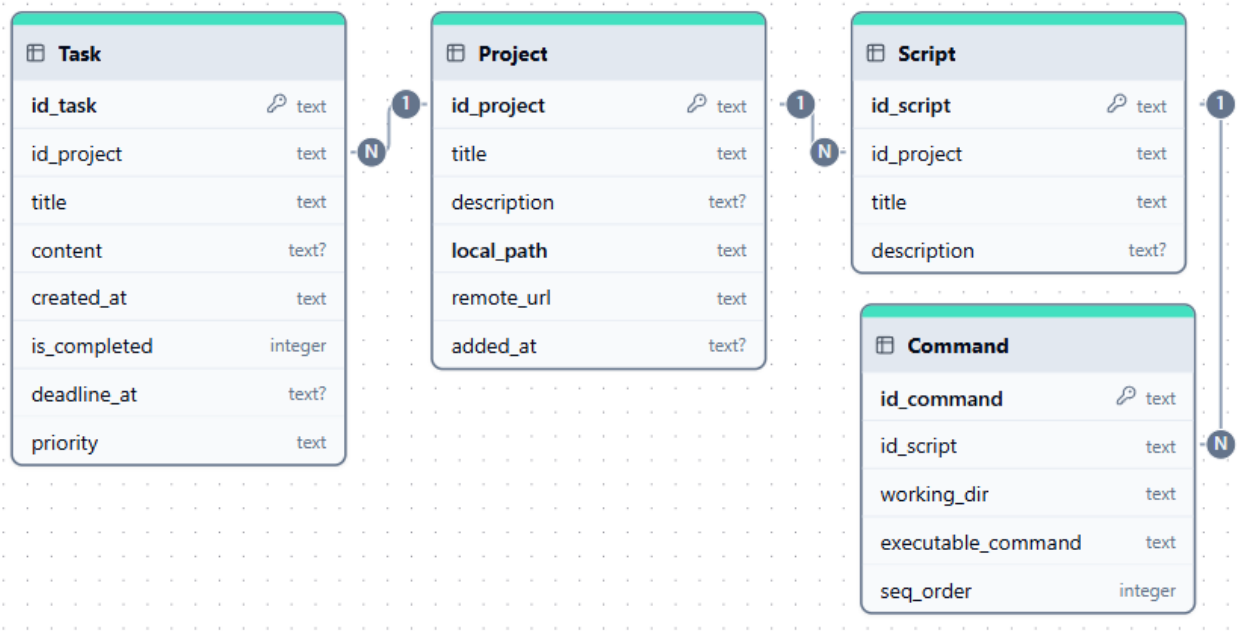

Рисунок 3 – Физическая модель базы данных

1.6 Проектирование дизайна интерфейса

Спроектируем в соответствии с ТЗ удобный и приятный пользовательский интерфейс.

Для придания продукту бренда и узнаваемости сформируем логотип. Так как продукт нацелен на взаимодействие и автоматизацию работы с Git-проектами, то целесообразно было бы сделать отсылку к Git. Git традиционно использует свой логотип разветвленной ветки. Поэтому заимствуем его, но в нейтральном цвете. Для указания процесса автоматизации и механических работ традиционно используется шестеренка. Поэтому будем использовать ее. Скомбинировав эти два образа, и избрав в качестве цветовой палитры мягкий мятный цвет, нарисуем в редакторе Inkscape логотип (см. рис. 4).

Рисунок 4 – Логотип GitPM

В целях поддержки пользовательского опыта (UX) добавим ярлыки на кнопки. Создадим узнаваемые иконки (см. рис. 5), которые впоследствии навесим на кнопки.

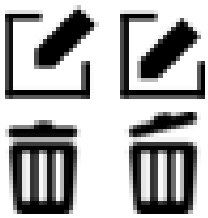

Рисунок 5 – Дизайн ярлыков кнопок

Определим также схему пользовательского интерфейса. Так как придется отображать слишком много информации (список проектов, информация о проекте, список задач, список скриптов и панель взаимодействия со скриптами) целесообразно использовать панель с вкладками. Слева расположим список проектов, а в правой части вкладки для каждой из предоставляемых функциональностей. На рис. 6 представлен дизайн пользовательского интерфейса, сделанный в инструменте Figma.

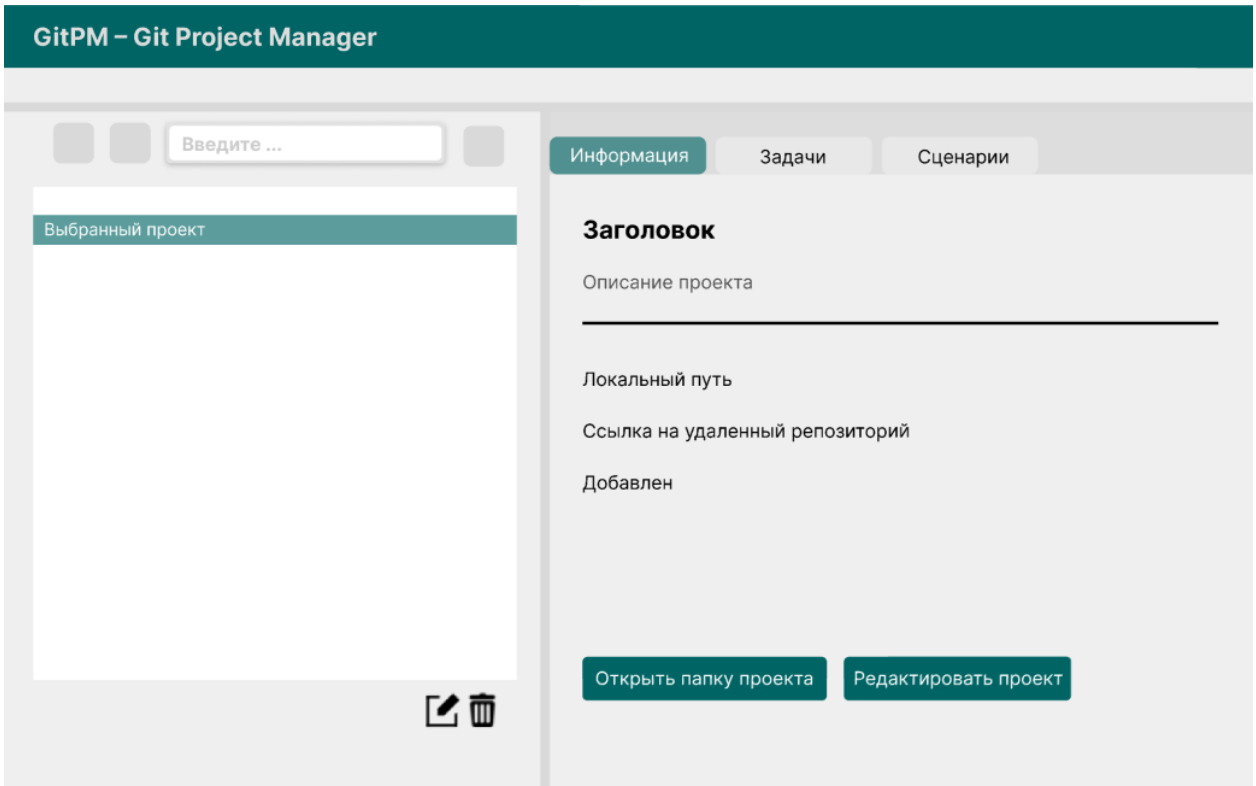

Рисунок 6 – Дизайн пользовательского интерфейса GitPM

2 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Реализация архитектуры

В соответствии с проектом архитектуры организуем модули исходного кода для каждого из слоев Чистой архитектуры. Каждый модуль разделим на пакеты в соответствии с компонентами слоев. Организация пакетов исходного кода представлена на рис. 7.

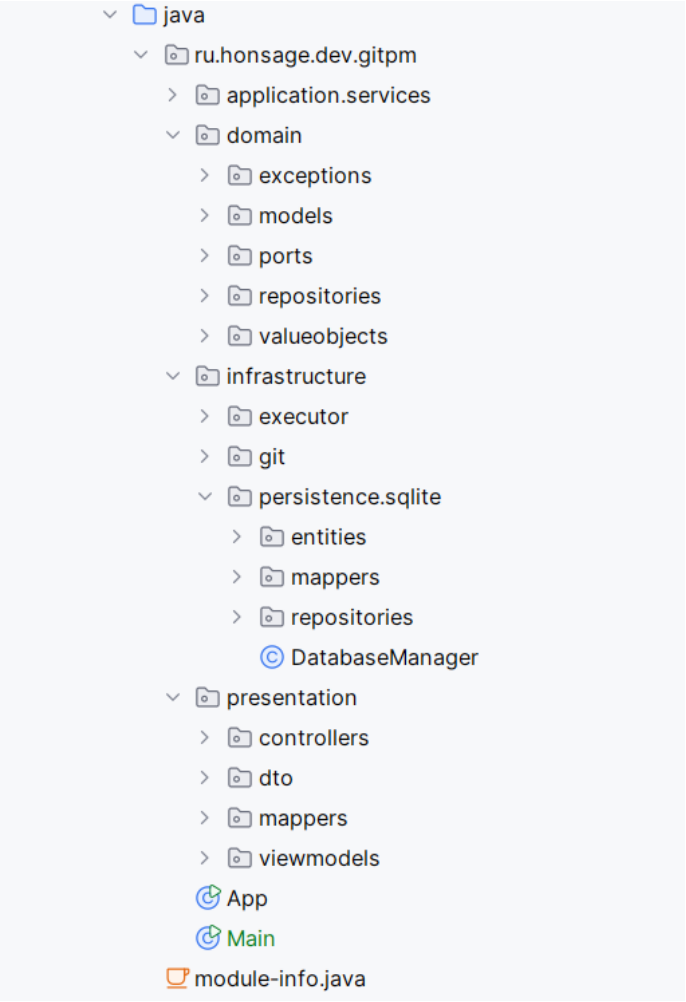

Рисунок 7 – Организация пакетов исходного кода

Для визуализации взаимосвязей компонентов кода сформируем UML диаграмму классов. Ввиду большого числа классов не будем специфицировать атрибуты и методы классов. Также опустим некоторые утилитарные классы, не влияющие на логику программы (например, mapper’ы). Результат построения при помощи инструмента PlantUML представлен на рис. 8.

