Studio
Shodwe

Movie Ticket Booking System

Design Pattern

01 Singlton	02 Factory	03 Builder
04 Prototype	05 Observer	06 Decorator
		07 GUI

1-Singlton Pattern

Ensures only one instance of the class exists.

Includes a getInstance() method to .create or retrieve the single instance

The bookTicket() method is used to book tickets and displays a confirmation message using .JOptionPane

```
// --- Singleton for Ticket Booking System ---

class TicketBookingSystem {
    private static TicketBookingSystem instance;

    private TicketBookingSystem() {}

    public static TicketBookingSystem getInstance() {
        if (instance == null) {
            instance = new TicketBookingSystem();
        }
        return instance;
    }

    public void bookTicket(String movieName, String theaterType, String genre, int seats) {
        JOptionPane.showMessageDialog(null, "Booking confirmed for " + movieName + " at " + theaterType + ".\nGenre: "
    }
}
```

2-Factory Pattern

The code implements the Factory Pattern to create different types of movies (Action, Comedy, Drama) based on the provided genre.

It includes a base class Movie with a genre attribute.

There are subclasses for each movie type (ActionMovie, ComedyMovie, DramaMovie).

```
abstract class Movie {
    String genre;
}

class ActionMovie extends Movie {
    public ActionMovie() {
        this.genre = "Action";
    }
}

class ComedyMovie extends Movie {
    public ComedyMovie() {
        this.genre = "Comedy";
    }
}

class DramaMovie extends Movie {
    public DramaMovie() {
        this.genre = "Drama";
    }
}

class MovieFactory {
    public static Movie createMovie(String genre) {
        switch (genre) {
            case "Action":
                return new ActionMovie();
            case "Comedy":
                return new ComedyMovie();
            case "Drama":
                return new DramaMovie();
            default:
                return null;
        }
    }
}
```

```
// --- Factory Pattern for Theater Types ---

abstract class TheaterType {
    String name;

    public String getName() {
        return name;
    }
}

class CinemaHall extends TheaterType {
    public CinemaHall() {
        this.name = "Cinema Hall";
    }
}

class IMAX extends TheaterType {
    public IMAX() {
        this.name = "IMAX";
    }
}

class TheaterFactory {
    public static TheaterType createTheater(String type) {
        switch (type) {
            case "Cinema Hall":
                return new CinemaHall();
            case "IMAX":
                return new IMAX();
            default:
                return null;
        }
    }
}
```

3-Builder Pattern

The code uses a constructor to initialize the Ticket object and getter methods to retrieve individual details

The Builder Pattern makes the code modular, clean, and scalable for adding new fields or customization

```java
// --- Builder Pattern ---

class Ticket {
    private String movieName;
    private String theaterType;
    private String genre;
    private int seats;
    private double price;

    public Ticket(String movieName, String theaterType, String genre, int seats, double price) {
        this.movieName = movieName;
        this.theaterType = theaterType;
        this.genre = genre;
        this.seats = seats;
        this.price = price;
    }

    public String getMovieName() {
        return movieName;
    }

    public String getTheaterType() {
        return theaterType;
    }

    public String getGenre() {
        return genre;
    }

    public int getSeats() {
        return seats;
    }

    public double getPrice() {
        return price;
    }

    public String getDetails() {
        return "Movie: " + movieName + "\nTheater: " + theaterType + "\nGenre: " + genre + "\nSeats: " + seats + "\nPrice: $" + price;
    }
}
```

4-Prototype Pattern

The code implements the Prototype Design Pattern using the Cloneable interface and the clone() method, allowing easy duplication of the TicketPrototype object.

It includes properties like movie name, theater type, genre, seats, and price, with a method to display ticket details.

```java
// The Ticket class implements the Prototype interface
class TicketPrototype implements Cloneable {
    private String movieName;
    private String theaterType;
    private String genre;
    private int seats;
    private double price;

    public TicketPrototype(String movieName, String theaterType, String genre, int seats, double price) {
        this.movieName = movieName;
        this.theaterType = theaterType;
        this.genre = genre;
        this.seats = seats;
        this.price = price;
    }

    public String getMovieName() {
        return movieName;
    }

    public String getTheaterType() {
        return theaterType;
    }

    public String getGenre() {
        return genre;
    }

    public int getSeats() {
        return seats;
    }

    public double getPrice() {
        return price;
    }

    public String getDetails() {
        return "Movie: " + movieName + "\nTheater: " + theaterType + "\nGenre: " + genre + "\nSeats: " + seats + "\nPrice: $" + price;
    }

    // Prototype - Cloning method
    public TicketPrototype clone() {
        try {
            return (TicketPrototype) super.clone();  // Return a clone of the current Ticket object
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

5-Observer Pattern

This code defines the Observer pattern, where the TicketBookingSystemObservable acts as the subject that maintains a list of observers and notifies them whenever a ticket is booked The TicketObserver interface provides a method update for observers to handle the ticket details when notified

```
// Subject Class (Observable)
class TicketBookingSystemObservable {
    private List<TicketObserver> observers = new ArrayList<>();
    private String ticketDetails;

    public void addObserver(TicketObserver observer) {
        observers.add(observer);
    }

    public void removeObserver(TicketObserver observer) {
        observers.remove(observer);
    }

    public void notifyObservers() {
        for (TicketObserver observer : observers) {
            observer.update(ticketDetails);
        }
    }

    public void bookTicket(String ticketDetails) {
        this.ticketDetails = ticketDetails;
        notifyObservers();
    }
}
```

6-Decorator Pattern

This code demonstrates the Decorator Pattern in a ticketing system. It starts with a basic ticket class containing essential details and pricing.

Additional features, like a Window Seat, are added ,dynamically through decorators enabling flexible and scalable enhancements without altering the core logic

