

TripArchitect

Laura Fulajtárová

Fakulta informatiky a informačných technológií STU v Bratislave

xfulajtarova@stuba.sk

ID: 120782

OOP – Streda 16:00

12.5.2023

Contents

General topic – Trip Planning	4
Project objective.....	4
System Structure	4
Main criteria	5
Inheritance	5
Polymorphism in at least two separate inheritance hierarchies	6
Static	6
Dynamic.....	6
Aggregation	8
Encapsulation	9
Further criteria	9
Application of design patterns.....	9
Factory pattern	10
Strategy pattern.....	10
Decorator pattern	11
Observer pattern	12
Handling exceptional states using own exceptions	13
Providing a graphical user interface separated from application logic and with at least part of the event handlers created manually	14
Explicit use of multithreading.....	14
Using generics in own classes.....	15
Explicit use of RTTI.....	15
Using nested classes and interfaces	16
Using lambda expressions or method reference.....	16
Using default method implementation in interfaces	17
Using serialization.....	18
Major program versions	18
1. Version - Commits on Mar 30, 2023.....	18
2. Version - Commits on Mar 31, 2023.....	19
3. Version - Commits on Apr 4, 2023.....	19
4. Version - Commits on May 2, 2023.....	19
5. Version - Commits on May 9, 2023.....	19
6. Version - Commits on May 12, 2023.....	19
Conclusion	20

General topic – Trip Planning

Trips can be very complex undertakings that can really benefit from software support. Often, they involve making detours to visit interesting sites in the vicinity of the main route. Different services and events can be offered at these sites. The sites may be recommended according to the traveler's needs and constraints. During the trip, the initial plan may be changed. Traveling in a group may involve synchronizing detours. Whether a trip plan is created by a professionals or by travelers themselves, it may be worthwhile sharing it with others.