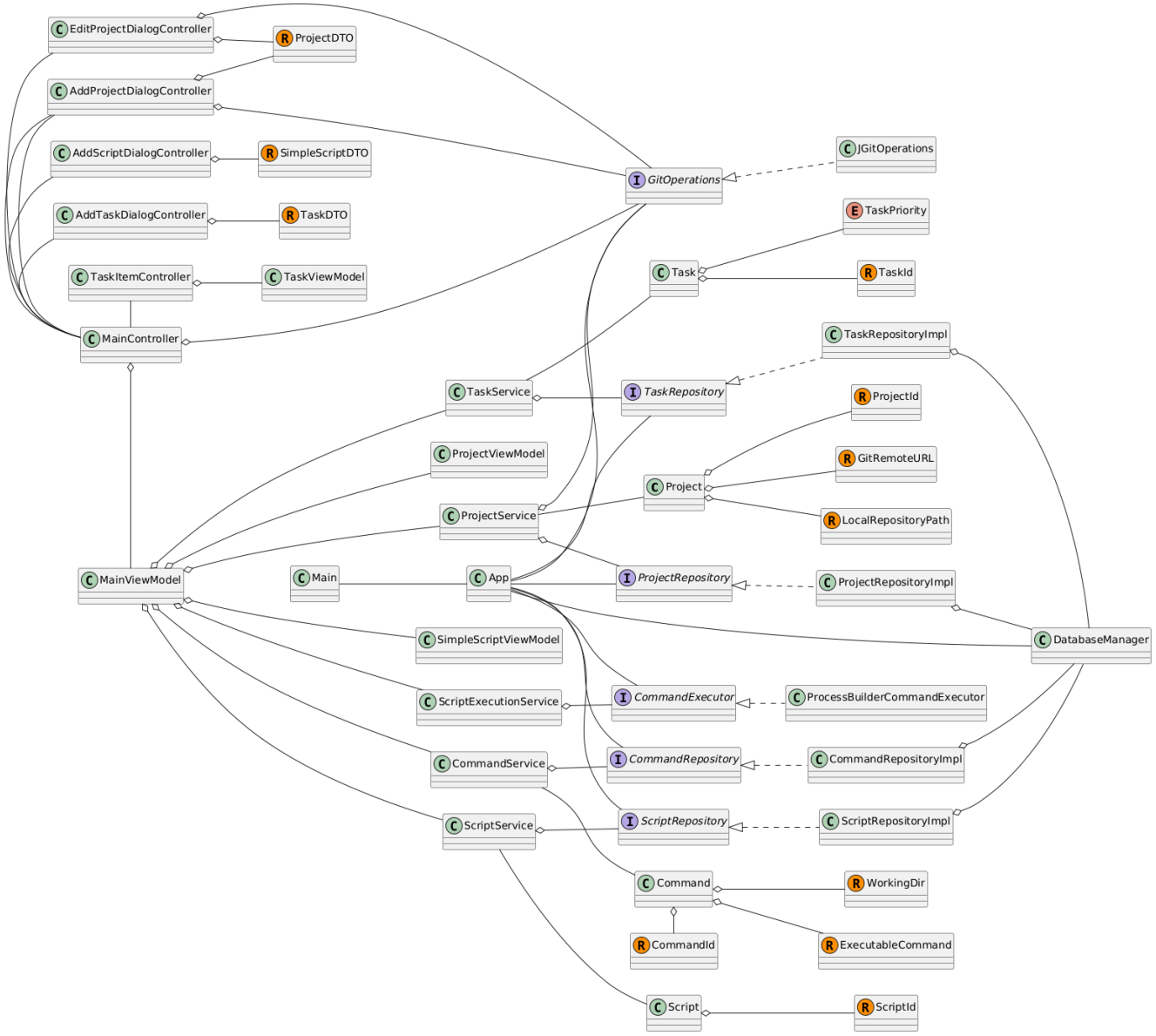

Рисунок 8 – UML диаграмма классов GitPM

2.2 Применение паттернов проектирования

В процессе разработки программных компонентов будем использовать известные «GoF» паттерны проектирования [6]. Рассмотрим некоторые используемые паттерны и опишем уместность их применения.

Для организации работы с базой данных (подключения, инициализации схемы) удобно организовать отдельный класс-менеджер. При этом важно, чтобы всякое обращение к базе данных шло именно к той единственной БД, в которой и будут храниться все данные программы. В целях предотвращения

множественности подключения и многократного пересоздания схемы нужно обеспечить единственность реализации менеджера. Т.е. у класса DatabaseManager должен быть единственный экземпляр. В этом случае правильным будет использование порождающего паттерна «Одиночка» (Singleton). Т.е. у класса будет статический метод для получения единственного экземпляра класса. При этом можно сделать его потокобезопасным (synchronized), чтобы избежать противоречия в транзакциях. В Листинге 1 представлен исходный код класса DatabaseManager, реализующего паттерн Singleton.

Листинг 1 – Исходный код класса DatabaseManager

```
public class DatabaseManager {
    private static DatabaseManager instance;
    private final Connection connection;

    private DatabaseManager(String path) {
        try {
            this.connection = DriverManager.getConnection("jdbc:sqlite:" +
path);
            initSchema();
        } catch (SQLException e) {
            throw new RuntimeException("Failed to connect to SQLite", e);
        }
    }

    public static synchronized DatabaseManager getInstance(String path) {
        if (instance == null) {
            instance = new DatabaseManager(path);
        }
        return instance;
    }

    public Connection getConnection() {
        return this.connection;
    }

    private void initSchema() {
        try (InputStream input = getClass().getResourceAsStream(
                "/ru/honsage/dev/gitpm/db/sqlite/schema.sql"
        )) {
            if (input == null) {
                throw new RuntimeException("Db schema not found in
resources");
            }

            String query = new String(input.readAllBytes(),
StandardCharsets.UTF_8);

            try (Statement st = connection.createStatement()) {
                st.executeUpdate(query);
            } catch (SQLException e) {
                throw new RuntimeException("DB error during query
```

```
execution", e);
                }
            } catch (IOException e) {
                throw new RuntimeException("Failed to load db schema", e);
            }
        }
}
```

Для удобства отладки и логирования в процессе разработки определим пользовательские классы исключений (Exceptions). Чтобы явно не инициализировать каждый из классов ошибок, определим фабрику, предоставляющую методы по созданию конкретных исключений. По сути, это соответствует порождающему паттерну «Фабричный метод» (Factory Method). Реализация паттерна в виде класса ExceptionFactory представлена в Листинге 2.

Листинг 2 – Исходный код класса ExceptionFactory

```
public final class ExceptionFactory {
    public static ValidationException validation(String message, String
context) {
        return new ValidationException(message, context);
    }

    public static BusinessRuleException businessRule(String message) {
        return new BusinessRuleException(message);
    }

    public static EntityNotFoundException entityNotFound(String entityName,
String key) {
        return new EntityNotFoundException(entityName, key);
    }
}
```

Так как наше приложение нацелено на работу с инфраструктурой Git было бы целесообразно использовать существующие интегрированные в Java инструменты по работе с Git. Поэтому воспользуемся внешней библиотекой JGit. Чтобы корректно встроить методы этой библиотеки в структуру нашего кода, создадим класс JGitOperations, который будет их использовать. При этом класс JGitOperations должен реализовывать контракт операций из домена. Т.е. JGitOperations использует сторонние библиотеки, но это скрыто от домена. Это похоже на структурный паттерн «Адаптер» (Adapter). Класс адаптирует стороннюю библиотеку под наш контракт для использования в наших сервисах. Исходный код класса представлен в Листинге 3.

Листинг 3 – Исходный код класса JGitOperations

```
//...
import org.eclipse.jgit.api.Git;

public class JGitOperations implements GitOperations {
    public JGitOperations() {}

    @Override
    public boolean isGitRepository(Path directory) {
        return Files.exists(directory.resolve(".git"));
    }

    @Override
    public List<Path> findGitRepositories(Path rootDirectory) {
        List<Path> found = new ArrayList<>();
        ExecutorService pool = Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()
        );
        try {
            Files.walk(rootDirectory, 5).forEach(path -> {
                pool.submit(() -> {
                    if (Files.isDirectory(path) &&
Files.exists(path.resolve(".git"))) {
                        synchronized (found) {
                            found.add(path);
                        }
                    }
                });
            });

            pool.shutdown();
            pool.awaitTermination(10, TimeUnit.SECONDS);
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }

        return found;
    }

    @Override
    public void cloneRepository(String url, Path directory) {
        try {
            Git.cloneRepository()
                    .setURI(url)
                    .setDirectory(directory.toFile())
                    .call();
        } catch (GitAPIException e) {
            throw new RuntimeException("Failed to clone repository: " + url,
e);
        }
    }

    @Override
    public String getRemoteURL(Path repository) {
        try {
            var repo = new FileRepositoryBuilder()
                    .setGitDir(repository.resolve(".git").toFile())
                    .build();

            StoredConfig config = repo.getConfig();
            String remote = config.getString("remote", "origin", "url");
            return this.convertRemoteToURL(remote);
```

```
        } catch (IOException e) {
            return null;
        }
    }

    private String convertRemoteToURL(String remote) {
        if (remote == null || remote.isBlank()) return remote;
        if (remote.startsWith("http://") || remote.startsWith("https://")) {
            return remote.replaceAll("\\.git$", "");
        }
        var sshPattern = Pattern.compile(
                "^git@([^:]+):(.+?)(?:\\.git)?$"
        );

        var m = sshPattern.matcher(remote);
        if (m.matches()) {
            String domain = m.group(1);
            String path = m.group(2);
            return "https://" + domain + "/" + path;
        }

        if (remote.startsWith("git://")) {
            return remote
                    .replaceFirst("^git://", "https://")
                    .replaceAll("\\.git$", "");
        }

        return remote.replaceAll("\\.git$", "");
    }
}
```

За логику представления данных в нашем проекте отвечает класс MainViewModel. Он агрегирует все сервисы из слоя Application и перенаправляет запросы контроллера на их методы. Соответственно, этот класс является неким посредником между моделями представления (viewmodels) и сервисами, перенаправляя и комбинируя результаты вызовов методов. Настройка большей части взаимосвязей в одном классе позволяет уменьшить связанность множества классов между собой. Это соответствует поведенческому паттерну «Посредник» (Mediator). Части реализации класса MainViewModel представлены в Листинге 4.

Листинг 4 – Часть исходного кода класса MainViewModel

```
public class MainViewModel {
    private final ProjectService projectService;
    private final TaskService taskService;
    private final ScriptService scriptService;
    private final CommandService commandService;
    private final ScriptExecutionService scriptExecutionService;

    private final ObservableList<ProjectViewModel> projects =
```

```
FXCollections.observableArrayList();
    private final FilteredList<ProjectViewModel> filteredProjects = new
FilteredList<>(projects);

    private final ObservableList<TaskViewModel> tasks =
FXCollections.observableArrayList();

    private final ObservableList<SimpleScriptViewModel> scripts =
FXCollections.observableArrayList();

    private ProjectViewModel selectedProject;
    private SimpleScriptViewModel selectedScript;

    public MainViewModel(
            ProjectService projectService,
            TaskService taskService,
            ScriptService scriptService,
            CommandService commandService,
            ScriptExecutionService scriptExecutionService
    ) {
        this.projectService = projectService;
        this.taskService = taskService;
        this.scriptService = scriptService;
        this.commandService = commandService;
        this.scriptExecutionService = scriptExecutionService;
    }

//...

    public void loadTasksForSelectedProject() {
        if (this.selectedProject == null) {
            tasks.clear();
            return;
        }

        ProjectId projectId = ProjectId.fromString(selectedProject.getId());
        this.tasks.clear();

        taskService.getAllTasks(projectId).stream()
                .map(TaskDTOMapper::toDTO)
                .map(TaskViewModel::new)
                .map(this::bindHandlersToTask)
                .forEach(tasks::add);
    }

//...

    public void runSelectedScript() {
        if (selectedScript == null) return;

        scriptExecutionService.runScript(
                selectedScript.getScriptId(),
                selectedScript.extractCommand()
        );

        selectedScript.setRunning(true);
    }

//...

}
```

С полной версией исходного кода программы можно ознакомиться в GitHub репозитории Автора: https://github.com/Honsage/GitPM.

2.3 Разработка базы данных

Реализуем ранее спроектированную схему БД. Для этого напишем запрос на создание таблиц и связей между ними. Учтем специфику SQLite: данная СУБД поддерживает небольшое количество типов данных. Поэтому большая часть полей будет храниться в текстовом формате. SQL-запрос на создание схемы БД представлен в Листинге 5.

Листинг 5 – SQL-запрос на создание схемы БД в СУБД SQLite

```
CREATE TABLE IF NOT EXISTS project (
    id_project TEXT PRIMARY KEY,
    title TEXT NOT NULL,
    description TEXT,
    local_path TEXT NOT NULL UNIQUE,
    remote_url TEXT,
    added_at TEXT NOT NULL
);

CREATE TABLE IF NOT EXISTS task (
    id_task TEXT PRIMARY KEY,
    id_project TEXT NOT NULL,
    title TEXT NOT NULL,
    content TEXT,
    created_at TEXT NOT NULL,
    is_completed INTEGER NOT NULL,
    deadline_at TEXT,
    priority TEXT NOT NULL,

    FOREIGN KEY (id_project) REFERENCES project(id_project) ON DELETE
CASCADE
);

CREATE TABLE IF NOT EXISTS script (
    id_script TEXT PRIMARY KEY,
    id_project TEXT NOT NULL,
    title TEXT NOT NULL,
    description TEXT,

    FOREIGN KEY (id_project) REFERENCES project(id_project) ON DELETE
CASCADE
);

CREATE TABLE IF NOT EXISTS command (
    id_command TEXT PRIMARY KEY,
    id_script TEXT NOT NULL,
    working_dir TEXT NOT NULL,
    executable_command TEXT NOT NULL,
    seq_order INTEGER NOT NULL,
```

```
    FOREIGN KEY (id_script) REFERENCES script(id_script) ON DELETE CASCADE
);
```

Данную схему будет инициализировать менеджер базы данных (о нем говорилось ранее). А отправка запросов к БД будет осуществляться внутри репозиториев. Для каждой интересующей нас выборки данных определим свой метод в соответствующем репозитории. Пример взаимодействия с БД из репозитория представлен в Листинге 6.

