LIBRARY MANAGEMENT SYSTEM DOCUMENTATION

TEAM MEMBERS

Mohamed Yehia Abdelhamed	21-02059
Mahmoud Gamal Ali	21-02062
Alaa Khaled Salah	21-01997
Hossam Ramadan Abdallah	21-01891
Amr Atef Ahmed	21-01994
Esraa Mohamed Abd-Elaziz	21-01852

OVERVIEW

- The **Library Management System** is a Java-based application that allows users to manage books in a library. It uses various **Design Patterns** to provide flexibility, maintainability, and scalability. The system is built with a **Graphical User Interface (GUI)** to make it user-friendly and easy to interact with.

KEY FEATURES

Design Patterns Used:

- Singleton Pattern
- Factory Pattern
- Prototype Pattern
- Builder Pattern
- Adapter Pattern

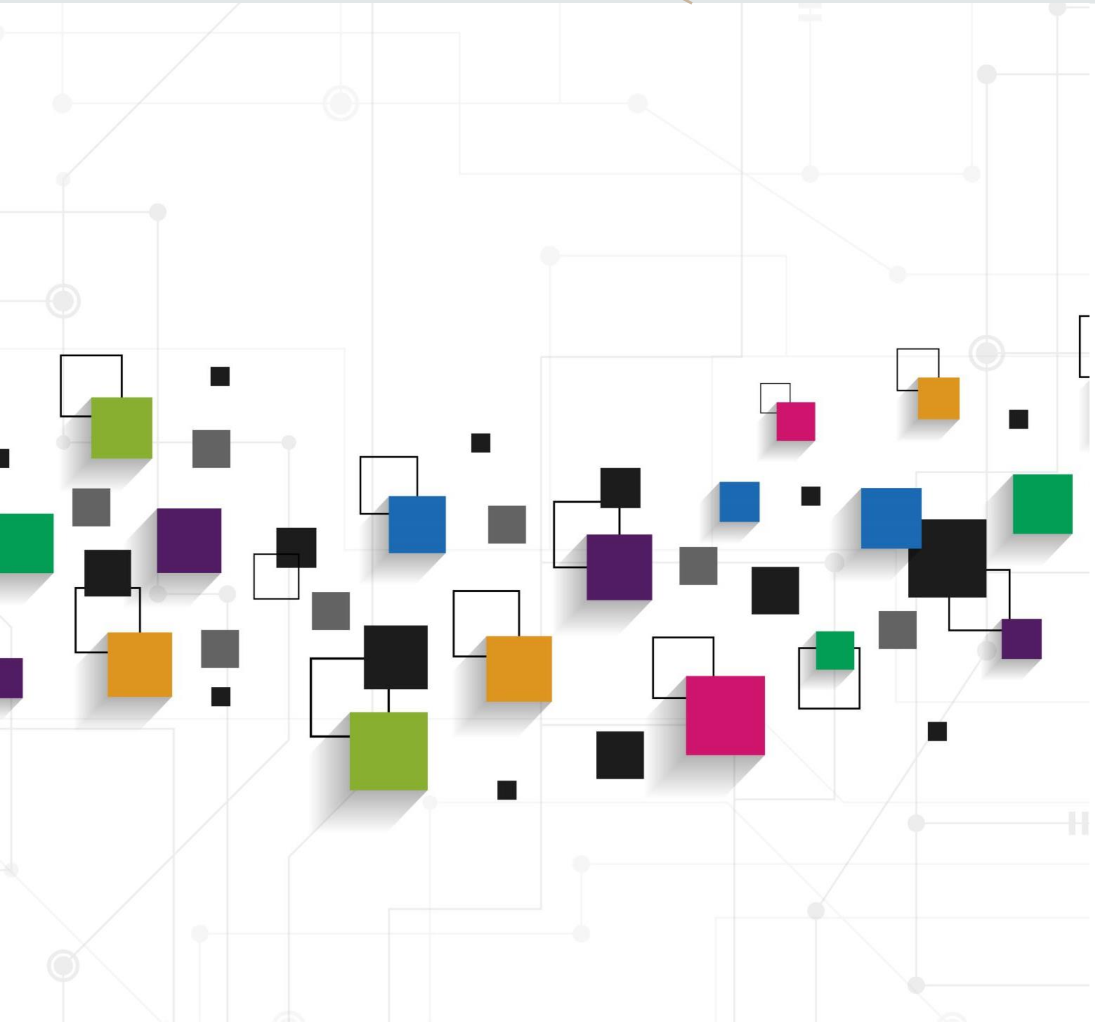
GUI Interface:

- Displays books and their details.
- Allows interaction with book data through buttons, text fields, and labels.

