Travel Management System

Team members:

1. Fady Amgad Naoum Andrawes (21-00667)
2. Shady mansour (21-01593)
3. Shady aziz (21-01592)
4. Aya gamal Mahmoud (21-00535)

Under Supervision of:

DR. Mahmoud Bassiouni

T.A Arwa Essam

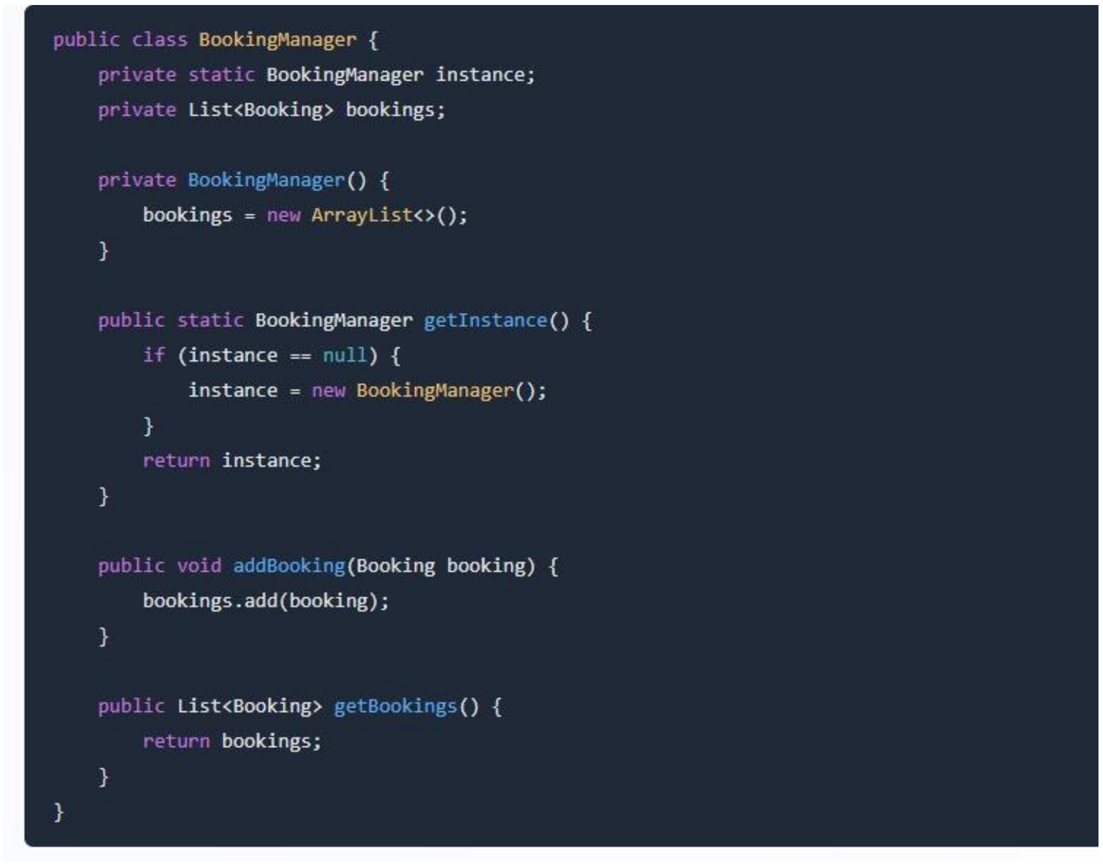
Introduction to the Project

This project is a Travel Management System that allows users to book flights, hotels, and packages. The system uses a combination of design patterns, including Singleton, Factory, and Adapter, builder to manage the booking process