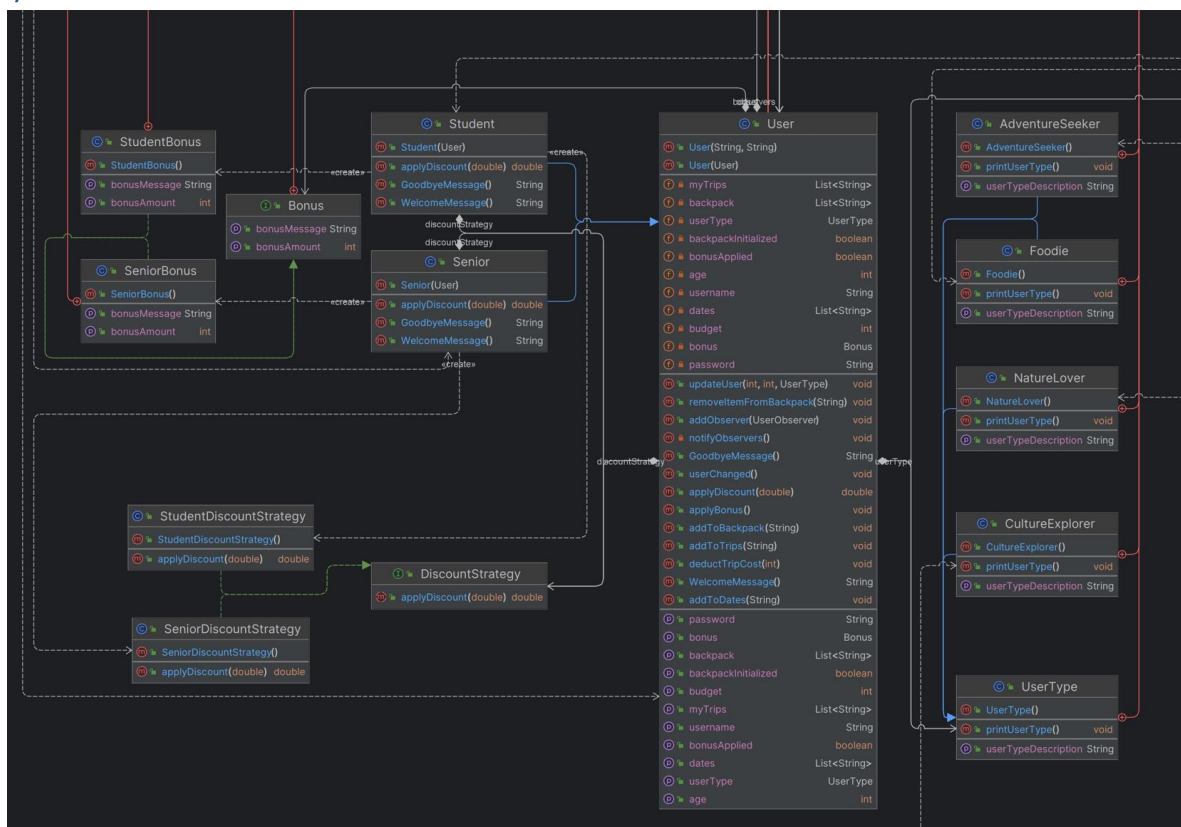
Project objective

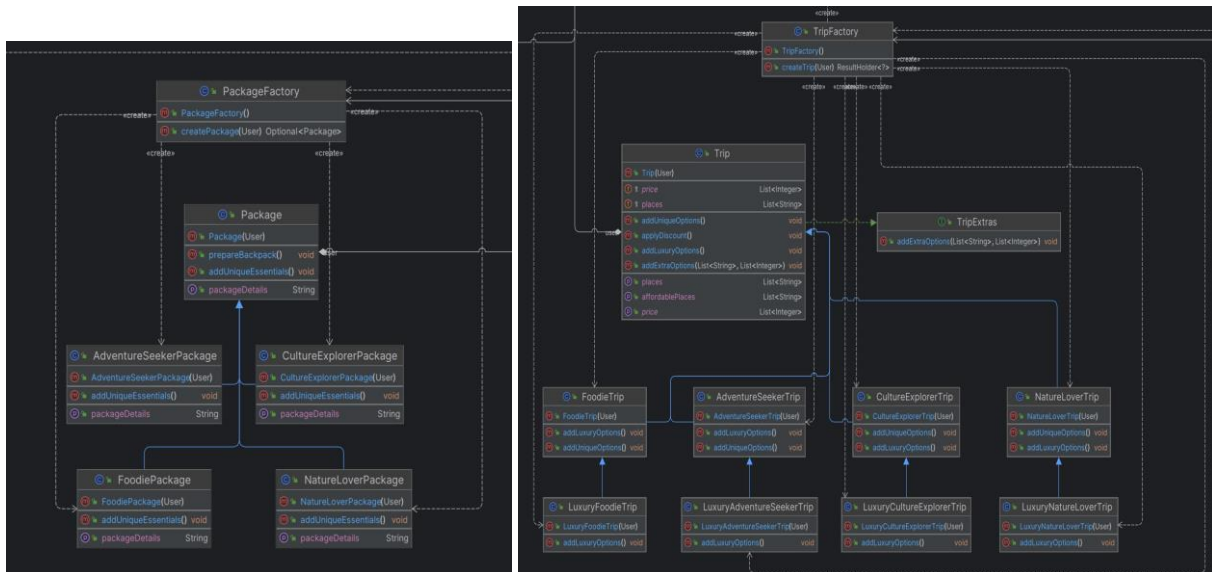
I aimed to create a user-friendly app to plan and organize trips. As a user, I would provide essential information during registration, which helps the app create tailored trips specifically for me. The app has four user types: "Adventure Seeker," "Culture Explorer," "Nature Lover," and "Foodie," allowing for customized experiences based on my interests.

The app also offers bonuses and discounts for students and seniors, and I can access VIP options by setting a higher budget. It gives me multiple destination choices, and I can plan, change, or delete my trip anytime. After creating my trips, I can share them with others.

The app generates a list of basic items for my backpack, and I can add more items as needed. This app simplifies trip planning and ensures I enjoy unforgettable experiences.

System Structure





*whole UML diagram can be found on GitHub

At the heart of the application structure is the user registration process. It's during this phase that the user identifies their travel type, forming the basis for the generation of customized trips and the packing of appropriate backpack items. When the user's budget exceeds 1000 euros, additional destination options become available.

Further enriching the structure are several user types that build upon the basic user: the 'Student' and 'Senior' categories. These special user types trigger the creation of unique travel destinations and backpack items. Additionally, these categories carry the perks of invitation bonuses and overall trip discounts, enhancing the user experience and value proposition.

This application goes the extra mile by enabling the addition of various trips and activities to the user's itinerary, including the scheduling of these events for seamless organization. Furthermore, these trip plans are shareable, fostering a sense of community among users.

The app's home window also showcases the user's backpack, featuring the initial package assembled during registration. For those who crave a more personalized experience, there's the option to add custom items to the backpack, ensuring every adventure is perfectly tailored to the user's needs.

*more information about each class and their metods can be found in JavaDoc on GitHub.

Main criteria

Inheritance

A concept where one class acquires or inherits the properties and behaviors of another class.

```

public class User implements Serializable {...}
public class Senior extends User {...}
public class Student extends User {...}

public abstract class Trip implements TripExtras {...}
public class NatureLoverTrip extends Trip {...}
public class FoodieTrip extends Trip {...}
public class CultureExplorerTrip extends Trip {...}
public class AdventureSeekerTrip extends Trip {...}
public class LuxuryNatureLoverTrip extends NatureLoverTrip {...}
public class LuxuryFoodieTrip extends FoodieTrip {...}
  
```

```

public class LuxuryCultureExplorerTrip extends CultureExplorerTrip {...}
public class LuxuryAdventureSeekerTrip extends AdventureSeekerTrip {...}

public class Package {
public class NatureLoverPackage extends Package {...}
public class FoodiePackage extends Package {...}
public class CultureExplorerPackage extends Package {...}
public class AdventureSeekerPackage extends Package {...}

```

Polymorphism in at least two separate inheritance hierarchies

An attribute that enables an entity to manifest in multiple forms, allowing diverse objects to be treated as instances of a superclass.