Листинг 6 – Пример работы с БД в коде программы

```
@Override
public List<Project> findByTitlePrefix(String titlePrefix) {
    List<Project> list = new ArrayList<>();
    String query = "SELECT * FROM project WHERE title LIKE ?;";
    try (PreparedStatement st = db.getConnection().prepareStatement(query))
{
        st.setString(1, titlePrefix + "%");

        ResultSet rs = st.executeQuery();
        while (rs.next()) {

list.add(ProjectEntityMapper.toDomain(ProjectEntityMapper.fromResultSet(rs))
);
        }
    } catch (SQLException e) {
        throw new RuntimeException("DB error during query execution", e);
    }
    return list;
}
```

2.4 Разработка пользовательского интерфейса

Перейдем к описанию разработки пользовательского интерфейса. Структура проекта на Java предполагает разумное отделение статических файлов (разметка, стили, изображения, запросы) от файлов с кодом. Поэтому в директории проекта есть отдельный пакет resources, в котором собраны статические файлы программы. Туда и поместим разметку оконного интерфейса в формате fxml и стили css (см. рис. 9).

Для работы с FXML разметкой есть специальный графический инструмент – SceneBuilder. Он позволяет в упрощенной форме конструировать интерфейс окон при помощи иерархии элементов. Реализуем

там структуру пользовательского интерфейса в соответствии с UI-дизайном. Внешний вид структуры главного окна в SceneBuilder представлен на рис. 10.

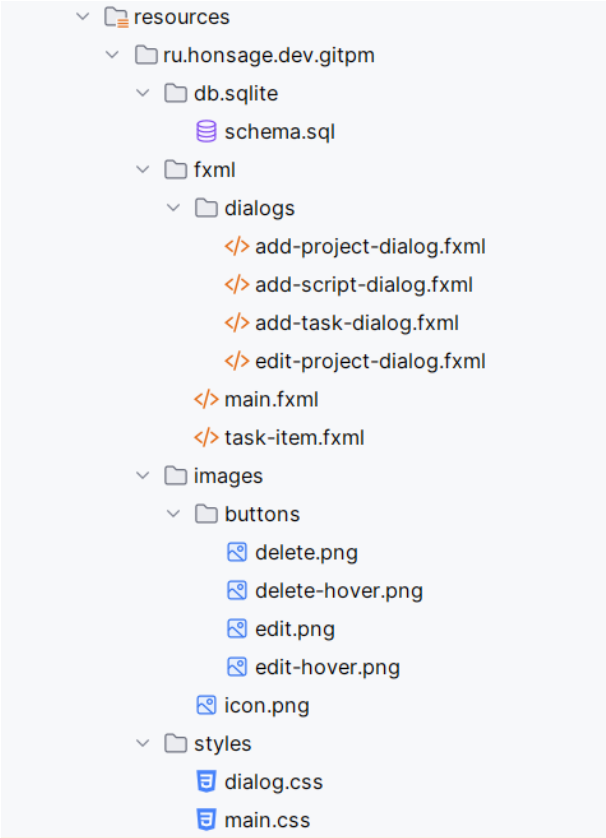

Рисунок 9 – Организация ресурсов (статических файлов) программы

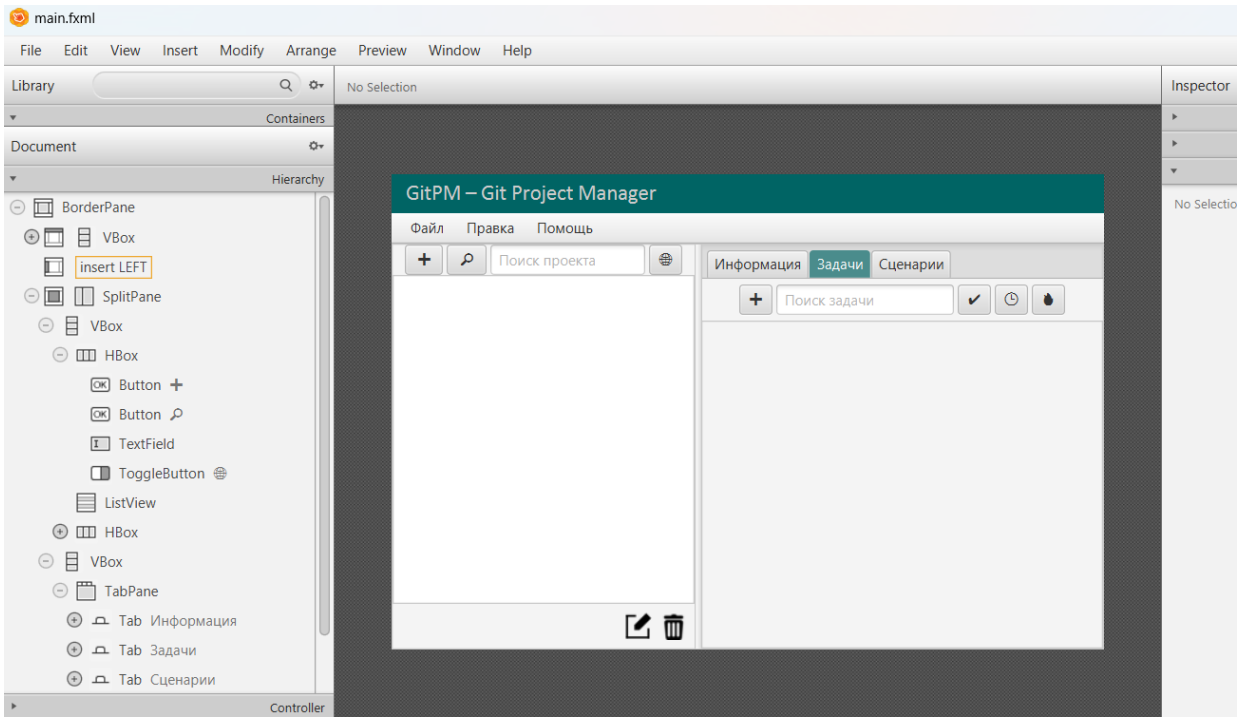

Рисунок 10 – Структура главного окна приложения в SceneBuilder

При помощи данного инструмента сверстаем и все остальные элементы пользовательского интерфейса. На рис. 11 представлен пример структуры диалогового окна.

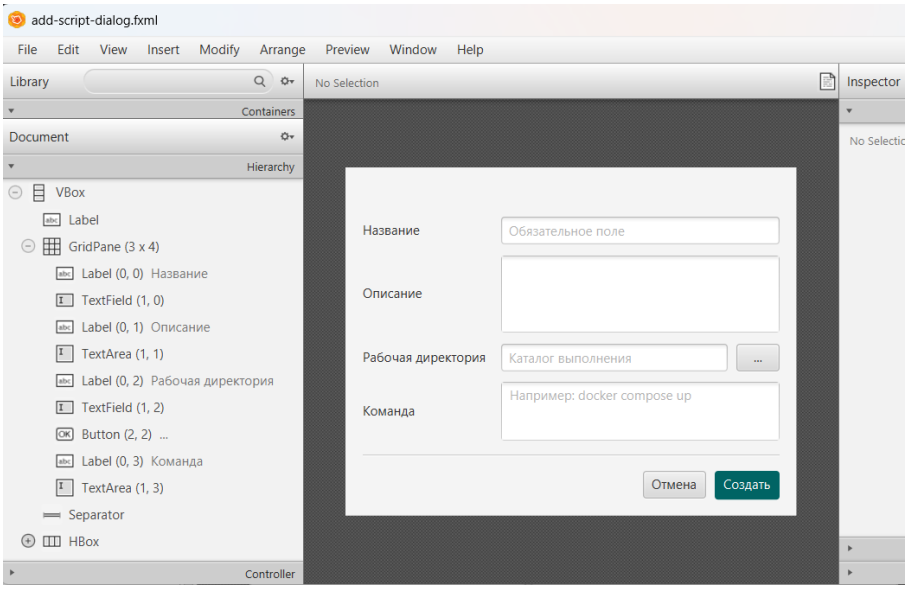

Рисунок 11 – Структура диалогового окна в SceneBuilder

Макет из SceneBuilder автоматически конвертируется в FXML-разметку. Таким образом, получили верстку всех необходимых компонентов UI. На рис. 12 представлен вид FXML верстки интерфейса.

```
20  <?import javafx.scene.layout.VBox?>
21  <?import javafx.scene.text.Font?>
22
23  <BorderPane fx:id="root" maxHeight="-Infinity" maxWidth="-Infinity"
24              minHeight="100.0" minWidth="200.0" prefHeight="400.0" prefWidth="600.0"
25              stylesheets="@../styles/main.css"
26              xmlns="http://javafx.com/javafx/23.0.1"
27              xmlns:fx="http://javafx.com/fxml/1"
28              fx:controller="ru.honsage.dev.gitpm.presentation.controllers.MainController">
29
30      <top>
31          <VBox fx:id="header" prefHeight="58.0" prefWidth="600.0" BorderPane.alignment="CENTER">
32              <HBox id="header-pane" fx:id="headerPane" prefHeight="33.0" prefWidth="600.0">
33                  <Label id="header__title" fx:id="titleLabel" prefHeight="33.0" prefWidth="608.0"
34                         text="GitPM - Git Project Manager" textFill="#cccccc" HBox.hgrow="ALWAYS">
35                      <font><Font name="Calibri" size="18.0" /></font>
36                      <padding><Insets left="12.0" /></padding>
37                  </Label>
38              </HBox>
39              <MenuBar id="menu-bar" fx:id="menuBar">
40                  <Menu fx:id="fileMenu" mnemonicParsing="false" styleClass="menu" text="Файл">
41                      <MenuItem mnemonicParsing="false" onAction="#onCloseApplication" styleClass="m
```

Рисунок 12 – Пример FXML разметки интерфейса

Для обеспечения стилизации верстки согласно нашему дизайну опишем в CSS файле стили компонентов. На рис. 13 представлен пример описания таких стилей.

```
#root {
    -fx-red-priority: #f0506e;
    -fx-yellow-priority: #f6c132;
    -fx-green-priority: #31c48d;
    -fx-main-color: #006464;
    -fx-transparent-main-color: #006464AA;
}

.approve-button,
.common-button {
    -fx-background-color: -fx-main-color;
    -fx-text-fill: white;
}

.approve-button:hover,
.common-button:hover {
    -fx-background-color: -fx-transparent-main-color;
}

.edit-button {
    -fx-max-width: 24px;
    -fx-background-color: transparent;
    -fx-background-image: url("../images/buttons/edit.png");
    -fx-background-repeat: no-repeat;
}
```

Рисунок 13 – Пример CSS стилей для интерфейса

2.5 Разработка тестов

В целях проверки корректности реализованной функциональности разработаем модульные тесты. Будем тестировать логику работы приложения. Тогда как проверку интерфейса оставим для ручного тестирования.

Модульное тестирование будем осуществлять при помощи фреймворка JUnit. Для тестирования напишем классы, методами которых и будут являться функциональные тесты.

Начнем тестирование со слоя Домена. Протестируем корректность валидации данных, так как от этого зависит правильность обработки вводимых данных. Основной упор сделаем на проверку value objects через

конкретные пограничные примеры. В Листинге 7 представлен исходный код класса для тестирования value object GitRemoteURL.