Code Implementation

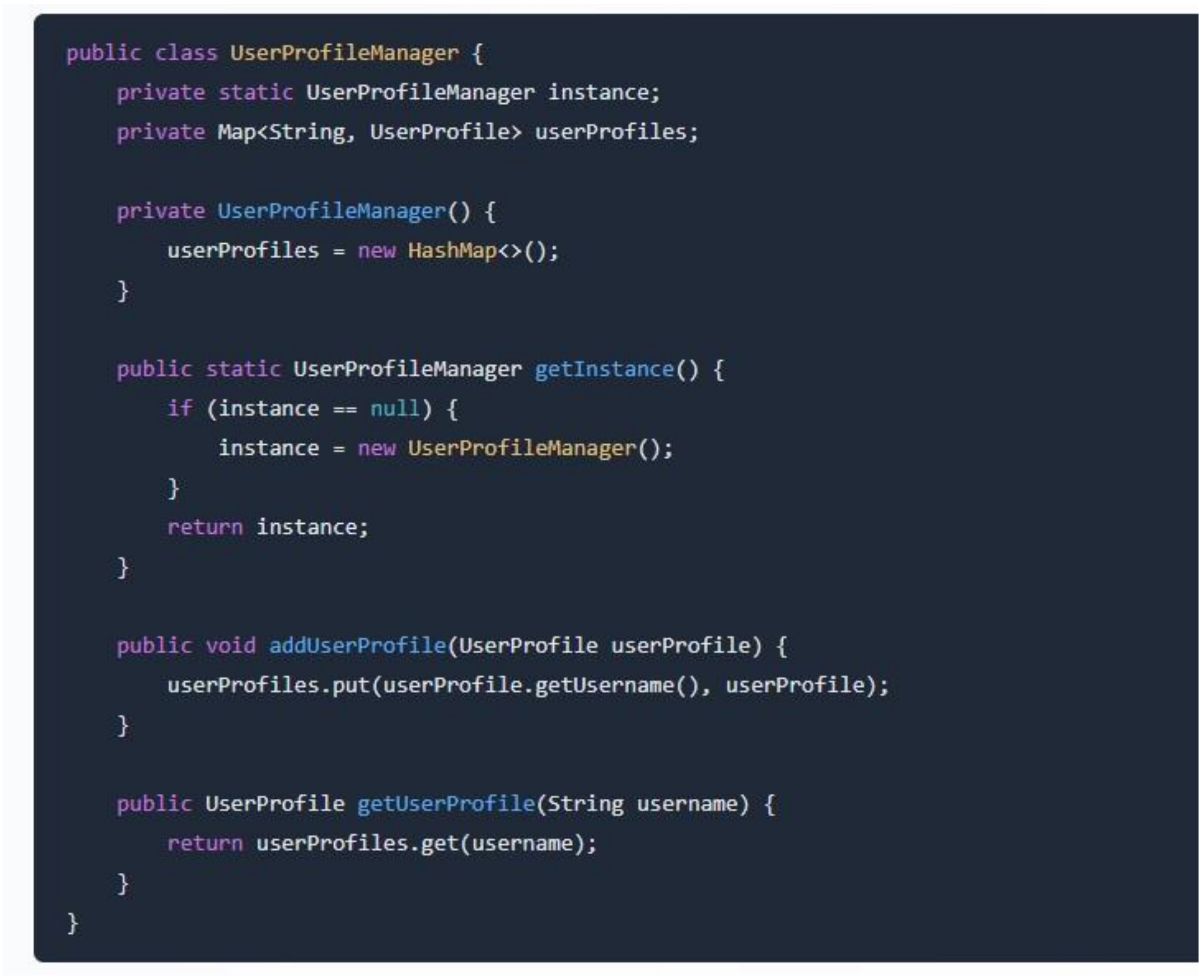
Singleton Pattern

The Singleton pattern is used to ensure that only one instance of the `BookingManager` and `UserProfileManager` classes are created.



```java
public class BookingManager {
    private static BookingManager instance;
    private List<Booking> bookings;

    private BookingManager() {
        bookings = new ArrayList<>();
    }

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

    public void addBooking(Booking booking) {
        bookings.add(booking);
    }

    public List<Booking> getBookings() {
        return bookings;
    }
}
```

- his code defines the `BookingManager` class, which is responsible for managing the booking process.
- The `getInstance` method is used to create a single instance of the `BookingManager` class.
- The `addBooking` method is used to add a new booking to the system.
- The `getBookings` method is used to retrieve a list of all bookings in the system.