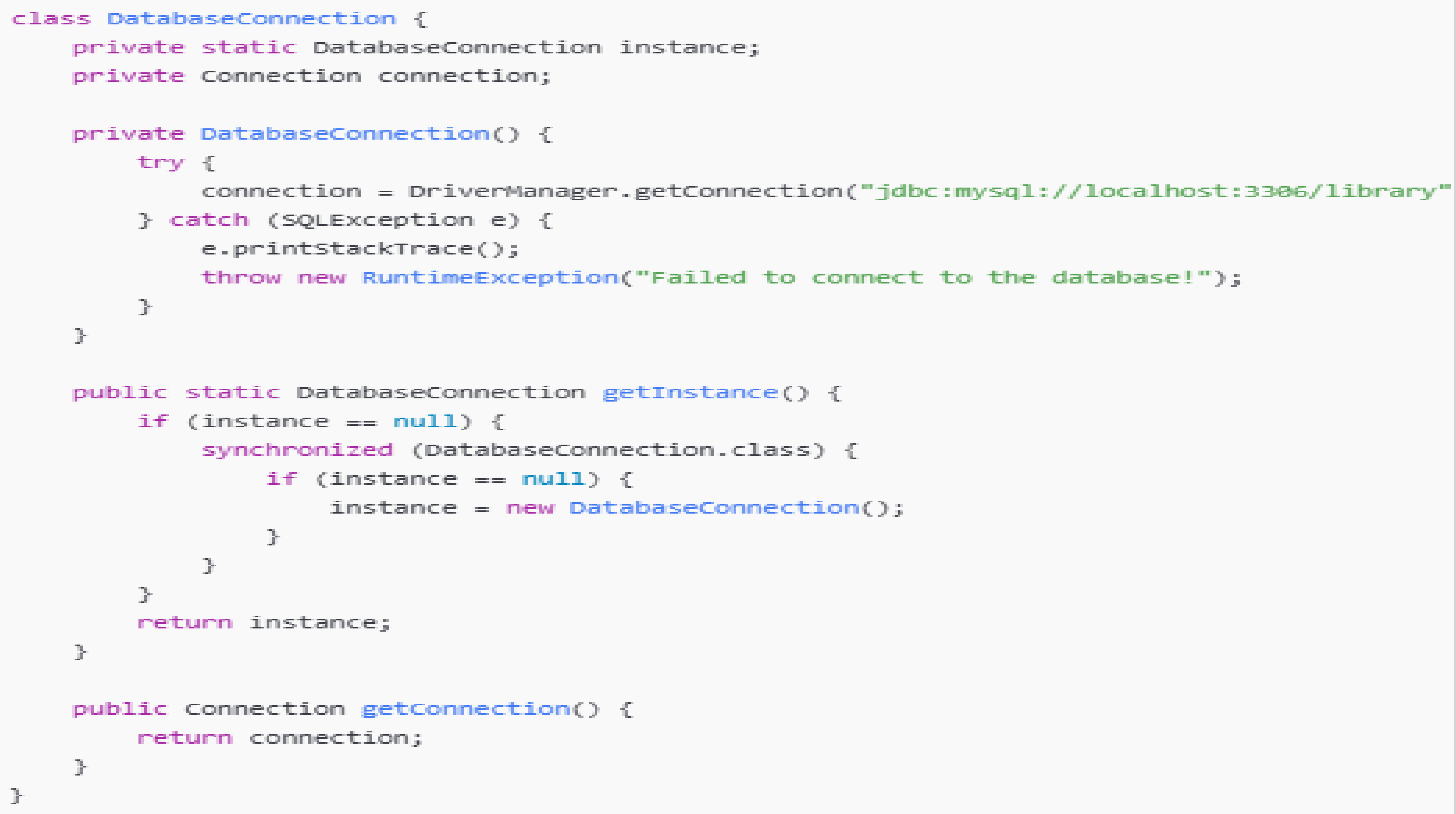
Database Connection:

- (Though not active in the current version) a Singleton pattern is used for managing a database connection.