Листинг 7 – Тестирование value object GitRemoteURL

```
class GitRemoteURLTest {

    @Test
    void shouldAcceptValidGitHubUrl() {
        GitRemoteURL url = new GitRemoteURL("https://github.com/user/repo");

        assertEquals("https://github.com/user/repo", url.value());
    }

    @Test
    void shouldAcceptValidGitLabUrlWithoutProtocol() {
        GitRemoteURL url = new GitRemoteURL("gitlab.com/user/repo");

        assertEquals("gitlab.com/user/repo", url.value());
    }

    @Test
    void shouldTrimSpaces() {
        GitRemoteURL url = new GitRemoteURL("  https://github.com/user/repo
");

        assertEquals("https://github.com/user/repo", url.value());
    }

    @Test
    void shouldThrowExceptionForInvalidUrl() {
        assertThrows(RuntimeException.class,
                () -> new GitRemoteURL("https://google.com/user/repo"));
    }

    @Test
    void shouldThrowExceptionForEmptyString() {
        assertThrows(RuntimeException.class,
                () -> new GitRemoteURL("   "));
    }

    @Test
    void shouldAllowNullValue() {
        GitRemoteURL url = new GitRemoteURL(null);

        assertNull(url.value());
    }
}
```

Помимо доменного уровня бизнес-логика содержится также и в сервисах. Поэтому отдельно напишем тесты для методов сервисов. В частности, на фейковых реализациях интерфейсов проверим корректность работы сервиса проектов: создание, обработку и получение проектов. Реализация тестирования сервиса представлена в Листинге 8.

Листинг 8 – Тестирование ProjectService

```
class ProjectServiceTest {

    private FakeProjectRepository repository;
    private FakeGitOperations git;
    private ProjectService service;
    private Path tempDir;

    @BeforeEach
    void setUp() throws Exception {
        repository = new FakeProjectRepository();
        git = new FakeGitOperations();
        service = new ProjectService(repository, git);

        tempDir = Files.createTempDirectory("gitpm-test");
    }

    @Test
    void createProject_success() {
        Project project = service.createProject(
                "Test Project",
                "Description",
                tempDir.toString(),
                null
        );

        assertNotNull(project.getId());
        assertEquals("Test Project", project.getTitle());
        assertEquals(tempDir.toAbsolutePath().normalize().toString(),
                project.getLocalPath().value());
    }

    @Test
    void createProject_failsIfNotGitRepository() {
        git.setGitRepository(false);

        assertThrows(BusinessRuleException.class, () ->
                service.createProject(
                        "Bad Project",
                        null,
                        tempDir.toString(),
                        null
                )
        );
    }

    @Test
    void getProject_returnsSavedProject() {
        Project created = service.createProject(
                "My Project",
                null,
                tempDir.toString(),
                null
        );

        Project found = service.getProject(created.getId());

        assertEquals(created.getId(), found.getId());
    }

    @Test
    void getAllProjects_returnsAll() {
```

```
        service.createProject("One", null, tempDir.toString(), null);

        Path dir2;
        try {
            dir2 = Files.createTempDirectory("gitpm-test-2");
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        service.createProject("Two", null, dir2.toString(), null);

        assertEquals(2, service.getAllProjects().size());
    }
}
```

Проверим корректность программы при помощи модульных тестов. Запустим их и посмотрим на отчет о выполнении. Отчет свидетельствует об успешном прохождении тестов (см. рис. 14). Т.е. программу можно признать функционально корректной.

```
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running ru.honsage.dev.gitpm.application.services.ProjectServiceTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.139 s --
[INFO] Running ru.honsage.dev.gitpm.domain.exceptions.ExceptionFactoryTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.006 s --
[INFO] Running ru.honsage.dev.gitpm.domain.valueobjects.ExecutableCommandTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s --
[INFO] Running ru.honsage.dev.gitpm.domain.valueobjects.GitRemoteURLTest
[INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s --
[INFO] Running ru.honsage.dev.gitpm.domain.valueobjects.LocalRepositoryPathTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s --
[INFO] Running ru.honsage.dev.gitpm.domain.valueobjects.WorkingDirTest
[INFO] Tests run: 5, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s --
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 26, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.872 s
[INFO] Finished at: 2025-12-19T20:54:58+03:00
[INFO] ------------------------------------------------------------------------
PS D:\Projects\GitPM\GitPM>
```

Рисунок 14 – Отчет об успешном выполнении тестов

2.6 Рефакторинг и презентация работы продукта

Для поддержания качества кода произведем его рефакторинг [7].

В частности, будем придерживаться единого стиля кода. В рамках данного проекта был выбран функциональный стиль работы с коллекциями. Осуществим его при помощи встроенного в новые версии Java Stream API. Примеры обработки коллекций в функциональном стиле представлены в Листинге 9.

Листинг 9 – Примеры обработки коллекций в функциональном стиле

```
taskService.getAllTasks(projectId).stream()
        .map(TaskDTOMapper::toDTO)
        .map(TaskViewModel::new)
        .map(this::bindHandlersToTask)
        .forEach(tasks::add);

//...

IntStream.range(0, projects.size())
        .filter(i -> projects.get(i).getId().equals(dto.id()))
        .findFirst()
        .ifPresent(i -> projects.set(i, new ProjectViewModel(dto)));
```

В процессе разработки уделялось время рефакторингу в целях соблюдения принципов Чистого кода (Clean Code). Так, длинные методы разбивались на более мелкие, формируя самодокументирующие компоненты. Использование чистой архитектуры с разделением кода на модули также способствовало соблюдениям принципов DRY (don’t repeat yourself).

Теперь наглядно посмотрим на результат разработки продукта. Запустим его и проверим основные пользовательские сценарии. При запуске приложения видим изначальное представление главного окна (см. рис. 15). Баннер в правой части предлагает выбрать проект. Но для этого его сначала нужно создать или отсканировать. Воспользуемся функциональной панелью в левой верхней части окна. Иконки кнопок говорят сами за себя. Сканируем локальные репозитории по нажатию на кнопку со значком лупы. Предоставляется возможность выбора корневой директории для сканирования. После сканирования список найденных проектов отображается в левой части окна. А в правую часть подгружается информация о проектах (см. рис. 16).

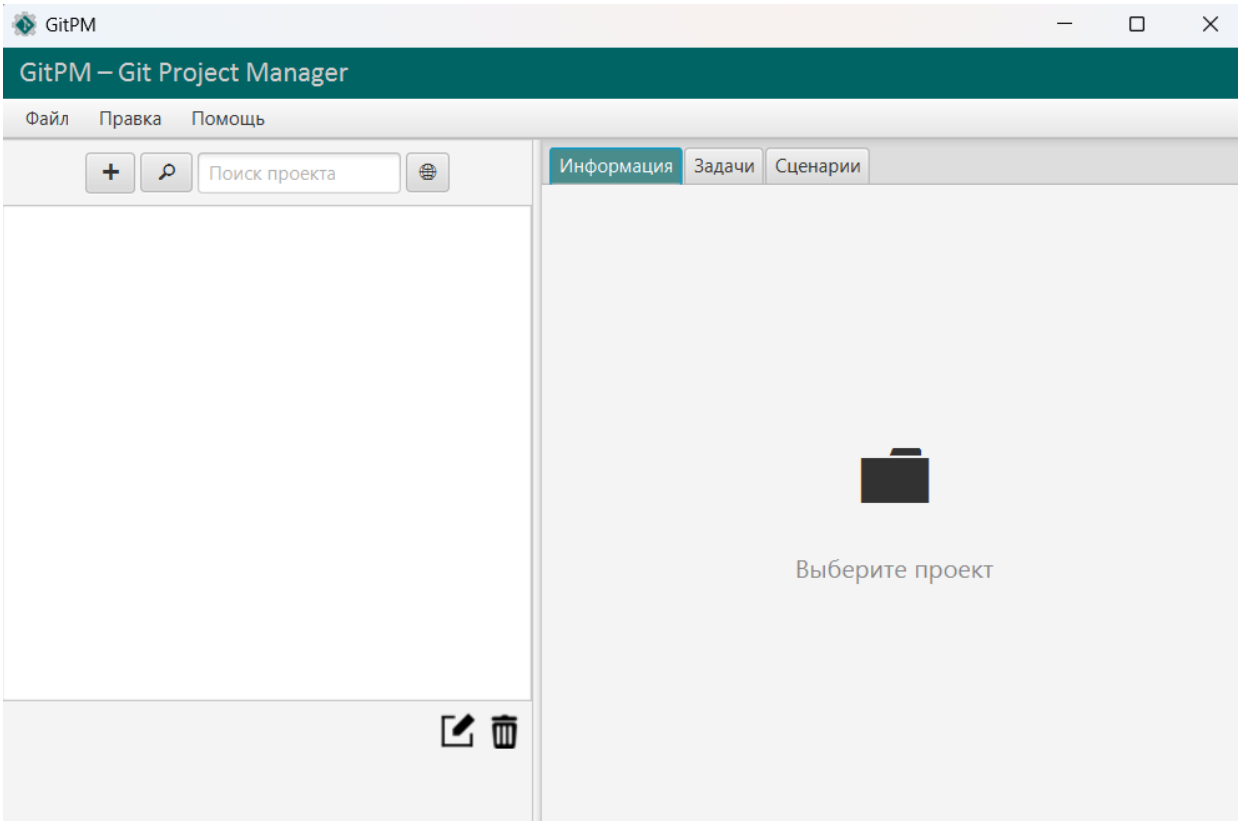

Рисунок 15 – Скриншот изначального состояния ПО

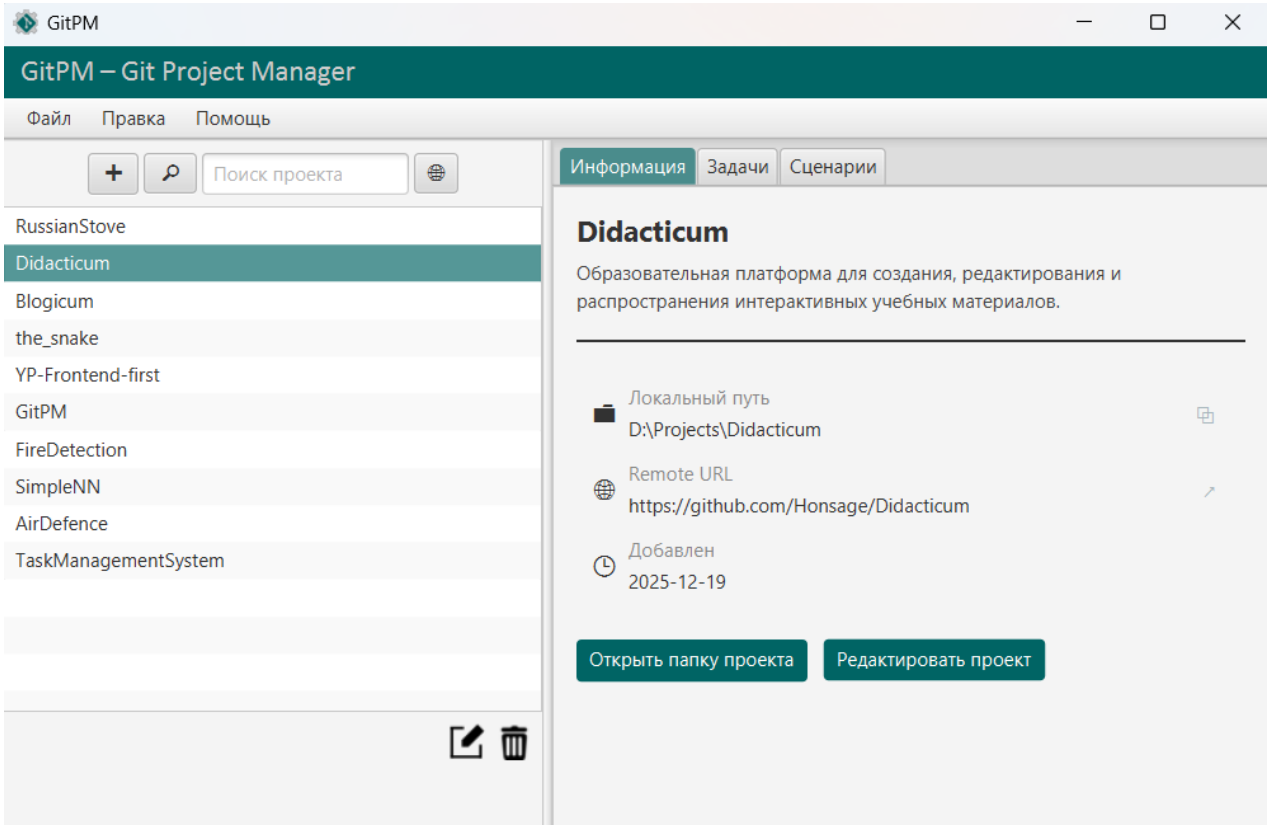

Рисунок 16 – Скриншот вкладки «Информация»

Выбрав вкладку «Задачи», можем перейти к планированию задач разработки. Здесь можно создавать и удалять задачи. Помечать их выполненными и устанавливать приоритет, который соответствует зеленому, желтому и красному цветам.