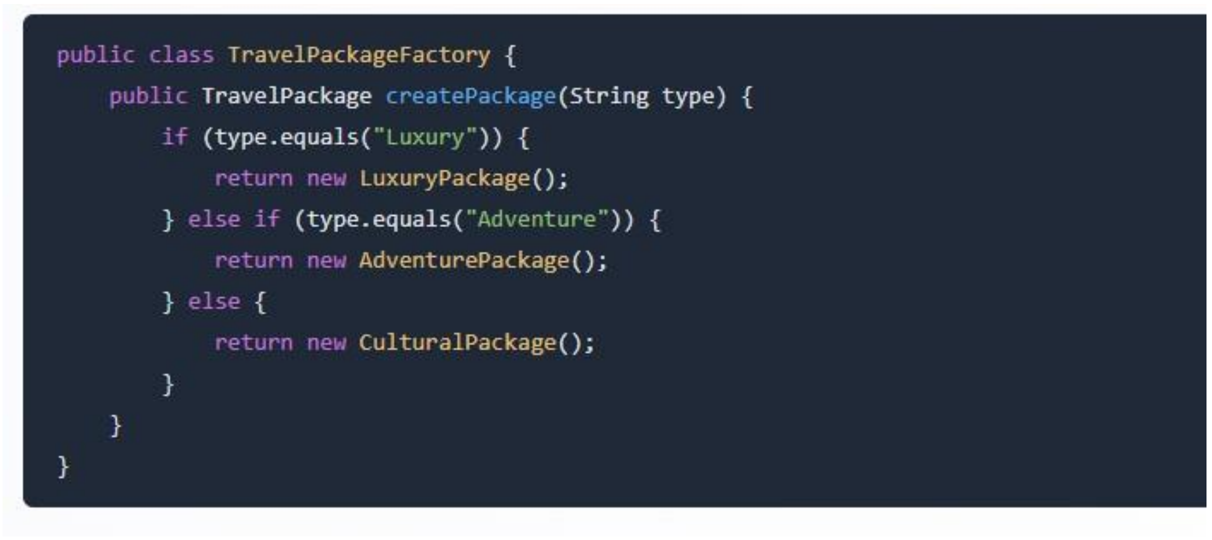


```java
public class UserProfileManager {
    private static UserProfileManager instance;
    private Map<String, UserProfile> userProfiles;

    private UserProfileManager() {
        userProfiles = new HashMap<>();
    }

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

    public void addUserProfile(UserProfile userProfile) {
        userProfiles.put(userProfile.getUsername(), userProfile);
    }

    public UserProfile getUserProfile(String username) {
        return userProfiles.get(username);
    }
}
```

- This code defines the `UserProfileManager` class, which is responsible for managing user profiles.
- The `getInstance` method is used to create a single instance of the `UserProfileManager` class.
- The `addUserProfile` method is used to add a new user profile to the system.
- The `getUserProfile` method is used to retrieve a user profile by username.

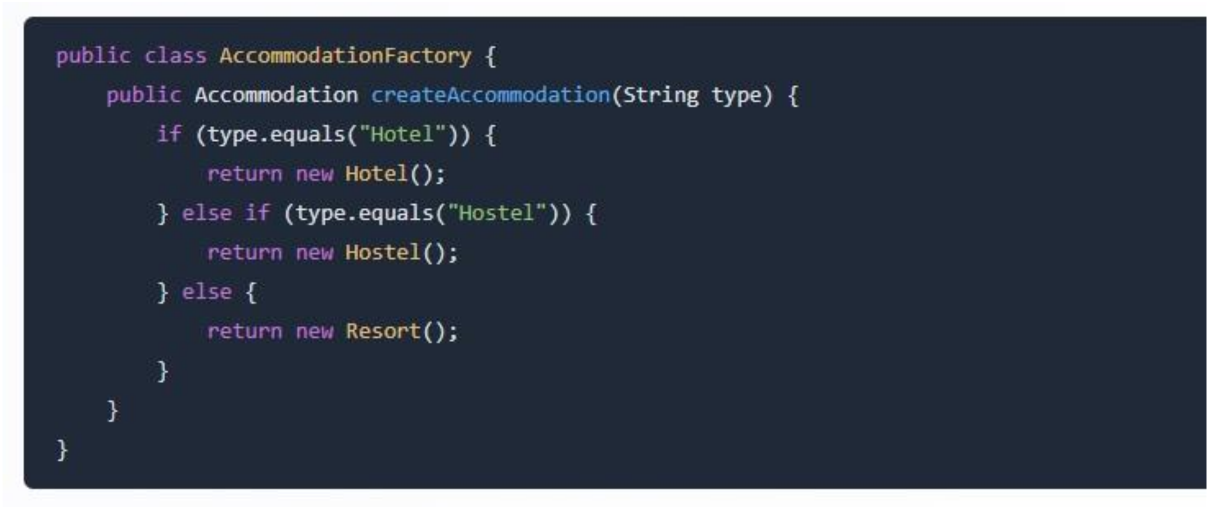
Factory Pattern

The Factory pattern is used to create objects without specifying the exact class of object that will be created.



```java
public class TravelPackageFactory {
    public TravelPackage createPackage(String type) {
        if (type.equals("Luxury")) {
            return new LuxuryPackage();
        } else if (type.equals("Adventure")) {
            return new AdventurePackage();
        } else {
            return new CulturalPackage();
        }
    }
}
```

