Movie Ticket Booking System

Team Members:

1.	Awsam lotfallah	21-00534
2.	Patrick Masry	21-01438
3.	Mostafa gamal Hassan Goda	21-01300
4.	Marwan ashraf Mohamed	21-01035
5.	Mohamed Tarik Abo Alhamd	21-01167

Under the supervision of Dr.Mahmoud Basuoni

And T.A Arwa Essam

1. Singleton Pattern:

a. Ticket Booking System

```
package movie_factory;

public abstract class Movie {
   public abstract void showDetails();
}
```

Class: `Movie`

This is an abstract class that serves as a blueprint for different types of movies. It defines an abstract method `showDetails()` which must be implemented by any class that extends `Movie`.

Attributes:

- There are no instance variables in this class.

Methods:

`public abstract void showDetails();`

- **Description**: This is an abstract method that must be implemented by any subclass of `Movie`. The method should define how the details of the movie will be shown or displayed.
- **Parameters**: None.
- **Return Type**: `void` (This method does not return any value).
- **Usage**: This method should be implemented in the subclass to provide the specific details of the movie, such as its title, genre, director, and so on.

b.Session Manager

```
public class SessionManager {

    private static SessionManager instance;

    private String loggedInUser;


    private SessionManager() {}


    public static SessionManager getInstance() {

        if (instance == null) {

            instance = new SessionManager();

        }

        return instance;

    }


    public boolean login(String username, String
password) {

        if (("admin".equals(username) &&
"admin123".equals(password)) ||

                ("user".equals(username) &&
"user123".equals(password))) {

            loggedInUser = username;

            return true;

        }
```

```
        return false;
    }

    public boolean isLoggedIn() {
        return loggedInUser != null;
    }

    public String getLoggedInUser() {
        return loggedInUser;
    }

    public boolean isAdmin(String username) {
        return "admin".equals(username);
    }

    public void logout() {
        loggedInUser = null;
    }
}
```

Class: `SessionManager`

The `SessionManager` class is a singleton that manages user sessions, including login, logout, and user authentication. It ensures that only one instance of the class exists throughout the application's lifecycle.

Key Points:

- **Singleton Pattern**:
  - `SessionManager` follows the Singleton design pattern to ensure only one instance of the class is created.
  - The instance is accessed via the `getInstance()` method, which initializes the instance if it doesn't already exist.
- **Attributes**:
  - `instance`: A private static variable to hold the single instance of the `SessionManager`.
  - `loggedInUser`: A private variable to store the username of the currently logged-in user.
- **Methods**:
  - `getInstance()`: Returns the singleton instance of `SessionManager`.
  - `login(String username, String password)`: Authenticates a user by checking the username and password against predefined credentials. If successful, the user is logged in.
  - `isLoggedIn()`: Checks if any user is logged in by verifying if `loggedInUser` is not null.
  - `getLoggedInUser()`: Returns the username of the currently logged-in user.
  - `isAdmin(String username)`: Checks if the given username is "admin".
  - `logout()`: Logs out the current user by setting `loggedInUser` to null.

2.Factory Pattern:

a.Factory for Movie Types

```
package movie_factory;

public class ActionMovie extends Movie {
   public void showDetails() {
       System.out.println("Action Movie details");
   } }
```

Class: `ActionMovie` (extends `Movie`)

This class represents a specific type of movie, an action movie, and provides an implementation for displaying the details of the movie.

Key Features:

- **Extends `Movie`**: Inherits from the abstract `Movie` class and provides an implementation for the `showDetails()` method.
- **Methods**:
    - `showDetails()`: Prints the details of the action movie.

```
package movie_factory;

public class MovieFactory {
    public static Movie createMovie(String genre) {
        if (genre.equalsIgnoreCase("Action")) {
            return new ActionMovie();
        } else if (genre.equalsIgnoreCase("Comedy")) {
            return new ComedyMovie();
        } else if (genre.equalsIgnoreCase("Drama")) {
            return new DramaMovie();
        }
        return null;
    }
}
```

Class: `MovieFactory`

The `MovieFactory` class is a factory that creates and returns different types of movie objects based on the specified genre.