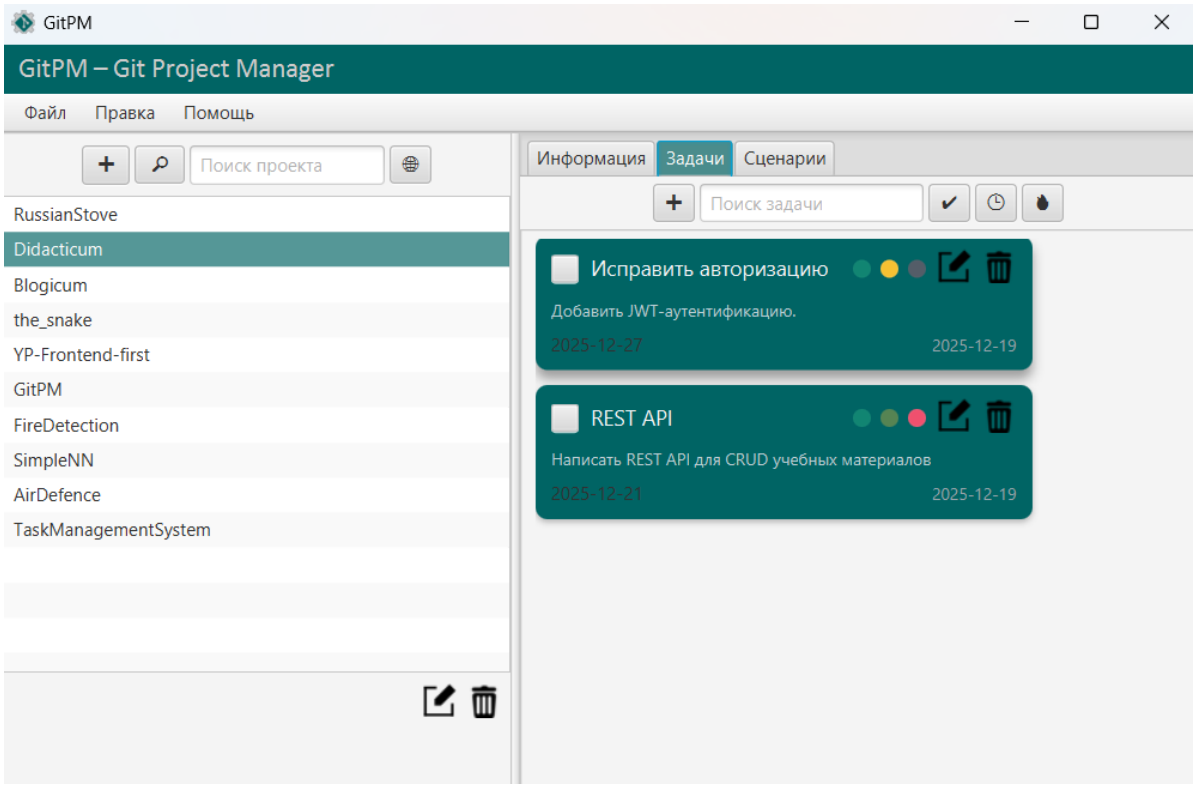

Рисунок 17 – Скриншот вкладки «Задачи»

Пожалуй, наиболее функционально полезная вкладка – «Сценарии». Здесь можно создать и настроить пользовательские сценарии взаимодействия с Git проектом. Например, можно настроить открытие проекта в среде разработки, запуск сервера, запуск тестов, открытие браузера – любые команды, которые рутинно выполняются в командной строке. Скриншот данной вкладки представлен на рис. 18.

У разрабатываемого приложения есть и перспективы развития. Помимо уже реализованной функциональности можно расширить интеграцию с Git и удаленными сервисами контроля версий. Например, добавить визуализацию дерева коммитов или интерфейс для взаимодействия с удаленными репозиториями. Правильно выстроенная архитектура и соблюдение

принципов чистого кода стали основой масштабируемости и гибкости проекта.

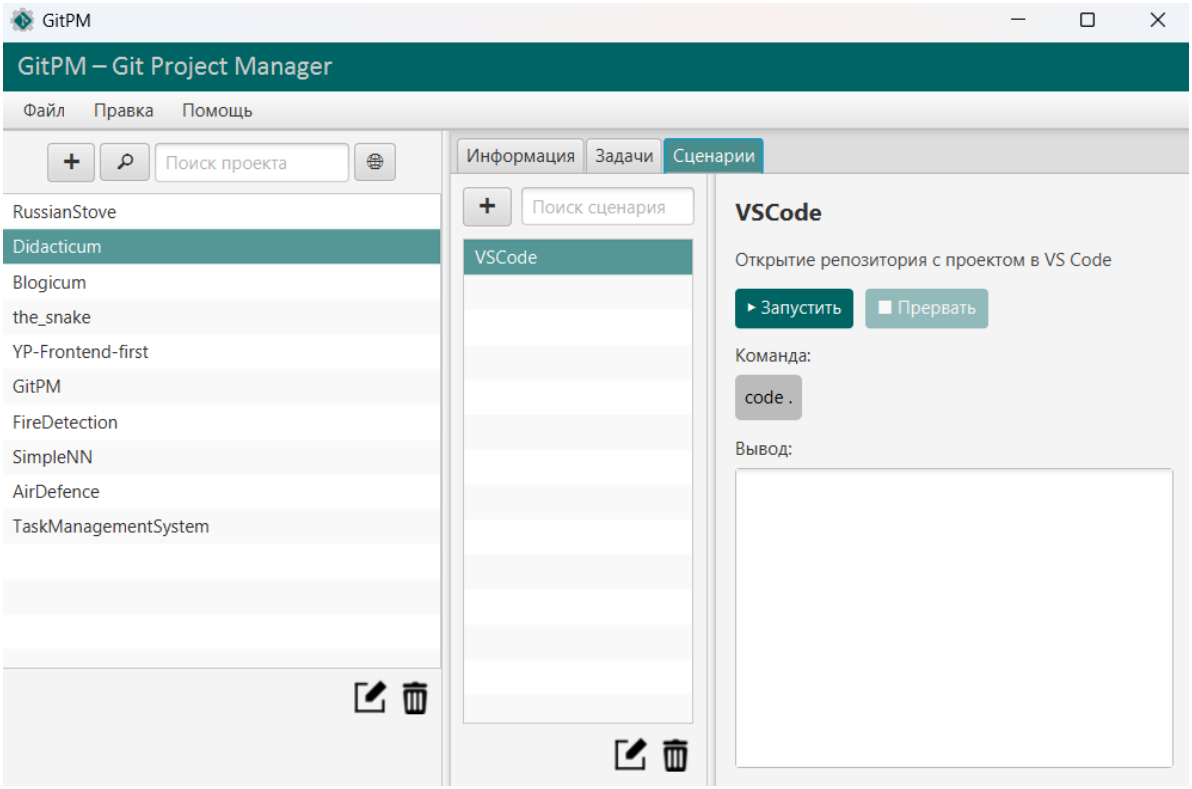

Рисунок 18 – Скриншот вкладки «Сценарии»

3 ДОКУМЕНТИРОВАНИЕ И ВЕРИФИКАЦИЯ

3.1 Верификация программного обеспечения

Проведем верификацию реализованного ПО. Проверим соответствие продукта требованиям, изложенным в Техническом задании.

Рассмотрим каждый пункт функциональных требований и опишем уровень соответствия продукта.

Добавление и редактирование информации о локальных репозиториях: ПП реализует функции добавления проекта (вручную и через сканирование), просмотра, а также редактирования информации. Функциональность реализуется посредством соответствующих кнопок графического интерфейса.

Комплексная фильтрация данных для поиска: в интерфейсе реализована строка поиска по названию проекта, которая фильтрует список в режиме реального времени. Помимо этого, есть переключаемая кнопка, позволяющая фильтровать проекты по наличию удаленных репозиториев.

Централизованное хранение данных обо всех Git репозиториях: для хранения информации о репозиториях используется единая встраиваемая база данных SQLite.

Предоставление возможностей добавления, редактирования и выполнения задач разработки: взаимодействие с задачами разработки представлено в отдельной вкладке графического интерфейса продукта. Обработка производится корректно.

Создание и настройка скриптов автоматизации: полноценное взаимодействие со скриптами также реализовано в соответствующей вкладке приложения. Поддерживается добавление, настройка и отображение скриптов.

Выполнение и прерывание скриптов автоматизации: в графическом интерфейсе реализованы кнопки, позволяющие запускать и прерывать процесс выполнения команд.

Таким образом, все пункты функциональных требований соблюдены и реализованы в полной мере.

Проверим теперь соответствие ПП критериям приемки. Для этого сформируем сводную таблицу (см. таблицу 1), где укажем требования, фактическое состояние ПП и степень соответствия требованиям.

Таблица 1 – Соответствие ПП критериям приемки

Требование	**Фактическое состояние ПП**	**Соответствие (+/-)**
Пользовательский интерфейс понятен и нагляден	В интерфейсе используются общеупотребимые обозначения и информационные сообщения	+
Пользователь имеет возможность редактирования внесенной информации	После добавления информации у пользователя есть возможность ее редактирования	+
Реализуется валидация логики	Валидация логики предметной области реализуется на уровне value objects	+
Данные сортируются в соответствии с фильтрами	При применении к спискам данных фильтров данные сортируются в соответствии с ними	+
Модульные тесты успешно проходят	Все реализованные модульные тесты выполняются успешно	+

Скрипты автоматизации запускаются корректно	При нажатии на кнопку выполнения скрипта запускается системный процесс в соответствии с командой скрипта	+
Программа не «вылетает» при каких бы то ни было действиях пользователя	В процессе тестирования сценариев использования вылетаний не произошло; В коде поддерживается многоуровневая обработка исключений	+

Отдельно рассмотрим соответствие продукта условиям эксплуатации. Инсталлированное приложение занимает 37 МБ на диске. Это соответствует требованиям к аппаратным характеристикам устройств пользователей (до 40 МБ дисковой памяти).

В требованиях к поддержке операционных систем указаны ОС Windows и Linux. Для Windows реализован отдельный установщик (installer) программного продукта. Для Unix систем есть универсальный JAR-архив продукта, который можно запустить при наличии JVM на устройстве (об этом подробнее в Инструкции по сборке и установке – Приложение Б).

Таким образом, разработанный программный продукт «GitPM» в полной мере соответствует установленным в Техническом задании функциональным и нефункциональным требованиям. Критерии приемки выполнены. Все основные функции работоспособны и прошли проверку как автоматическими модульными тестами, так и ручным тестированием. Продукт готов к приемке.

3.2 Описание аппаратных и программных требований

Для успешной установки и стабильной работы программного продукта «GitPM» на компьютере пользователя должны быть соблюдены следующие минимальные требования к аппаратному обеспечению:

- любой современный процессор с архитектурой x86/x64 (достаточно одного ядра);
- 1 ГБ ОЗУ;
- 40 МБ свободного дискового пространства.

Представим также и программные требования. В соответствии с типом установки ПО (см. Инструкцию по сборке и установке) может быть три вида требований:

Для запуска установщика (инсталлятора) ПО никаких дополнительных компонентов не требуется.

Для запуска JAR-архива требуется JRE 21+ (Java Runtime Environment).

Для сборки и запуска исходного кода требуется JDK 21+ (Java Development Kit) и Maven 3.9.9+.

3.3 Руководство по эксплуатации

Для работы с программным продуктом «GitPM» пользователю необходимо выполнить его установку и ознакомиться с основными функциями. Полное и подробное описание этих процессов представлено в Приложениях к данному отчету.