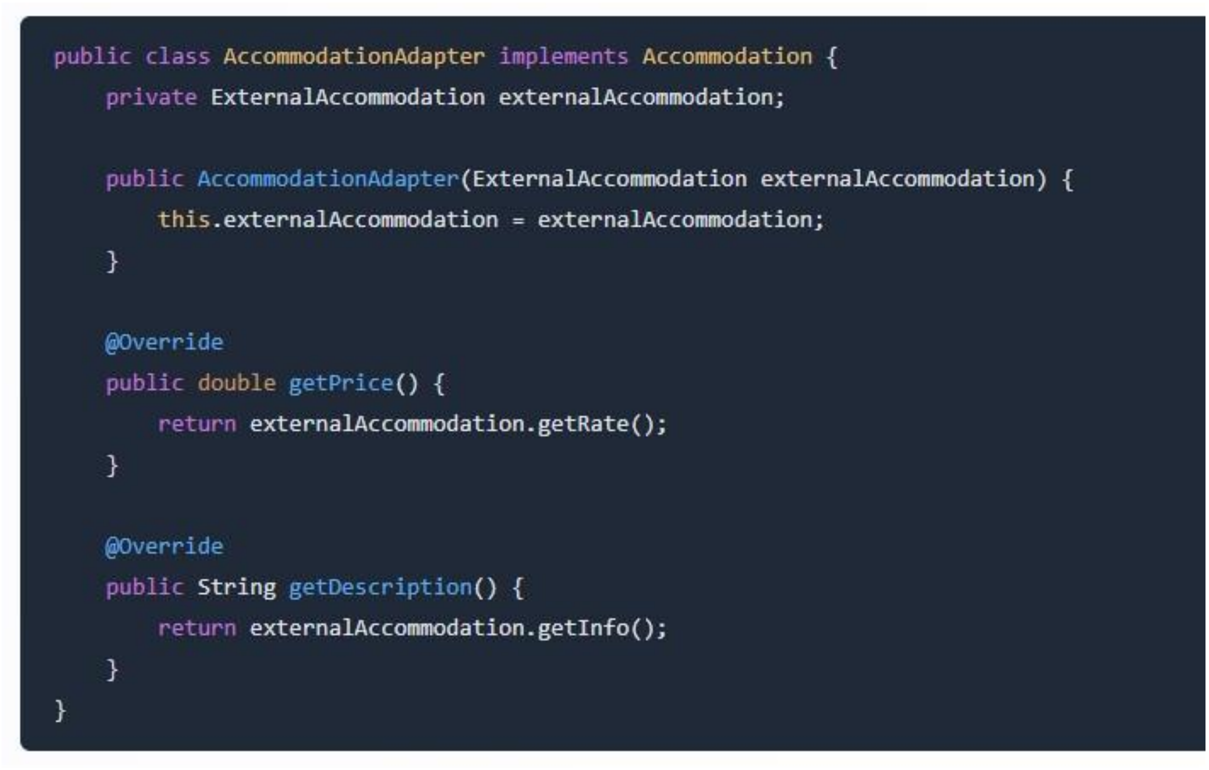
- This code defines the `TravelPackageFactory` class, which is responsible for creating travel packages.
- The `createPackage` method is used to create a new travel package based on the specified type.