Key Features:

- **Methods**:
    - `createMovie(String genre)`: Takes a genre as input (e.g., "Action", "Comedy", "Drama") and returns the corresponding `Movie` object. If the genre does not match any predefined types, it returns `null`.

```
package movie_factory;


public abstract class Movie {

   public abstract void showDetails();

}
```

Class: `Movie`

An abstract class that serves as a base for different movie genres. It defines an abstract method for displaying movie details, which must be implemented by subclasses.

Key Features:

- **Methods**:
    - `showDetails()`: An abstract method that must be implemented by subclasses to define how the movie's details are displayed.

---

b.Factory for Theater Locations:

```
package theatre_factory;

public class CinemaHall extends Theater {
   public void showDetails() {
        System.out.println("Cinema Hall details");
   }
}
```

Class: `CinemaHall` (extends `Theater`)

This class represents a specific type of theater, a cinema hall, and provides an implementation for displaying its details.

Key Features:

- **Extends `Theater`**: Inherits from the abstract `Theater` class and provides a concrete implementation of the `showDetails()` method.
- **Methods**:
    - `showDetails()`: Prints the details of the cinema hall.

```
package theatre_factory;


public class IMAXTheater extends Theater {

   public void showDetails() {

        System.out.println("IMAX Theater details");

   }

}
```

Class: `IMAXTheater` (extends `Theater`)

This class represents a specific type of theater, an IMAX theater, and provides an implementation for displaying its details.

Key Features:

- **Extends `Theater`**: Inherits from the abstract `Theater` class and provides a concrete implementation of the `showDetails()` method.
- **Methods**:
    - `showDetails()`: Prints the details of the IMAX theater.

```
package theatre_factory;

public abstract class Theater {
   public abstract void showDetails();
}
```

Class: `Theater`

An abstract class that serves as a base for different types of theaters. It defines an abstract method for displaying theater details, which must be implemented by subclasses.

Key Features:

- **Methods**:
    - `showDetails()`: An abstract method that must be implemented by subclasses to define how the theater's details are displayed.

```
package theatre_factory;

public class TheaterFactory {
```

```
    public static Theater createTheater(String type) {
        if (type.equalsIgnoreCase("Cinema")) {
            return new CinemaHall();
        } else if (type.equalsIgnoreCase("IMAX")) {
            return new IMAXTheater();
        }
        return null;
    }
}
```

Class: `TheaterFactory`

The `TheaterFactory` class is a factory that creates and returns different types of theater objects based on the specified type.

Key Features:

- **Methods**:
    - `createTheater(String type)`: Takes a theater type as input (e.g., "Cinema", "IMAX") and returns the corresponding `Theater` object. If the type does not match any predefined types, it returns `null`.

3.Builder Pattern

```java
package builder_pattern;

import prototype_pattern.MovieTicket;

/**
* MovieTicketBuilder implements the Builder pattern to
construct MovieTicket objects.
* Provides a fluent interface for setting ticket
properties and creating tickets.
* This builder ensures a clean and flexible way to create
movie tickets with all required attributes.
*/
public class MovieTicketBuilder {
   // Properties for building a MovieTicket
   private String genre;          // Movie genre (e.g.,
Action, Comedy, Drama)
   private String theaterType;    // Type of theater
(e.g., Cinema, IMAX)
   private int ticketNumber;      // Unique ticket
identifier

   /**
    * Sets the movie genre for the ticket being built
    * Uses method chaining for fluent interface
    *
    * @param genre The movie genre to set
    * @return The builder instance for method chaining
    */
   public MovieTicketBuilder setGenre(String genre) {
       this.genre = genre;
       return this;
   }

   /**
```

```
     * Sets the theater type for the ticket being built
     * Uses method chaining for fluent interface
     *
     * @param theaterType The theater type to set
     * @return The builder instance for method chaining
     */
    public MovieTicketBuilder setTheaterType(String
theaterType) {
        this.theaterType = theaterType;
        return this;
    }

    /**
     * Sets the ticket number for the ticket being built
     * Uses method chaining for fluent interface
     *
     * @param ticketNumber The ticket number to set
     * @return The builder instance for method chaining
     */
    public MovieTicketBuilder setTicketNumber(int
ticketNumber) {
        this.ticketNumber = ticketNumber;
        return this;
    }

    /**
     * Builds and returns a new MovieTicket instance with
the configured properties
     * Creates a new ticket using the Prototype pattern
via MovieTicket constructor
     *
     * @return A new MovieTicket instance with the
specified properties
     */
    public MovieTicket build() {
```

```
        return new MovieTicket(genre, theaterType,
ticketNumber);
    }
}
```