DESIGN PATTERNS IMPLEMENTED

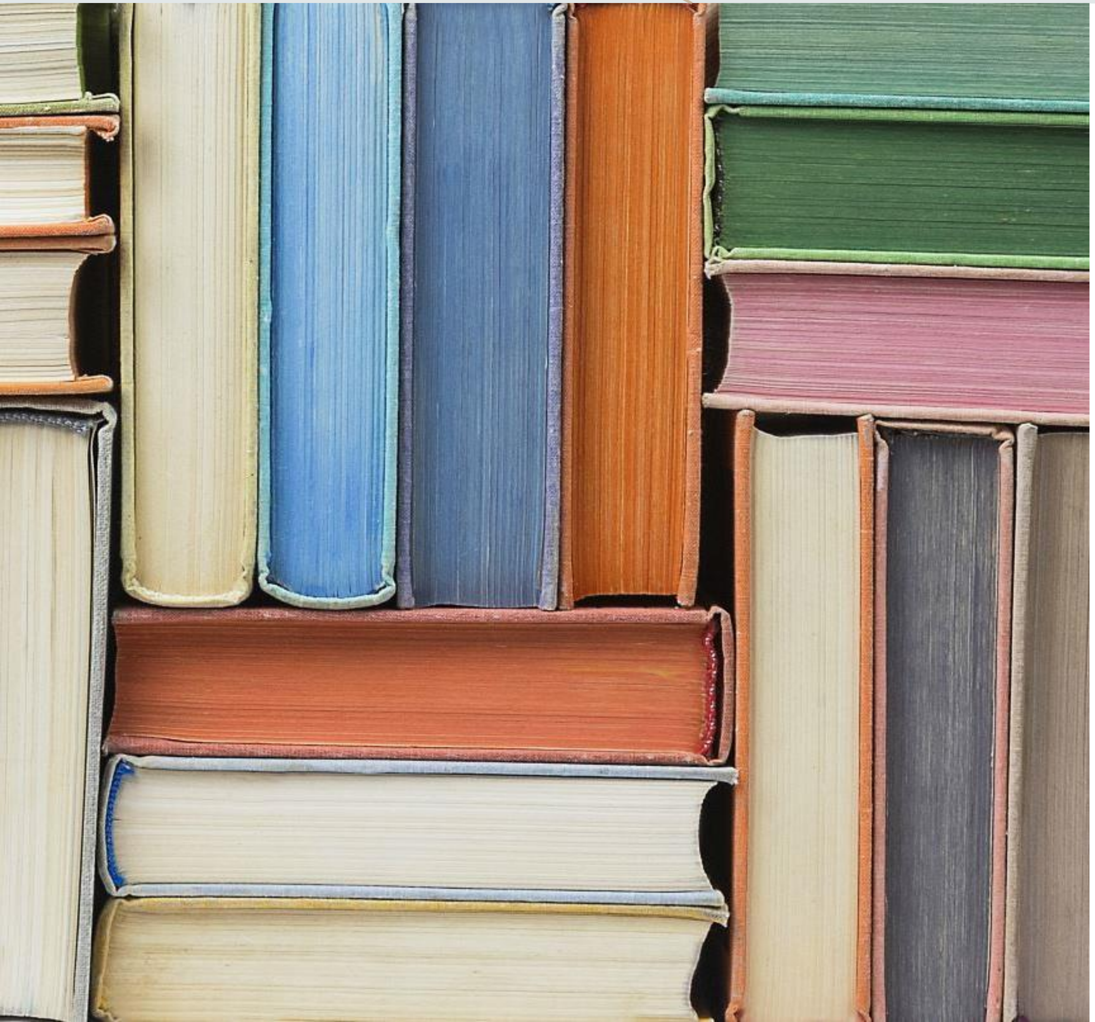


- **Design Patterns Implemented**
- **1. Singleton Pattern**
- The **Singleton Pattern** ensures that there is only **one instance** of the database connection throughout the application. This prevents unnecessary overhead of creating multiple connections and guarantees that the connection is managed properly.
- **Class**: DatabaseConnection
- **Purpose**: Provides a single, shared database connection instance.



```java
class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;

    private DatabaseConnection() {
        try {
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/library"
        } catch (SQLException e) {
            e.printStackTrace();
            throw new RuntimeException("Failed to connect to the database!");
        }
    }

    public static DatabaseConnection getInstance() {
        if (instance == null) {
            synchronized (DatabaseConnection.class) {
                if (instance == null) {
                    instance = new DatabaseConnection();
                }
            }
        }
        return instance;
    }

    public Connection getConnection() {
        return connection;
    }
}
```