В Приложении Б содержится пошаговая инструкция по сборке и установке программного продукта, а также представлены три варианта запуска программы.

Приложение В представляет собой Руководство пользователя, в котором детально описаны все элементы графического интерфейса, основные сценарии использования ПО, а также ожидаемые результаты этого использования.

Для корректной эксплуатации ПП рекомендуется последовательно изучить указанные документы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была выполнена разработка программного продукта, предназначенного для управления локальными Git-проектами. В рамках работы были изучены и применены на практике основные технологии разработки программного обеспечения: проектирование, реализация, тестирование, рефакторинг, ведение документации.

В соответствии с поставленной целью разработки была сформирована концепция программного продукта. На основе имеющихся требований к продукту было разработано техническое задание, а также спроектирована архитектура приложения и структура базы данных. При проектировании были применены принципы разделения ответственности и модульности, что позволило обеспечить масштабируемую и гибкую структуру кода.

В ходе программной реализации были разработаны основные функции управления проектами, задачами и сценариями автоматизации. Для повышения сопровождаемости и качества кода применялись паттерны проектирования.

С целью проверки корректности работы программного продукта были реализованы модульные тесты, позволившие выполнить верификацию ключевых компонентов системы. Результаты верификации показали корректное выполнение поставленного плана и соответствие продукта техническому заданию.

По итогам выполнения курсовой работы был получен программный продукт, соответствующий поставленной цели, а также подготовлен комплект сопроводительной документации, включающий техническое задание, инструкцию по сборке и установке и руководство пользователя. Полученные в ходе работы знания и навыки помогли в освоении ключевых принципов технологий разработки программного обеспечения.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. ГОСТ 34.602-2020 Межгосударственный стандарт. Информационные технологии. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы. Дата введения 2022-01-01. – Москва: Изд-во стандартов, 2021. – 12 с.

2. SQLite Documentation: [Электронный ресурс]. URL: https://www.sqlite.org/docs.html (дата обращения: 18.12.2025).

3. Getting Started with JavaFX: [Электронный ресурс]. URL: https://openjfx.io/openjfx-docs/ (дата обращения: 18.12.2025).

4. Maven Documentation – Maven: [Электронный ресурс]. URL: https://maven.apache.org/guides/index.html (дата обращения: 18.12.2025).

5. Мартин, Р. Чистая архитектура. Искусство разработки программного обеспечения / пер. с англ. – СПб.: Питер, 2018. – 352 с.

6. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб.: Питер, 2015. – 368 с.

7. Рефакторинг и Паттерны проектирования: [Электронный ресурс]. URL: https://refactoringguru.cn/ru (дата обращения: 19.12.2025).

8. ГОСТ 19.505-79 Межгосударственный стандарт. Единая система программной документации. Руководство оператора. Требования к содержанию и оформлению. Дата введения 1980-01-01. – Москва: Изд-во стандартов, 2010. – 3 с.

Приложение А. Техническое задание

Техническое задание на разработку программного обеспечения «GitPM»

1. Общие сведения о Программном Продукте

Настоящий Программный Продукт «GitPM – Git Project Manager» (далее – ПП) представляет собой автоматизированную систему для комплексного управления локальными Git проектами.

С архитектурной точки зрения, ПП представляет собой Desktop-приложение с графическим интерфейсом и базой данных.

Назначением данного ПП является учет сведений о локальных Git репозиториях, управление задачами разработки, а также создание и исполнение автоматизированных сценариев взаимодействия с проектами. Областью применимости ПП является разработка программного обеспечения, осуществляемая как частными лицами, так и в рамках IT-компаний.

2. Основания для разработки

Основанием для разработки данного ПП служит задание, постановка которого сформулирована в рамках «Задания на курсовую работу» учебного плана дисциплины «Технологии разработки программного обеспечения».

3. Назначение разработки

ПП предназначен для сбора информации о локальных репозиториях, создания и редактирования задач разработки, а также для настройки автоматизированных сценариев. Таким образом, объектом автоматизации является совокупность шаблонных действий по учету, планированию и обработке локальных проектов. Вид автоматизируемой деятельности – разработка ПО.

От ПП ожидается решение следующих задач:

– учет сведений о локальных Git репозиториях;

– централизованное хранение данных о репозиториях;

– планирование и фиксация выполнения задач разработки;

– настройка автоматизированных скриптов взаимодействия с программными решениями;

– автоматизация процессов сборки, запуска и тестирования ПО.

4. Требования к программе

В соответствии с целями разработки ПП определена следующая группа требований.

4.1. Функциональные требования

ПП должен корректно реализовывать следующие функции:

– добавление и редактирование информации о локальных репозиториях;

– комплексная фильтрация данных для поиска;

– сканирование директорий на наличие Git репозиториев;

– централизованное хранение данных обо всех Git репозиториях;

– предоставление возможностей добавления, редактирования и выполнения задач разработки;

– создание и настройка скриптов автоматизации;

– выполнение и прерывание скриптов автоматизации;

– обеспечение постоянного хранения данных.

4.2. Условия эксплуатации

ПП должно корректно работать на устройствах со следующими аппаратными характеристиками: минимум 1 ГБ ОЗУ и 40 МБ свободной памяти на диске.

Требования к поддержке операционных систем: ОС Windows (7+), ОС Linux.

4.3. Требования к совместимости

Для обеспечения совместимости ПП со сторонними сервисами требуется при необходимости обеспечить интеграцию с различными удаленными сервисами контроля версий.

Для корректного сопровождения и обеспечения обновлений ПО необходимо организовать совместимость с будущими версиями ПП путем формирования масштабируемой архитектуры и универсальной и целостной структуры хранения данных.

5. Требования к интерфейсу

В целях обеспечения удобства взаимодействия пользователя с ПП требуется разработать пользовательский интерфейс, удовлетворяющий следующим критериям:

– отзывчивость элементов управления путем изменения внешнего вида;

– поддержание единообразия дизайна и цветовой палитры;

– использование уникального логотипа и бренда продукта;

– наличие полноценного взаимодействия с пользователем в виде диалоговых окон и алертов;

– простота и использование общеупотребимых иконок и знаков;

– упорядоченность данных в соответствии с логикой их сортировки;

6. Критерии приемки

В качестве критериев соответствия ПП требованиям определены следующие параметры:

– пользовательский интерфейс интуитивно понятен и нагляден;

– пользователь имеет возможность редактирования информации;

– реализуется валидация логики взаимодействия с объектами ПО;

– при применении фильтров данные сортируются в соответствии с ними;

– все модульные тесты проходят успешно;

– скрипты автоматизации запускаются корректно;

– программа не «вылетает» при каких бы то ни было действиях пользователя.

7. Требования к документации

В перечень обязательных документов сопровождающих ПП входят Инструкция по сборке и установке и Руководство пользователя (см. соответствующие приложения).

8. Порядок контроля и приемки

Основным методом контрольного тестирования ПП является модульное тестирование программных компонентов. Помимо этого, также ожидается ручное тестирование пользовательского интерфейса и сценариев взаимодействия с ПП.

Приложение Б. Инструкция по сборке и установке

Инструкция по сборке и установке программного обеспечения «GitPM»

1. Общая информация

Данный документ содержит инструкции по сборке, установке и запуску программного продукта с открытым исходным кодом «GitPM – Git Project Manager». Продукт распространяется под лицензией свободного программного обеспечения MIT. ПО представляет собой кроссплатформенное desktop приложение с графическим интерфейсом и предназначено для управления локальными Git-репозиториями.

Исходный код продукта представлен в GitHub репозитории: https://github.com/Honsage/GitPM.

2. Подготовка системы

Перед установкой убедитесь, что Ваша система соответствует минимальным требованиям:

– ОС Windows 7+ или современные дистрибутивы Linux;

– не менее 70 МБ свободного места на диске;

– не менее 1 ГБ ОЗУ;

– наличие установленной системы контроля версий Git.

3. Способы установки и запуска

Существует три способа запуска приложения в зависимости от возможностей и целей пользователя.

3.1 Сборка из исходного кода (для разработчиков)

Способ предназначен для разработчиков, желающих расширить код или собрать актуальную версию программы самостоятельно.

Необходимое ПО:

– JDK (Java Development Kit) версии 21 или выше (можно скачать с официального сайта Oracle: https://www.oracle.com/java/technologies/downloads/);

– Apache Maven версии 3.9.9 или выше (доступно скачивание с официального сайта Maven https://maven.apache.org/download.cgi).

Пошаговая инструкция:

1. Сохраните исходный код проекта на свое устройство при помощи команды “git clone https://github.com/Honsage/GitPM.git”.

2. Выполните сборку проекта: из корневой директории проекта вызовите команду “mvn clean package”.

3. Найдите собранный JAR-файл: после сборки в папке target/ будет создан исполняемый JAR-файл со всеми зависимостями.

4. Запустите приложение: запустите JAR-файл командой “java -jar target/<имя_файла>.jar”.

3.2 Запуск из JAR-файла

Данный способ является рекомендуемым для большинства пользователей ввиду своей кроссплатформенности.

Необходимое ПО:

– JRE (Java Runtime Environment) версии 21 или выше (можно скачать с официального сайта Oracle: https://www.oracle.com/java/technologies/downloads/);

Убедитесь также, что команда “java” доступна из терминала или командной строки. Если команда недоступна, необходимо создать переменную среды JAVA_HOME.

Пошаговая инструкция:

1. Скачайте JAR-архив: перейдите во вкладку Releases в репозитории проекта (https://github.com/Honsage/GitPM/releases) и скачайте файл с расширением “.jar”.

2. Запустите приложение: запустите JAR-файл командой “java -jar <имя_файла>.jar”.

3.3 Установка через инсталлятор для Windows

Данный способ является самым простым для пользователей Windows.

Пошаговая инструкция:

1. Скачайте установщик: перейдите во вкладку Releases в репозитории проекта (https://github.com/Honsage/GitPM/releases) и скачайте файл установщика с расширением “.exe”.

2. Инсталлируйте приложение: двойным кликом по файлу установщика запустите процесс установки; следуйте инструкциям мастера установки.

3. Запустите приложение: после завершения установки запустите приложение GitPM через созданный ярлык или из меню «Пуск».

4. Проверка установки

При успешном запуске откроется главное окно приложения (Рисунок 1). Освойте интерфейс в соответствии с Руководством пользователя. Теперь приложение GitPM готово к использованию.

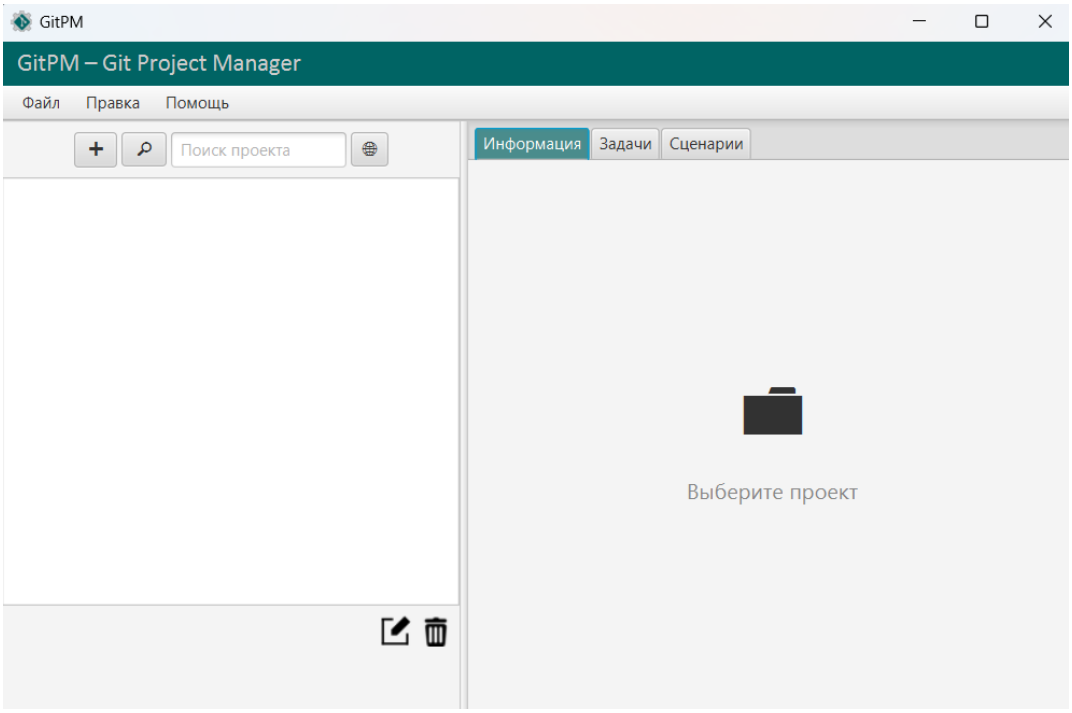

Рисунок 1 – Изначальный вид главного окна приложения GitPM

5. Дополнительная информация

При первом запуске в директории с инсталлированным проектом (или рядом с JAR файлом) создастся файл базы данных “gitpm.db”.