Class: `MovieTicketBuilder`

The `MovieTicketBuilder` class implements the Builder design pattern to construct `MovieTicket` objects. It provides a fluent interface to set the properties of a movie ticket and ensures a clean and flexible way to create tickets with all required attributes.

Key Features:

- **Attributes**:
  - `genre`: Movie genre (e.g., Action, Comedy, Drama).
  - `theaterType`: Type of theater (e.g., Cinema, IMAX).
  - `ticketNumber`: Unique ticket identifier.
- **Methods**:
  - `setGenre(String genre)`: Sets the movie genre for the ticket. Returns the builder instance for method chaining.
  - `setTheaterType(String theaterType)`: Sets the theater type for the ticket. Returns the builder instance for method chaining.
  - `setTicketNumber(int ticketNumber)`: Sets the unique ticket number for the ticket. Returns the builder instance for method chaining.
  - `build()`: Constructs and returns a new `MovieTicket` instance using the configured properties.

Design Pattern:

- **Builder Pattern**: The class is designed to construct complex `MovieTicket` objects step by step.
- **Fluent Interface**: Uses method chaining to make the construction process more readable and flexible.

4.Prototype Pattern

```
package prototype_pattern;

// MovieTicket class implements the TicketPrototype
interface, supporting cloning of ticket objects
public class MovieTicket implements TicketPrototype {
   private String genre;         // Genre of the movie
(e.g., Action, Drama)
   private String theaterType;  // Type of theater (e.g.,
IMAX, Standard)
   private int ticketNumber;    // Unique ticket number

   /**
    * Constructor to initialize a MovieTicket with its
details.
    *
    * @param genre         The genre of the movie.
    * @param theaterType  The type of theater.
    * @param ticketNumber The unique ticket number.
    */
   public MovieTicket(String genre, String theaterType,
int ticketNumber) {
       this.genre = genre;
       this.theaterType = theaterType;
       this.ticketNumber = ticketNumber;
   }

   /**
    * Creates a clone of the current MovieTicket
instance.
    *
    * @return A new MovieTicket object with the same
attributes as the original.
    */
   @Override
```

```
    public TicketPrototype cloneTicket() {
        return new MovieTicket(this.genre,
this.theaterType, this.ticketNumber);
    }

    /**
     * Returns a string representation of the MovieTicket
object.
     *
     * @return A string describing the ticket details.
     */
    @Override
    public String toString() {
        return "Ticket [Genre: " + genre + ", Theater: " +
theaterType + ", Ticket Number: " + ticketNumber + "]";
    }
}
```

Class: `MovieTicket` (implements `TicketPrototype`)

The `MovieTicket` class implements the `TicketPrototype` interface, allowing it to create clones of ticket objects. It stores details about a movie ticket, including its genre, theater type, and unique ticket number.

Key Features:

- **Attributes**:
  - `genre`: The genre of the movie (e.g., Action, Drama).
  - `theaterType`: The type of theater (e.g., IMAX, Standard).
  - `ticketNumber`: A unique identifier for the ticket.
- **Methods**:
  - `MovieTicket(String genre, String theaterType, int ticketNumber)`: Constructor to initialize a movie ticket with its details.
  - `cloneTicket()`: Creates and returns a new instance of `MovieTicket` with the same attributes as the original, supporting the Prototype pattern.

- `toString()`: Returns a string representation of the ticket, including its genre, theater type, and ticket number.