```
    @Override
    public String getDetails() {
        return decoratedTicket.getDetails();
    }

    @Override
    public double getPrice() {
        return decoratedTicket.getPrice();
    }
}

class WindowSeatDecorator extends TicketDecorator {
    public WindowSeatDecorator(Ticket decoratedTicket) {
        super(decoratedTicket);
    }

    @Override
    public String getDetails() {
        return super.getDetails() + "\nFeature: Window Seat";
    }

    @Override
    public double getPrice() {
        return super.getPrice() + 5.0; // Additional cost for a window seat
    }
}

class MealIncludedDecorator extends TicketDecorator {
    public MealIncludedDecorator(Ticket decoratedTicket) {
        super(decoratedTicket);
    }

    @Override
    public String getDetails() {
        return super.getDetails() + "\nFeature: Meal Included";
    }
```

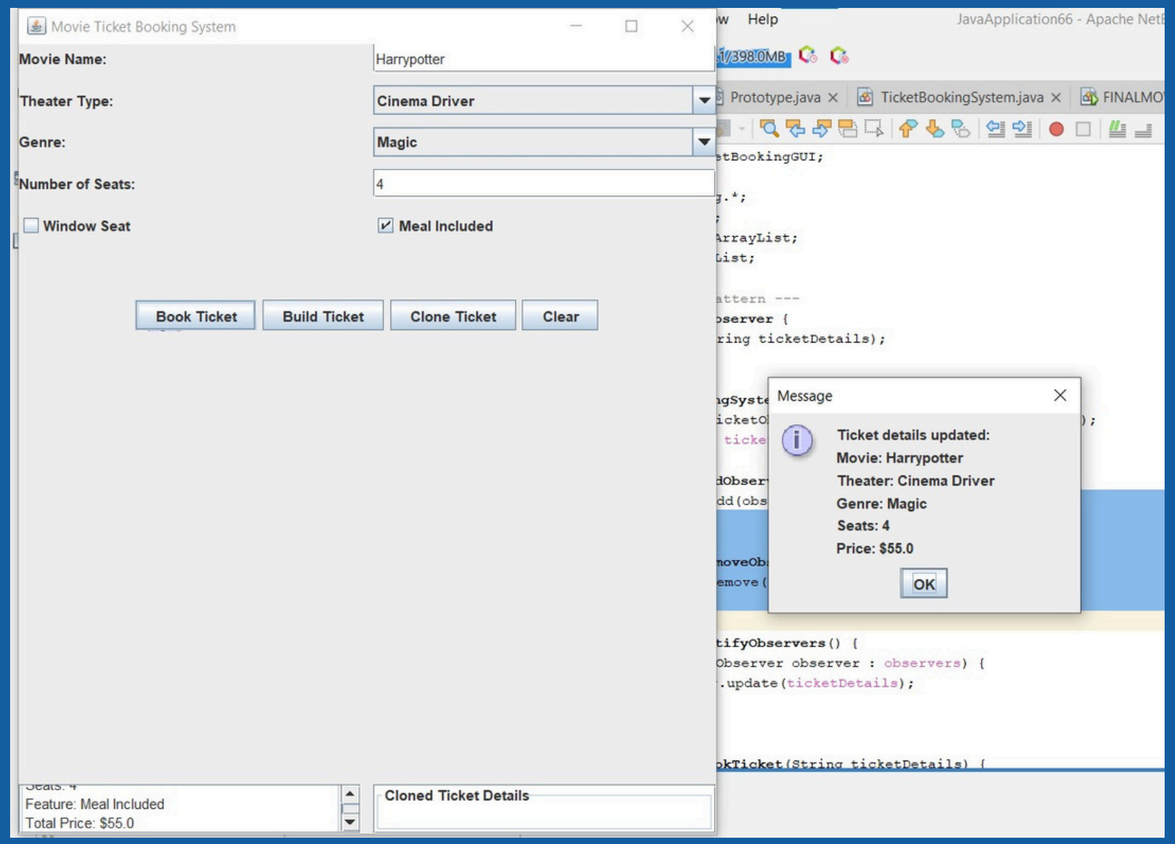
6-GUI

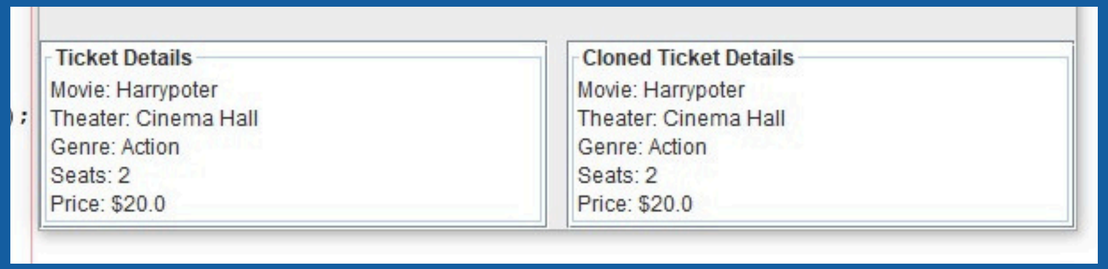
GUI for a movie ticket booking system.

Provides a user interface with options to input movie details, theater type, genre, and seats.

Features include booking functionality through a button interface.

Uses Swing components to create a structured layout with panels and interactive elements.





Thank You