Практическая работа

Система управления библиотекой книг

1. Введение и цели работы

В рамках данной практической работы вы создадите консольное приложение на языке Java, реализующее базовую систему управления библиотекой книг. Параллельно вы освоите продвинутые возможности системы контроля версий Git, включая работу с ветками, перебазирование и выборочное применение коммитов.

1.1. Образовательные цели

- Закрепить навыки объектно-ориентированного программирования на Java
- Изучить концепцию вложенных (nested) и внутренних (inner) классов
- Освоить продвинутые команды Git: rebase, reset, cherry-pick
- Научиться работать с удалённым репозиторием на GitHub
- Понять принципы работы с историей коммитов и её модификации

2. Предварительные требования

2.1. Программное обеспечение

- JDK 17 или выше (рекомендуется OpenJDK или Amazon Corretto)
- Git версии 2.30 или выше
- Учётная запись на GitHub
- Текстовый редактор или IDE (рекомендуется IntelliJ IDEA Community Edition)

2.2. Необходимые знания

- Базовый синтаксис Java (переменные, циклы, условия, методы)
- Понятие класса и объекта в ООП
- Базовые команды Git (init, add, commit, push, pull)

3. Описание предметной области

Вам необходимо разработать систему управления библиотекой, которая позволяет хранить информацию о книгах, авторах и операциях выдачи/возврата книг. Система должна поддерживать следующие сущности:

Книга (Book)

Содержит информацию о названии, авторе, годе издания, ISBN и статусе доступности. Каждая книга имеет уникальный идентификатор и может находиться либо в библиотеке, либо быть выданной читателю.

Библиотека (Library)

Основной класс системы, хранящий коллекцию книг. Содержит вложенный класс для ведения журнала операций. Предоставляет методы для добавления книг, поиска, выдачи и возврата.

Журнал операций (OperationLog)

Вложенный статический класс внутри Library, отвечающий за запись всех операций: добавление книги, выдача, возврат. Каждая запись содержит тип операции, дату/время и описание.

4. Часть 1: Инициализация проекта и Git (30 минут)

4.1. Создание репозитория

1. Создайте новый публичный репозиторий на GitHub с названием library-management-system
2. Инициализируйте репозиторий с файлом README.md
3. Склонируйте репозиторий на локальную машину
4. Создайте структуру проекта:

```
library-management-system/
├── src/
│   └── main/
│       └── java/
│           └── library/
│               ├── Book.java
│               ├── Library.java
│               └── Main.java
├── README.md
└── .gitignore
```

4.2. Настройка .gitignore

Создайте файл .gitignore со следующим содержимым:

```
# Compiled class files
*.class

# IDE files
.idea/
*.iml
.vscode/

# Build output
out/
```

1. Сделайте первый коммит с сообщением: "Initial project structure"

2. Отправьте изменения на GitHub (git push)

5. Часть 2: Реализация Java-классов (60 минут)

5.1. Класс Book

Создайте ветку feature/book-class и переключитесь на неё:

```
git checkout -b feature/book-class
```

Реализуйте класс Book со следующими требованиями:

Поля класса (private):

• id (int) — уникальный идентификатор книги
• title (String) — название книги
• author (String) — автор книги
• year (int) — год издания
• isbn (String) — международный стандартный книжный номер
• available (boolean) — флаг доступности (true = в библиотеке)

Методы:

• Конструктор с параметрами (id, title, author, year, isbn)
• Геттеры для всех полей
• setAvailable(boolean) — для изменения статуса доступности
• toString() — переопределённый метод для красивого вывода информации о книге

Пример вывода toString():

```
[ID: 1] "Война и мир" — Л.Н. Толстой (1869)
ISBN: 978-5-17-090335-2 | Статус: Доступна
```

После реализации:

```
git add .
git commit -m "Add Book class with basic fields and methods"
```

5.2. Класс Library со вложенным классом

Создайте новую ветку от main:

```
git checkout main
git checkout -b feature/library-class
```

Важно! Обратите внимание: вы создаёте ветку от main, а не от feature/book-class. Это намеренно сделано для последующей практики с командами Git.

Структура класса Library:

```
public class Library {
    private List<Book> books;
    private OperationLog operationLog;
```

```
    // Вложенный статический класс
    public static class OperationLog {
        // Внутренний класс для записи операции
        public class LogEntry {
            private OperationType type;
            private LocalDateTime timestamp;
            private String description;
            // конструктор, геттеры, toString()
        }

        public enum OperationType {
            ADD_BOOK, BORROW, RETURN
        }

        private List<LogEntry> entries;
        // методы: addEntry(), getEntries(), printLog()
    }
}
```

Требования к классу Library:

• Конструктор инициализирует пустой список книг и журнал операций
• addBook(Book book) — добавляет книгу и записывает операцию в журнал
• findBookById(int id) — возвращает книгу по ID или null
• findBooksByAuthor(String author) — возвращает список книг автора
• borrowBook(int id) — выдаёт книгу (меняет available на false)
• returnBook(int id) — принимает книгу обратно
• getAvailableBooks() — возвращает список доступных книг
• printOperationLog() — выводит журнал всех операций

После реализации сделайте ДВА коммита:

```
# Первый коммит – только структура класса Library
git commit -m "Add Library class structure"

# Второй коммит – вложенный класс OperationLog
git commit -m "Add OperationLog nested class with LogEntry"
```