Design Pattern:

- **Prototype Pattern**: This class supports cloning by creating copies of movie tickets with identical attributes.

```
package prototype_pattern;


// MovieTicket class implements the TicketPrototype
interface, supporting cloning of ticket objects
public class MovieTicket implements TicketPrototype {
   private String genre;         // Genre of the movie
(e.g., Action, Drama)
   private String theaterType;  // Type of theater (e.g.,
IMAX, Standard)
   private int ticketNumber;    // Unique ticket number


   /**
    * Constructor to initialize a MovieTicket with its
details.
    *
    * @param genre         The genre of the movie.
    * @param theaterType  The type of theater.
    * @param ticketNumber The unique ticket number.
```

```
     */
    public MovieTicket(String genre, String theaterType,
int ticketNumber) {
        this.genre = genre;
        this.theaterType = theaterType;
        this.ticketNumber = ticketNumber;
    }

    /**
     * Creates a clone of the current MovieTicket
instance.
     *
     * @return A new MovieTicket object with the same
attributes as the original.
     */
    @Override
    public TicketPrototype cloneTicket() {
        return new MovieTicket(this.genre,
this.theaterType, this.ticketNumber);
    }

    /**
     * Returns a string representation of the MovieTicket
object.
```

```
     *
     * @return A string describing the ticket details.
     */
    @Override
    public String toString() {
        return "Ticket [Genre: " + genre + ", Theater: " +
theaterType + ", Ticket Number: " + ticketNumber + "]";
    }
}
```

Class: `MovieTicket` (implements `TicketPrototype`)

The `MovieTicket` class implements the `TicketPrototype` interface, enabling cloning of ticket objects. It represents the details of a movie ticket, including its genre, theater type, and unique ticket number.

Key Features:

- **Attributes**:
  - `genre`: The genre of the movie (e.g., Action, Drama).
  - `theaterType`: The type of theater (e.g., IMAX, Standard).
  - `ticketNumber`: The unique ticket number.
- **Methods**:
  - `MovieTicket(String genre, String theaterType, int ticketNumber)`: Constructor to initialize the movie ticket with genre, theater type, and ticket number.
  - `cloneTicket()`: Creates and returns a new `MovieTicket` instance with the same attributes as the original, supporting the Prototype pattern.
  - `toString()`: Returns a string representation of the `MovieTicket`, displaying its genre, theater type, and ticket number.

---

5. Adapter Pattern

```
/**
* TicketBookingAdapter implements the Adapter pattern to
provide a unified interface
* for ticket booking operations. It adapts the
TicketBookingSystem to the MovieBooking interface.
* <p>
* This adapter ensures compatibility between the
MovieBooking interface requirements
* and the existing TicketBookingSystem implementation.
*/
public class TicketBookingAdapter implements MovieBooking
{
   // Reference to the singleton instance of
TicketBookingSystem
   private TicketBookingSystem ticketBookingSystem;

   /**
    * Constructor initializes the adapter with a
reference to the TicketBookingSystem
    * Uses singleton pattern to get the system instance
    */
   public TicketBookingAdapter() {
       this.ticketBookingSystem =
TicketBookingSystem.getInstance();
   }

   /**
    * Adapts the ticket booking request to the
TicketBookingSystem implementation
    * Implements the MovieBooking interface method
    *
    * @param genre        The movie genre for the ticket
(e.g., "Action", "Comedy", "Drama")
    * @param theaterType  The type of theater (e.g.,
"Cinema", "IMAX")
```

```
     * @param ticketNumber The number of tickets to book
     * @return String containing the booking confirmation
or error message
     */
    @Override
    public String bookTicket(String genre, String
theaterType, int ticketNumber) {
        // Delegates the actual booking operation to the
TicketBookingSystem
        return ticketBookingSystem.bookTicket(genre,
theaterType, ticketNumber);
    }
}
```

Class: `TicketBookingAdapter` (implements `MovieBooking`)

The `TicketBookingAdapter` class implements the Adapter pattern to provide a unified interface for ticket booking operations. It adapts the existing `TicketBookingSystem` to the `MovieBooking` interface, ensuring compatibility between the system and the interface requirements.

Key Features:

- **Attributes**:
  - `ticketBookingSystem`: A reference to the singleton instance of the `TicketBookingSystem`.
- **Methods**:
  - `TicketBookingAdapter()`: Constructor that initializes the adapter with a reference to the `TicketBookingSystem`, using the singleton pattern.
  - `bookTicket(String genre, String theaterType, int ticketNumber)`: Implements the `MovieBooking` interface method. It adapts the movie booking request and delegates the actual booking operation to the `TicketBookingSystem`.