2- FACTORY PATTERN



- The **Factory Pattern** is used to create different types of books without specifying the exact class that needs to be instantiated. It abstracts the creation of objects based on user input.
- **Class**: BookFactory
- **Purpose**: Creates and returns different types of books based on the provided type.

```java
class BookFactory {
    public static Book createBook(String type) {
        switch (type.toLowerCase()) {
            case "software engineering":
                return new SoftwareEngineeringBook();
            case "management":
                return new ManagementBook();
            case "ai":
                return new AIBook();
            default:
                throw new IllegalArgumentException("Unknown book type: " + type);
        }
    }
}
```

3- PROTOTYPE PATTERN

- The **Prototype Pattern** allows objects to be cloned without re-creating them from scratch. This pattern is used for making copies of existing book objects.
- **Class**: BookPrototype
- **Purpose**: Provides a mechanism for cloning objects.

```java
abstract class BookPrototype implements Cloneable {
    public abstract BookPrototype clone();
    public abstract void displayDetails();
}

class PrototypeSoftwareEngineeringBook extends BookPrototype {
    @Override
    public BookPrototype clone() {
        return new PrototypeSoftwareEngineeringBook();
    }

    @Override
    public void displayDetails() {
        System.out.println("This is a prototype Software Engineering book.");
    }
}
```

4. BUILDER PATTERN

- The **Builder Pattern** is used to create complex Book objects step by step. This pattern allows the creation of books with various attributes (like title, author, publisher, etc.) without needing a constructor with too many parameters.
- **Class**: BookBuilder
- **Purpose**: Provides a step-by-step approach for creating a Book object.

```java
class BookBuilder {
    private String title;
    private String author;
    private String publisher;
    private String releaseDate;
    private double price;

    public BookBuilder setTitle(String title) {
        this.title = title;
        return this;
    }

    public BookBuilder setAuthor(String author) {
        this.author = author;
        return this;
    }

    public BookBuilder setPublisher(String publisher) {
        this.publisher = publisher;
        return this;
    }

    public BookBuilder setReleaseDate(String releaseDate) {
        this.releaseDate = releaseDate;
        return this;
    }

    public BookBuilder setPrice(double price) {
        this.price = price;
        return this;
    }

    public Book build() {
        return new CustomBook(title, author, publisher, releaseDate, price);
    }
}
```

5. ADAPTER PATTERN

- The **Adapter Pattern** is used to convert the interface of one class into another expected by the client. In this case, it adapts the book's display method so that it can be easily shown in the GUI.
- **Class**: BookDetailsAdapter
- **Purpose**: Adapts the book details so that they can be easily displayed using the GUI.