Для обновления приложения скачайте последнюю версию JAR/EXE файла из ранее указанного репозитория. Пользовательские данные, хранящиеся в базе данных, можно будет перенести в новую версию путем перемещения файла базы данных в директорию с инсталлированным проектом (или JAR-файлом).

Приложение В. Руководство пользователя

Руководство пользователя (оператора) программного обеспечения «GitPM»

1. Знакомство с GitPM

«GitPM – Git Project Manager» представляет собой desktop-утилиту с графическим интерфейсом. Приложение предназначено для управления локальными Git проектами.

Программа позволяет:

– найти и упорядочить все Git репозитории в одном окне;

– планировать задачи разработки по каждому проекту;

– автоматизировать рутинные действия посредством настройки пользовательских скриптов.

2. Первые шаги

Запустите приложение. Если Вы еще не добавляли никаких проектов в менеджер, то отобразится изначальный вид главного окна (Рисунок 1).

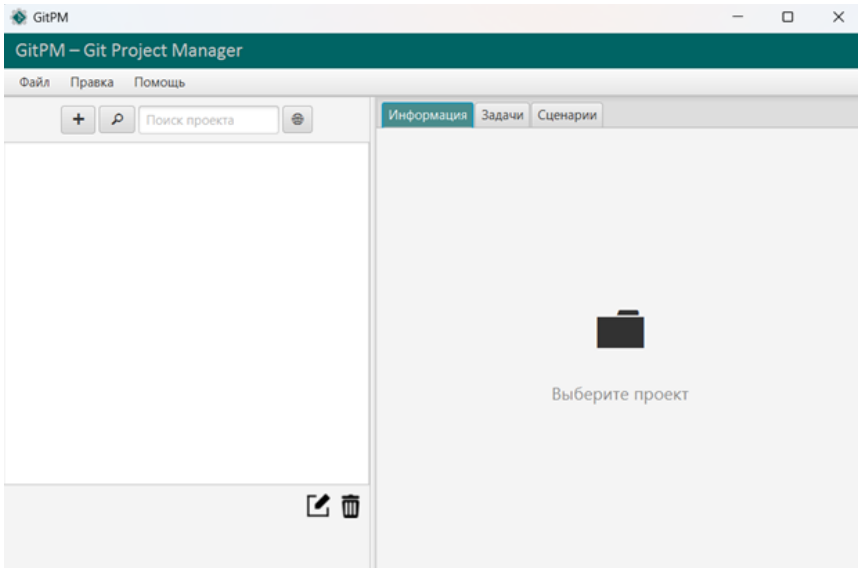

Рисунок 1 – Изначальный вид главного окна приложения GitPM

Интерфейс имеет следующую схему: в левой части окна отображается список добавленных Git-проектов, а правая часть отвечает за функциональное наполнение. Представлены три вкладки: «Информация», «Задачи», «Сценарии».

Во вкладке «Информация» при выбранном в списке проектов проекте отображаются основные сведения о проекте.

Во вкладке «Задачи» отображается список задач разработки, с которыми можно взаимодействовать.

Вкладка «Сценарии» отвечает за создание скриптов взаимодействия с проектом. Она состоит из списка созданных скриптов в левой части и из функционально-информационного наполнения в правой части. Первоначальный внешний вид вкладки представлен на Рисунке 2.

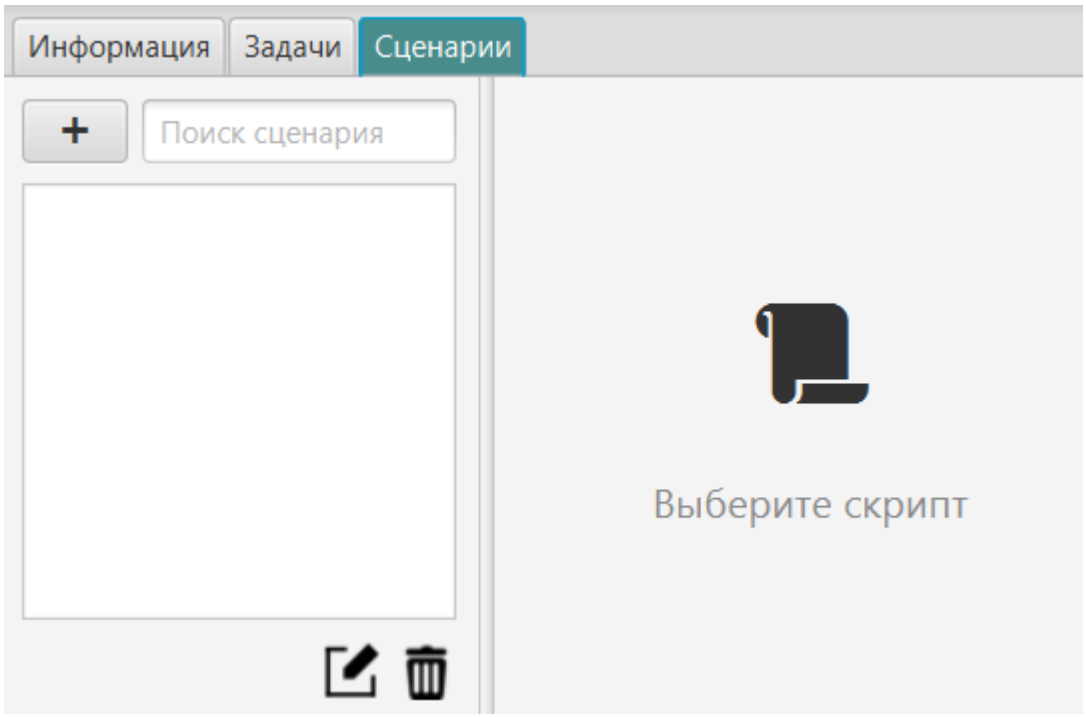

Рисунок 2 – Изначальный вид вкладки «Сценарии»

3. Работа с проектами

Вся функциональность работы с проектами расположена в левой части окна. В верхней части представлена панель с кнопками и строкой ввода (см.

Рисунок 3). Нажимая на кнопку со значком «плюс» откроется диалоговое окно добавления проекта (см. Рисунок 4).

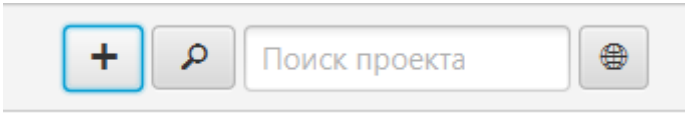

Рисунок 3 – Панель взаимодействия с проектами

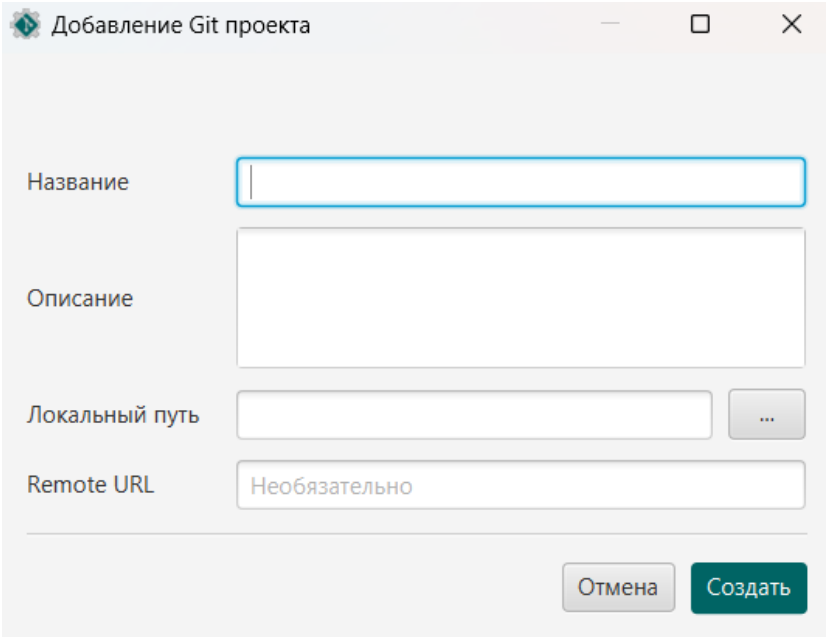

Рисунок 4 – Диалоговое окно добавления проекта

Указав корректные данные и подтвердив создание проекта, можно увидеть появившееся название нового проекта в списке проектов.

Редактировать информацию о проекте либо удалить его можно через кнопки, расположенные на нижней панели (см. Рисунок 5).

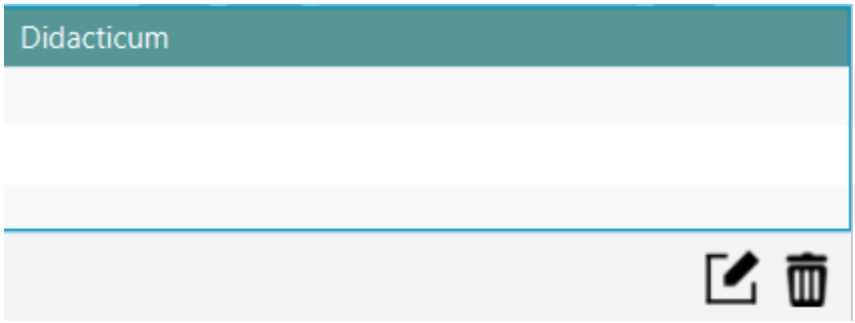

Рисунок 5 – Нижняя панель редактирования и удаления проектов

При выбранном в списке проектов проекте на вкладках будут отображаться соответствующие данные. В частности, на вкладке «Информация» отобразятся сведения о проекте (см. Рисунок 6).

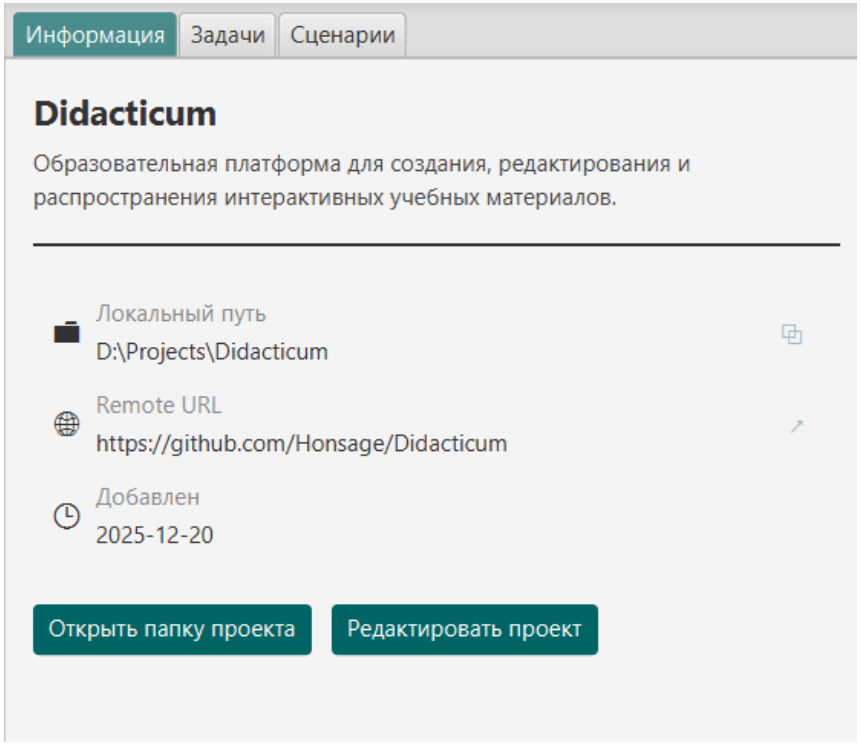

Рисунок 6 – Вкладка «Информация»

4. Управление задачами

Управление задачами осуществляется во вкладке «Задачи». Изначально отображается только панель взаимодействия с задачами (см. Рисунок 7). По нажатию на кнопку с «плюсом» можно добавить задачу.