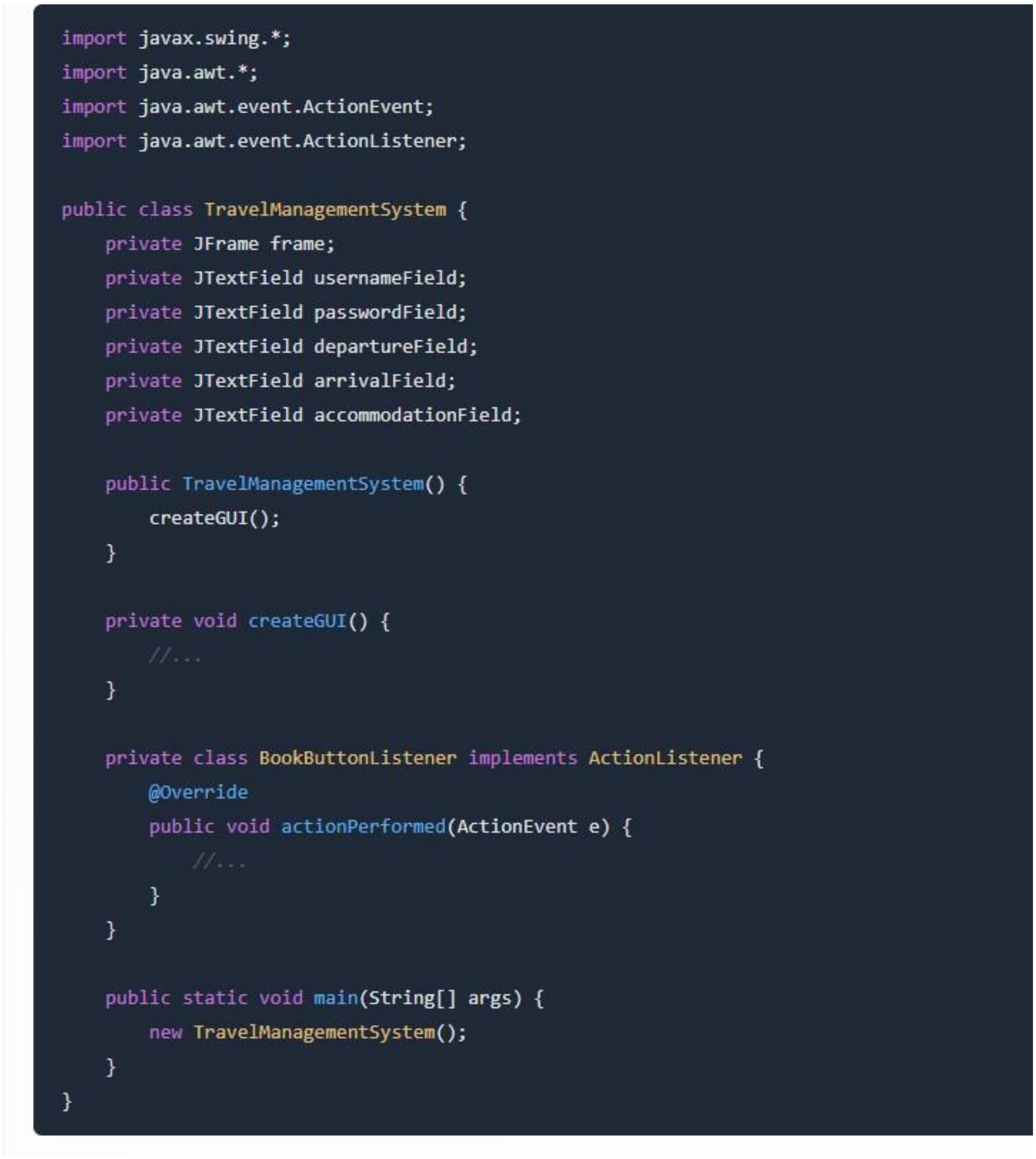
```java
public class AccommodationFactory {
    public Accommodation createAccommodation(String type) {
        if (type.equals("Hotel")) {
            return new Hotel();
        } else if (type.equals("Hostel")) {
            return new Hostel();
        } else {
            return new Resort();
        }
    }
}
```

- This code defines the `AccommodationFactory` class, which is used for creating Accommodations.
- The `createAccommodation` method is used for creating Accommodations based on type.

Adapter Pattern



```java
public class AccommodationAdapter implements Accommodation {
    private ExternalAccommodation externalAccommodation;

    public AccommodationAdapter(ExternalAccommodation externalAccommodation) {
        this.externalAccommodation = externalAccommodation;
    }

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

    @Override
    public String getDescription() {
        return externalAccommodation.getInfo();
    }
}
```



```java
import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class TravelManagementSystem {
    private JFrame frame;
    private JTextField usernameField;
    private JTextField passwordField;
    private JTextField departureField;
    private JTextField arrivalField;
    private JTextField accommodationField;

    public TravelManagementSystem() {
        createGUI();
    }

    private void createGUI() {
        //...
    }

    private class BookButtonListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            //...
        }
    }

    public static void main(String[] args) {
        new TravelManagementSystem();
    }
}
```

Conclusion

In conclusion, this project demonstrates the use of design patterns in a travel management system. The Singleton pattern is used to ensure that only one instance of the booking manager and user profile manager are created. The Factory pattern is used to create objects without specifying the exact class of object that will be created. The Adapter pattern is used to convert the interface of a class into another interface that clients expect. The GUI is used to interact with the user and display the booking options. The booking process involves creating a new booking object and adding it to the booking manager.