Design Pattern:

- **Adapter Pattern**: This class adapts the `TicketBookingSystem` to the `MovieBooking` interface, enabling compatibility between two incompatible systems.

```java
/**
* TicketBookingAdapter implements the Adapter pattern to
provide a unified interface
* for ticket booking operations. It adapts the
TicketBookingSystem to the MovieBooking interface.
* <p>
* This adapter ensures compatibility between the
MovieBooking interface requirements
* and the existing TicketBookingSystem implementation.
*/
public class TicketBookingAdapter implements MovieBooking
{
   // Reference to the singleton instance of
TicketBookingSystem
   private TicketBookingSystem ticketBookingSystem;


   /**
    * Constructor initializes the adapter with a
reference to the TicketBookingSystem
    * Uses singleton pattern to get the system instance
    */
   public TicketBookingAdapter() {
```

```
        this.ticketBookingSystem =
TicketBookingSystem.getInstance();
    }


    /**
     * Adapts the ticket booking request to the
TicketBookingSystem implementation
     * Implements the MovieBooking interface method
     *
     * @param genre         The movie genre for the ticket
(e.g., "Action", "Comedy", "Drama")
     * @param theaterType  The type of theater (e.g.,
"Cinema", "IMAX")
     * @param ticketNumber The number of tickets to book
     * @return String containing the booking confirmation
or error message
     */
    @Override
    public String bookTicket(String genre, String
theaterType, int ticketNumber) {
        // Delegates the actual booking operation to the
TicketBookingSystem
        return ticketBookingSystem.bookTicket(genre,
theaterType, ticketNumber);
    }}
```

Class: `TicketBookingAdapter` (implements `MovieBooking`)

The `TicketBookingAdapter` class implements the Adapter design pattern, providing a unified interface for ticket booking operations. It adapts the existing `TicketBookingSystem` to the `MovieBooking` interface, ensuring compatibility between the system and the interface.

Key Features:

- **Attributes**:
  - `ticketBookingSystem`: A reference to the singleton instance of the `TicketBookingSystem`.
- **Methods**:
  - `TicketBookingAdapter()`: Constructor that initializes the adapter with a reference to the `TicketBookingSystem`, utilizing the singleton pattern.
  - `bookTicket(String genre, String theaterType, int ticketNumber)`: Implements the `MovieBooking` interface method. It adapts the booking request and delegates the operation to the `TicketBookingSystem`.

```
import builder_pattern.MovieTicketBuilder;

import movie_factory.Movie;

import movie_factory.MovieFactory;

import prototype_pattern.MovieTicket;

import theatre_factory.Theater;

import theatre_factory.TheaterFactory;



import javax.swing.*;

import java.awt.event.ActionListener;
```

```java
/**
* Main GUI class for the Movie Ticket Booking System
* Implements a user interface with login, user, and admin functionalities
*/
public class MovieTicketBookingGUI {
    // GUI Components
    private JFrame frame;
    private JTextField ticketNumberField, usernameField;
    private JPasswordField passwordField;
    private JComboBox<String> genreComboBox, theaterComboBox;
    private JLabel bookedTicketsLabel;

    // System Components using Singleton pattern
    private TicketBookingSystem ticketBookingSystem = TicketBookingSystem.getInstance();
    private SessionManager sessionManager = SessionManager.getInstance();

    /**
```

```
    * Constructor: Initializes the main application
window
    */
   public MovieTicketBookingGUI() {
       frame = new JFrame("Movie Ticket Booking");

frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       frame.setSize(500, 400);
       showLogPanel();  // Display log panel first
       frame.setLayout(null);
       frame.setVisible(true);
   }

   /**
    * Creates and displays the login panel with username
and password fields
    */
   private void showLogPanel() {
       JPanel logPanel = createPanel();

       // Username label and text field
```

```
        logPanel.add(new
JLabel("Username:")).setBounds(50, 50, 100, 20);

        usernameField = new JTextField();

        usernameField.setBounds(150, 50, 150, 20);

        logPanel.add(usernameField);



        // Password label and password field

        logPanel.add(new
JLabel("Password:")).setBounds(50, 80, 100, 20);

        passwordField = new JPasswordField();

        passwordField.setBounds(150, 80, 150, 20);

        logPanel.add(passwordField);



        // Login button

        addButton(logPanel, "Login", 150, 110, e ->
loginUser());



        frame.setContentPane(logPanel);

        frame.revalidate();

        frame.repaint();

    }
```

```
    /**
     * Validates user credentials and proceeds to login
panel if successful
     */
    private void loginUser() {
        String username = usernameField.getText();
        String password = new
String(passwordField.getPassword());

        if (sessionManager.login(username, password)) {
            JOptionPane.showMessageDialog(frame, "Login
successful!");
            showLoginPanel();
        } else {
            JOptionPane.showMessageDialog(frame, "Invalid
username or password", "Login Failed",
JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Displays the main menu panel after successful login
     * Shows options for User and Admin access
```

```
     */
    private void showLoginPanel() {
        JPanel panel = createPanel();
        addButton(panel, "User", 150, 50, e ->
showUserPanel());
        addButton(panel, "Admin", 150, 100, e ->
showAdminPanel());
        addButton(panel, "Logout", 150, 150, e ->
logoutUser());

        frame.setContentPane(panel);
        frame.revalidate();
        frame.repaint();
    }

    /**
     * Handles user logout and returns to login screen
     */
    private void logoutUser() {
        sessionManager.logout(); // Clear the session
        JOptionPane.showMessageDialog(frame, "Logged out
successfully!");
```

```
        showLogPanel(); // Show the log panel again
    }

    /**
     * Creates a new panel with null layout
     *
     * @return JPanel with null layout
     */
    private JPanel createPanel() {
        JPanel panel = new JPanel();
        panel.setLayout(null);
        return panel;
    }

    /**
     * Helper method to add buttons with specific
properties
     *
     * @param panel    Panel to add button to
     * @param text     Button text
     * @param x        X coordinate
```

```
     * @param y         Y coordinate
     * @param listener Action listener for button
     */
    private void addButton(JPanel panel, String text, int
x, int y, ActionListener listener) {
        JButton button = new JButton(text);
        button.setBounds(x, y, 150, 30);
        button.addActionListener(listener);
        panel.add(button);
    }

    /**
     * Displays the user panel with ticket booking options
     */
    private void showUserPanel() {
        JPanel userPanel = createPanel();

        genreComboBox = addComboBox(userPanel, "Genre:",
new String[]{"Action", "Comedy", "Drama"}, 50, 50);
        theaterComboBox = addComboBox(userPanel, "Theater
Type:", new String[]{"Cinema", "IMAX"}, 50, 80);
```

```
        ticketNumberField = addTextField(userPanel,
"Ticket Number:", 50, 110);

        bookedTicketsLabel = new JLabel("Booked Tickets:
0");

        bookedTicketsLabel.setBounds(50, 140, 200, 20);

        userPanel.add(bookedTicketsLabel);



        addButton(userPanel, "Book Ticket", 150, 170, e ->
bookTicketForUser());

        addButton(userPanel, "Back", 10, 10, e ->
showLoginPanel());



        frame.setContentPane(userPanel);

        frame.revalidate();

        frame.repaint();

    }



    /**

     * Helper method to add text fields with labels

     *

     * @param panel     Panel to add field to

     * @param labelText Label text
```

```
     * @param x          X coordinate
     * @param y          Y coordinate
     * @return Created JTextField
     */
    private JTextField addTextField(JPanel panel, String
labelText, int x, int y) {
        panel.add(new JLabel(labelText)).setBounds(x, y,
100, 20);
        JTextField textField = new JTextField();