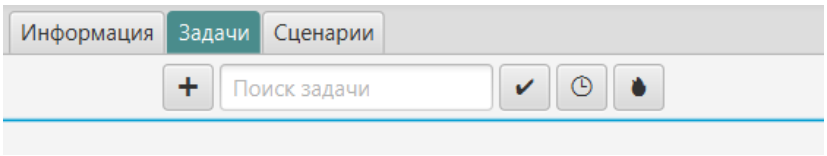

Рисунок 7 – Панель взаимодействия с задачами

При добавлении задачи откроется диалоговое окно (см. Рисунок 8). После введения информации в него и подтверждения создания задачи задача будет добавлена в отображаемый список.

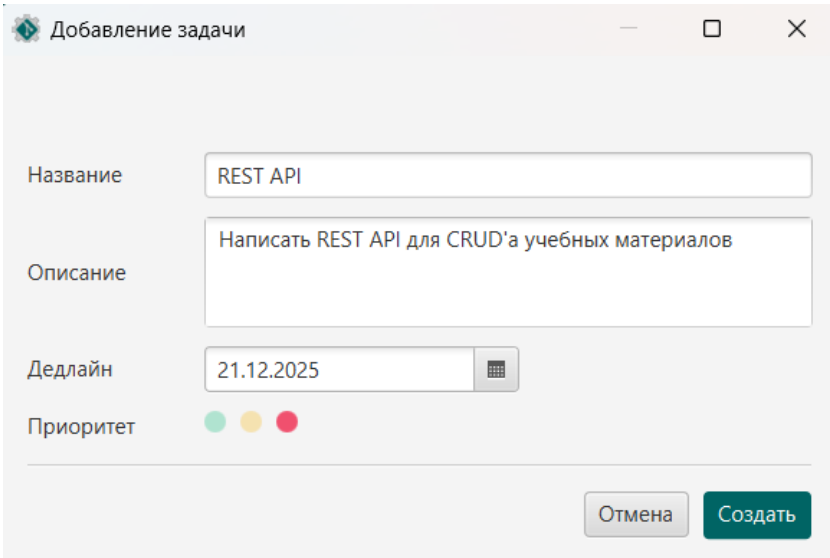

Рисунок 8 – Диалоговое окно добавления задачи

Можно взаимодействовать с задачами и непосредственно из списка задач. Например, отмечать их выполненными, изменять приоритет, редактировать или удалять (см. Рисунок 9).

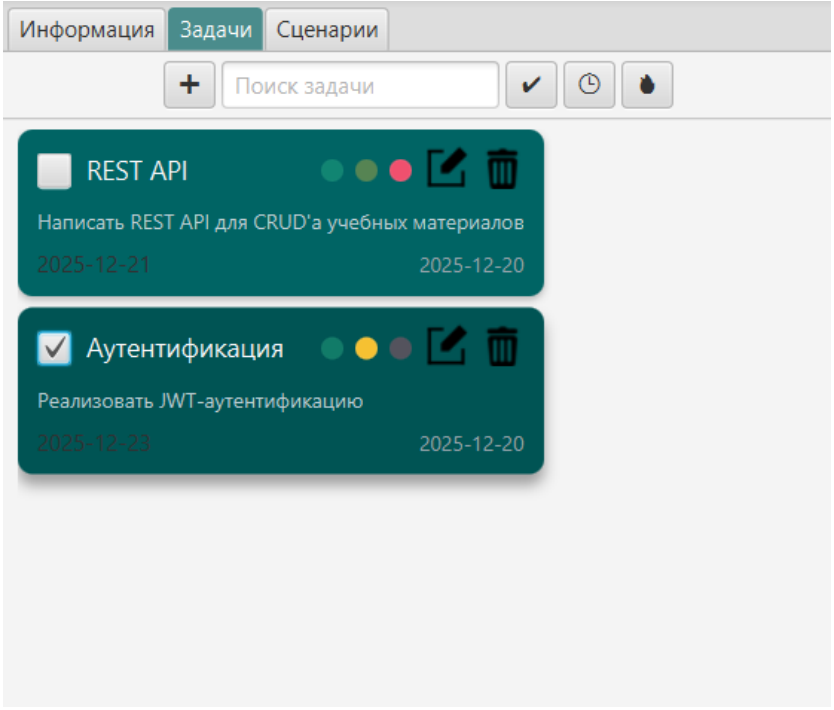

Рисунок 9 – Вкладка «Задачи»

5. Настройка сценариев

Работа со скриптами осуществляется во вкладке «Сценарии».

Новый скрипт можно добавить по нажатию на кнопку с ярлыком «плюс». Помимо этого представлена также строка поиска сценариев, кнопки редактирования и удаления (см. Рисунок 10).

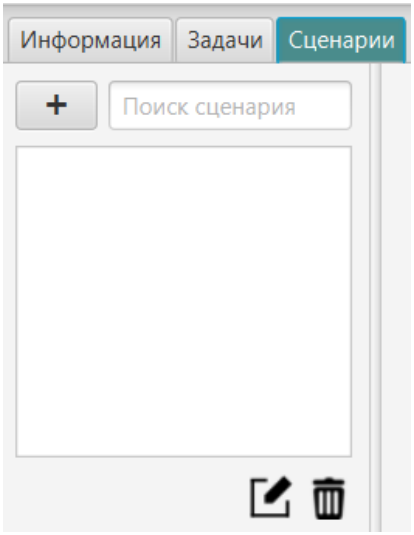

Рисунок 10 – Панель взаимодействия со скриптами

После того, как скрипт создан, его можно запустить, нажав на соответствующую кнопку. Впоследствии можно будет остановить выполнение скрипта кнопкой «Прервать» (см. Рисунок 11).

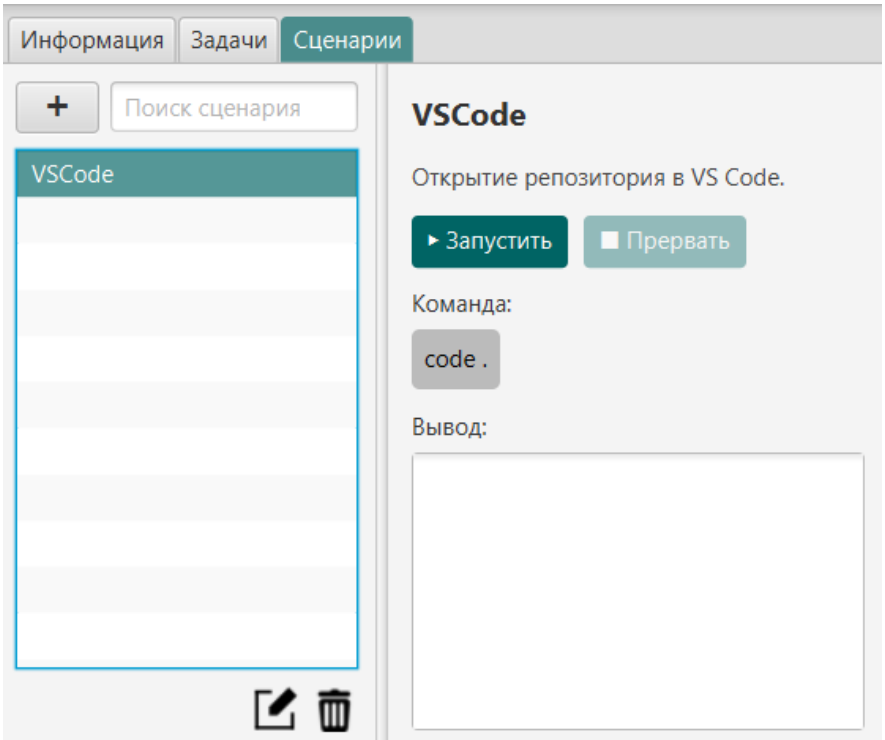

Рисунок 11 – Вкладка «Сценарии»

Команды могут быть совершенно различными. Главное, чтобы они могли корректно выполняться в командной строке Вашей операционной системы. Например, можно запустить бэкенд-сервер приложения (см. Рисунок 12).

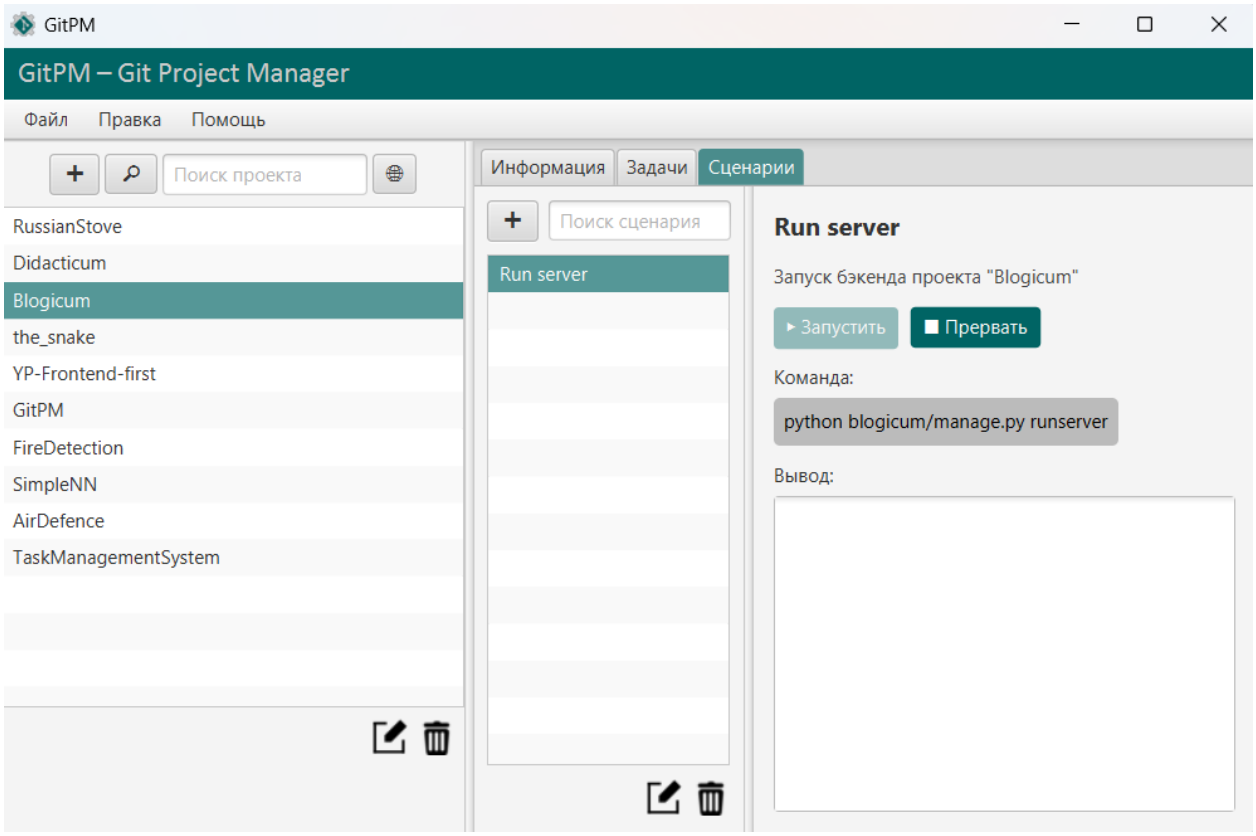

Рисунок 12 – Пример использования GitPM для запуска бэкенд-сервера