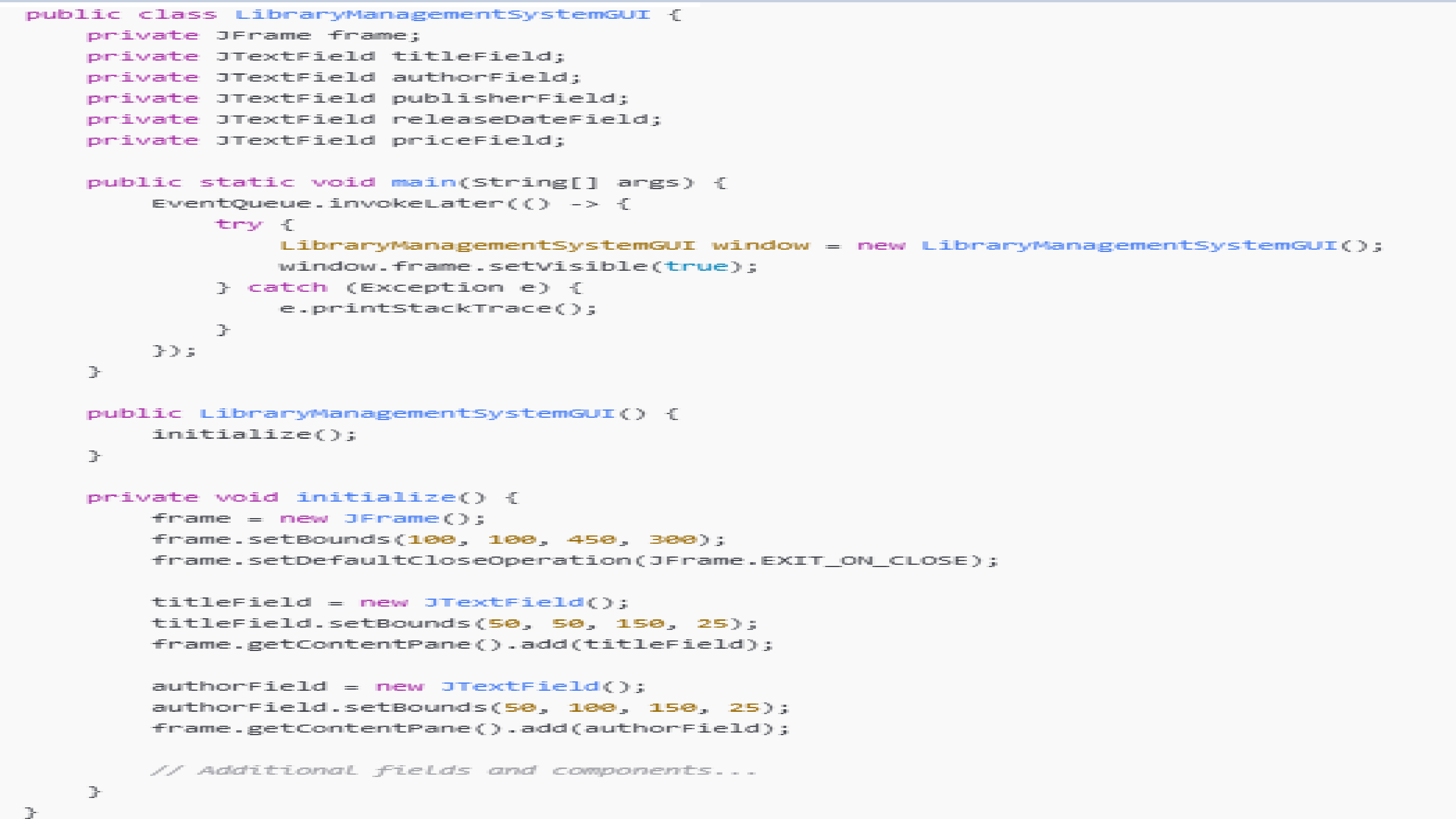
```java
class BookDetailsAdapter {
    private Book book;

    public BookDetailsAdapter(Book book) {
        this.book = book;
    }

    public void showBookDetails() {
        System.out.println("Displaying book details:");
        book.displayDetails();
    }
}
```

GUI DESIGN

- The system uses **JFrame** for the main window.
- **JButtons** are used to trigger actions like creating books or displaying details.
- **JTextFields** and **JLabels** are used to input and display book details.

GUI COMPONENTS:

- **Books Display Area**: A list of books with their details.
- **Buttons** for selecting book types and displaying their details.
- **Text fields** for displaying book attributes such as title, author, price, etc.



```
public class LibraryManagementSystemGUI {
    private JFrame frame;
    private JTextField titleField;
    private JTextField authorField;
    private JTextField publisherField;
    private JTextField releaseDateField;
    private JTextField priceField;

    public static void main(String[] args) {
        EventQueue.invokeLater(() -> {
            try {
                LibraryManagementSystemGUI window = new LibraryManagementSystemGUI();
                window.frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
    }

    public LibraryManagementSystemGUI() {
        initialize();
    }

    private void initialize() {
        frame = new JFrame();
        frame.setBounds(100, 100, 450, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        titleField = new JTextField();
        titleField.setBounds(50, 50, 150, 25);
        frame.getContentPane().add(titleField);

        authorField = new JTextField();
        authorField.setBounds(50, 100, 150, 25);
        frame.getContentPane().add(authorField);

        // Additional fields and components...
    }
}
```

CONCLUSION

- The **Library Management System** utilizes five key design patterns to achieve flexibility, scalability, and ease of maintenance. The GUI makes it easy for users to interact with the system, while the backend system is designed for expansion and future features.
- With the implementation of **Singleton**, **Factory**, **Prototype**, **Builder**, and **Adapter** patterns, the system can easily be extended to include more functionalities such as database integration, user management, and more complex book features.

Thanks