        textField.setBounds(x + 100, y, 150, 20);
        panel.add(textField);
        return textField;
    }

    /**
     * Helper method to add combo boxes with labels
     *
     * @param panel     Panel to add combo box to
     * @param labelText Label text
     * @param items     Combo box items
     * @param x         X coordinate
```

```
     * @param y          Y coordinate
     * @return Created JComboBox
     */
    private JComboBox<String> addComboBox(JPanel panel,
String labelText, String[] items, int x, int y) {
        panel.add(new JLabel(labelText)).setBounds(x, y,
100, 20);
        JComboBox<String> comboBox = new
JComboBox<>(items);
        comboBox.setBounds(x + 100, y, 150, 20);
        panel.add(comboBox);
        return comboBox;
    }

    /**
     * Displays the admin panel with inventory management
option
     */
    private void showAdminPanel() {
        JPanel adminPanel = createPanel();
        addButton(adminPanel, "Manage Inventory", 150,
100, e -> manageTicketInventory());
```

```
        addButton(adminPanel, "Back", 10, 10, e ->
showLoginPanel());

        frame.setContentPane(adminPanel);

        frame.revalidate();

        frame.repaint();

    }



    /**

     * Displays the inventory management panel for admins

     */

    private void manageTicketInventory() {

        JPanel inventoryPanel = createPanel();



        // Add IMAX title

        JLabel cinemaTitleLabel = new JLabel("IMAX");

        cinemaTitleLabel.setBounds(260, 17, 70, 20);

        cinemaTitleLabel.setFont(new
java.awt.Font("Arial", java.awt.Font.BOLD, 13));

        inventoryPanel.add(cinemaTitleLabel);



        // Add Cinema title
```

```
        JLabel imaxTitleLabel = new JLabel("Cinema");

        imaxTitleLabel.setBounds(147, 15, 70, 20);

        imaxTitleLabel.setFont(new java.awt.Font("Arial",
java.awt.Font.BOLD, 14));

        inventoryPanel.add(imaxTitleLabel);



        // Add ticket information for each genre and
theater type

        String[] genres = {"Action", "Comedy", "Drama"};

        int yPos = 50;

        for (String genre : genres) {

            addTicketInfo(inventoryPanel, genre +
"-Cinema", 50, yPos);

            addTicketInfo(inventoryPanel, genre + "-IMAX",
150, yPos);

            yPos += 80;

        }



        JButton backButton = new JButton("Back");

        backButton.setBounds(10, 10, 70, 20);

        backButton.addActionListener(e ->
showAdminPanel());

        inventoryPanel.add(backButton);
```

```
        frame.setContentPane(inventoryPanel);
        frame.revalidate();
        frame.repaint();
    }

    /**
     * Adds ticket information display for a specific genre and theater type
     *
     * @param panel     Panel to add information to
     * @param ticketKey Combined key of genre and theater type
     * @param x         X coordinate
     * @param y         Y coordinate
     */
    private void addTicketInfo(JPanel panel, String ticketKey, int x, int y) {
        // Extract genre from ticketKey (before "-")
        String genre = ticketKey.split("-")[0];
        // Extract theater type from ticketKey (after "-")
        String theaterType = ticketKey.split("-")[1];
```

```
        // Create label for movie genre
        JLabel label = new JLabel(genre);
        label.setBounds(x, y, 100, 20);
        panel.add(label);
        // Get available tickets for this type
        int availableTickets =
ticketBookingSystem.getAvailableTickets(genre,
theaterType);
        // Create text field to display available tickets
        JTextField ticketField = new
JTextField(String.valueOf(availableTickets));
        ticketField.setBounds(x + 100, y, 50, 20);
        ticketField.setEditable(false);
        panel.add(ticketField);
    }


    /**
     * Handles the ticket booking process for users
     * Implements the Builder pattern for ticket creation
     */
    private void bookTicketForUser() {
```

```
        String genre = (String)
genreComboBox.getSelectedItem();

        String theaterType = (String)
theaterComboBox.getSelectedItem();

        int ticketNumber;



        try {

            ticketNumber =
Integer.parseInt(ticketNumberField.getText());

            if (ticketNumber < 1 || ticketNumber > 50) {

                showError("Ticket number must be between 1
and 50.");

                return;

            }

        } catch (NumberFormatException e) {

            showError("Please enter a valid ticket
number.");

            return;

        }



        // Create movie and theater using respective
factories

        Movie movie = MovieFactory.createMovie(genre);
```

```
        Theater theater =
ThеaterFactory.createTheater(theaterType);



        if (movie != null) movie.showDetails();

        if (theater != null) theater.showDetails();



        // Use the Builder pattern to create MovieTicket

        MovieTicket movieTicket = new MovieTicketBuilder()

                .setGenre(genre)

                .setTheaterType(theaterType)

                .setTicketNumber(ticketNumber)

                .build();



        String message =
ticketBookingSystem.bookTicket(genre, theaterType,
ticketNumber);

        JOptionPane.showMessageDialog(frame, message);



        // Update display of booked tickets

        updateBookedTicketsLabel(genre, theaterType);

    }
```

```
    /**
     * Displays error message dialog
     *
     * @param message Error message to display
     */
    private void showError(String message) {
        JOptionPane.showMessageDialog(frame, message,
"Error", JOptionPane.ERROR_MESSAGE);
    }


    /**
     * Updates the label showing number of booked tickets
     *
     * @param genre       Movie genre
     * @param theaterType Theater type
     */
    private void updateBookedTicketsLabel(String genre,
String theaterType) {
        int bookedCount =
ticketBookingSystem.getBookedTickets(genre, theaterType);
        bookedTicketsLabel.setText("Booked Tickets: " +
bookedCount);
```

```
    }


    /**
     * Main method to launch the application
     */
    public static void main(String[] args) {
        new MovieTicketBookingGUI();
    }
}
```