5.3. Класс Main

Оставаясь в ветке feature/library-class, создайте класс Main с демонстрацией работы системы:

```
public class Main {
    public static void main(String[] args) {
        Library library = new Library();

        // Добавление книг
```

```
        library.addBook(new Book(1, "Война и мир",
            "Л.Н. Толстой", 1869, "978-5-17-090335-2"));
        library.addBook(new Book(2, "Преступление и наказание",
            "Ф.М. Достоевский", 1866, "978-5-17-090336-9"));
        library.addBook(new Book(3, "Анна Каренина",
            "Л.Н. Толстой", 1877, "978-5-17-090337-6"));

        // Демонстрация всех методов...
        // Поиск, выдача, возврат, вывод журнала
    }
}
```

```
git add .
git commit -m "Add Main class with demo scenario"
```

6. Часть 3: Продвинутая работа с Git (45 минут)

В этой части вы освоите три важнейшие команды Git для работы с историей коммитов. Внимательно читайте инструкции и выполняйте команды последовательно.

6.1. Команда git rebase

Теория: Команда rebase позволяет перенести коммиты одной ветки на верхушку другой, создавая линейную историю. В отличие от merge, rebase "переписывает" историю, создавая новые коммиты.

Задание:

Сейчас у вас есть две ветки: feature/book-class и feature/library-class. Ветка library-class не содержит класс Book, потому что была создана от main. Вам нужно перебазировать library-class на book-class, чтобы получить все изменения.

Выполните следующие действия:

```
# 1. Убедитесь, что находитесь в ветке feature/library-class
git branch

# 2. Посмотрите текущую историю коммитов
git log --oneline --graph --all

# 3. Выполните перебазирование на ветку feature/book-class
git rebase feature/book-class

# 4. Посмотрите обновлённую историю
git log --oneline --graph --all
```

Что произошло:

Коммиты из feature/library-class были "перемещены" поверх feature/book-class. Теперь ваш код Library видит класс Book, и история выглядит линейно.

Скопируйте вывод команды git log в отчёт.

6.2. Команда git reset

Теория: Команда reset позволяет отменить коммиты, перемещая указатель HEAD. Существует три режима: --soft (сохраняет изменения в staging area), --mixed (сохраняет в рабочей директории), --hard (удаляет полностью).

Задание:

Представим, что вы решили разбить коммит "Add Main class with demo scenario" на два отдельных коммита: один для структуры Main, другой для демонстрационного сценария.

```
# 1. Посмотрите хэш последнего коммита
git log --oneline -3

# 2. Отмените последний коммит, сохранив изменения
git reset --soft HEAD~1

# 3. Проверьте статус — файлы должны быть в staged
git status

# 4. Уберите файлы из staging area
git reset HEAD

# 5. Теперь добавьте и закоммитьте в два этапа
# Сначала закоммитьте только структуру класса Main
# (метод main с созданием Library и пустым телом)
git add src/main/java/library/Main.java
git commit -m "Add Main class skeleton"

# Затем добавьте демонстрационный код
# (добавление книг, вызовы методов)
git add .
git commit -m "Add demo scenario with sample books"
```

Важно: Для выполнения этого задания вам придётся вручную отредактировать файл Main.java между коммитами. Это учебный сценарий для понимания работы reset.

6.3. Команда git cherry-pick

Теория: Команда cherry-pick позволяет выборочно применить конкретный коммит из одной ветки в другую. Это полезно, когда нужен только один коммит, а не вся ветка.

Задание:

Создайте отдельную ветку с дополнительной функциональностью, а затем перенесите только один коммит из неё в основную ветку разработки.

```
# 1. Создайте новую ветку для экспериментов
git checkout -b feature/experimental

# 2. Добавьте метод getStatistics() в класс Library
# (возвращает строку с общим количеством книг,
# количеством доступных и выданных)
git commit -m "Add getStatistics method"

# 3. Добавьте метод removeBook(int id)
git commit -m "Add removeBook method"

# 4. Добавьте метод updateBook(int id, Book newData)
git commit -m "Add updateBook method"

# 5. Посмотрите историю и запишите хэш коммита getStatistics
git log --oneline -5

# 6. Вернитесь в ветку library-class
git checkout feature/library-class

# 7. Примените только коммит getStatistics
git cherry-pick <хэш-коммита-getStatistics>

# 8. Проверьте результат
git log --oneline -5
```

Результат: Метод getStatistics() теперь есть в feature/library-class, но методы removeBook и updateBook — только в experimental.

6.4. Финальное слияние

```
# 1. Переключитесь на main
git checkout main

# 2. Слейте feature/library-class
git merge feature/library-class
```

```
# 3. Отправьте все изменения на GitHub
git push origin main
git push origin feature/book-class
git push origin feature/library-class
git push origin feature/experimental
```

7. Контрольные вопросы

1. В чём разница между merge и rebase? В каких случаях предпочтительнее использовать каждую из команд?

2. Объясните разницу между git reset --soft, --mixed и --hard. Приведите примеры ситуаций для использования каждого режима.

3. Почему операция rebase считается "опасной" для публичных веток? Что произойдёт, если выполнить rebase ветки, которую уже запушили на GitHub?

4. В чём отличие вложенного статического класса (static nested class) от внутреннего класса (inner class) в Java? Когда следует использовать каждый из них?

5. Опишите сценарий, когда cherry-pick будет более уместен, чем merge целой ветки.