Class Overview:

- **Main Functionality:**
  The class provides a GUI for users and administrators to interact with a movie ticket booking system. It offers different views such as login, user options, and admin management.

Key Components:

1. **Login Panel** (`showLogPanel`):
   - Allows users to enter their credentials (username/password) and log into the system.
   - Validates the credentials using `sessionManager`.
2. **Main Menu Panel** (`showLoginPanel`):
   - Displays options for regular users and administrators after successful login.
3. **User Panel** (`showUserPanel`):
   - Enables users to select a movie genre and theater type to book tickets.
   - It also shows the number of booked tickets and allows ticket bookings.
4. **Admin Panel** (`showAdminPanel`):
   - Provides an option to manage the ticket inventory.
   - Displays available tickets for each genre and theater type.

5. **Inventory Management Panel** (`manageTicketInventory`):
    - Allows admins to see available tickets for each genre and theater type and manage the inventory.
6. **Ticket Booking Process** (`bookTicketForUser`):
    - Uses the **Builder pattern** to create `MovieTicket` objects.
    - Utilizes the **Factory pattern** to create movie and theater objects based on user selections.
    - Books tickets using the **TicketBookingSystem** and updates the GUI to reflect the booked tickets.
7. **Helper Methods:**
    - `addButton`: Adds buttons to panels with defined properties.
    - `addComboBox`: Adds combo boxes for genre and theater type selection.
    - `addTextField`: Adds text fields for user input (like ticket number).
8. **Error Handling**:
    - Validates input (like ticket number) and displays error messages using `showError`.

Design Patterns:

- **Singleton Pattern**: Used in `TicketBookingSystem` and `SessionManager` to ensure only one instance of these components.
- **Builder Pattern**: Used in the `MovieTicketBuilder` class to provide a fluent interface for building `MovieTicket` objects.
- **Factory Pattern**: Utilized in `MovieFactory` and `TheaterFactory` to create objects of movies and theaters based on user input.
- **Prototype Pattern**: In `MovieTicket`, the ticket object can be cloned.
- **Adapter Pattern**: The `TicketBookingAdapter` adapts the `MovieBooking` interface to work with the `TicketBookingSystem`.

Conclusion:

The Movie Ticket Booking System implemented in this code leverages multiple design patterns, including Singleton, Builder, Factory, Prototype, and Adapter, to provide a structured, efficient, and maintainable solution for managing movie ticket bookings. The system features a user-friendly GUI for both users and administrators, with functionalities such as user authentication, ticket booking, and inventory management. By using these design patterns, the system ensures scalability, flexibility, and separation of concerns, making it easier to extend and modify in the future. The implementation successfully integrates these patterns to address the complexities of a real-world movie ticket booking system while maintaining a clean and modular codebase.