Static

```

public User(String username, String password) {
    ...
}
public User(User user) {
    ...
}

```

Dynamic

```

public abstract class Trip implements TripExtras {
    public void addUniqueOptions() {

    }
    public void addLuxuryOptions() {

    }
}

public class AdventureSeekerTrip extends Trip {
    @Override
    public void addUniqueOptions() {
        places.add("Bungee jumping");
        price.add(120);
        places.add("Water park");
        price.add(20);
    }
    @Override
    public void addLuxuryOptions() {

    }
}

public class LuxuryAdventureSeekerTrip extends AdventureSeekerTrip {
    @Override
    public void addLuxuryOptions() {
        places.add("Visit Machu Picchu");
        price.add(200);
        places.add("Visit the Grand Canyon");
        price.add(300);
    }
}

public class NatureLoverTrip extends Trip {
    @Override
    public void addUniqueOptions() {
        places.add("Camping");
        price.add(30);
        places.add("Paddle-boarding");
        price.add(15);
    }
    @Override

```

```

        public void addLuxuryOptions() {
        }

public class LuxuryNatureLoverTrip extends NatureLoverTrip {
    @Override
    public void addLuxuryOptions() {
        places.add("Visit the Great Barrier Reef");
        price.add(200);
        places.add("Visit the Stonehenge ");
        price.add(300);
    }
}

```

```

public class Package {
    public String getPackageDetails() {
        return null;
    }
    public void addUniqueEssentials() {
    }
}

public class NatureLoverPackage extends Package {
    @Override
    public String getPackageDetails() {
        return "Nature lover package includes binoculars and camping gear.";
    }
    @Override
    public void addUniqueEssentials() {
        essentials.add("Binoculars");
        essentials.add("Camping gear");
    }
}

public class FoodiePackage extends Package {
    @Override
    public String getPackageDetails() {
        return "Foodie package includes local cuisine guide and food tour tickets.";
    }
    @Override
    public void addUniqueEssentials() {
        essentials.add("Local cuisine guide");
        essentials.add("Protein bar");
    }
}

```

```

public static class UserType implements Serializable {
    public String getUserTypeDescription() {
        return null;
    }

    public void printUserType() {
    }
}

public static class AdventureSeeker extends UserType {
    @Override
    public String getUserTypeDescription() {
        return "Adventure seeker";
    }
    @Override
    public void printUserType() {
        System.out.println("User type: Adventure seeker");
    }
}

public static class CultureExplorer extends UserType {
    @Override
    public String getUserTypeDescription() {
        return "Culture explorer";
    }
}

```

```

    @Override
    public void printUserType() {
        System.out.println("User type: Culture explorer");
    }
}

```

```

public class User implements Serializable {
    public String WelcomeMessage() {
        return "Welcome!";
    }
    public String GoodbyeMessage() {
        return "Goodbye!";
    }
}

public class Senior extends User {
    @Override
    public String WelcomeMessage() {
        return "Welcome Senior!";
    }
    @Override
    public String GoodbyeMessage() {
        return "Goodbye Senior!";
    }
}

public class Student extends User {
    public Student(User user) {
        @Override
        public String WelcomeMessage() {
            return "Welcome Student!";
        }
        @Override
        public String GoodbyeMessage() {
            return "Goodbye Student!";
        }
    }
}

```

Aggregation

A relationship between classes characterized by one class being a complex collection or composition of other classes.

```

public class User implements Serializable {
    private UserType userType;
    private DiscountStrategy discountStrategy;
    private Bonus bonus;
}

```

```

public abstract class Trip implements TripExtras {
    protected static User user;
}

```

```

public class Package {
    private final User user;
}

```

```

public abstract class ItemDecorator {
    protected Item item;
}

```

```

public class HomeController {
    private final PackageFactory packageFactory;
    private final TripFactory tripFactory;
    private User user;
    private Weather randomWeather;
    private WeatherService weatherService;
}

```


Encapsulation

A practice of bundling the data and the methods that operate on the data into a single unit, restricting direct access to private data.

```
public class User implements Serializable {
    private final String username;
    private final String password;
    private final List<String> backpack;
    private final List<String> myTrips;
    private final List<String> dates;
    private UserType userType;
    private int budget;
    private int age;
    private boolean backpackInitialized;
    private boolean bonusApplied;

    public String getUsername() {
        return username;
    }
    public String getPassword() {
        return password;
    }
    public UserType getUserType() {
        return userType;
    }
    public int getBudget() {
        return budget;
    }
    public void setBudget(int budget) {
        this.budget = budget;
    }
    public int getAge() {
        return age;
    }
    public boolean isBackpackInitialized() {
        return backpackInitialized;
    }
    public void setBackpackInitialized(boolean backpackInitialized) {
        this.backpackInitialized = backpackInitialized;
    }
    public List<String> getBackpack() {
        return backpack;
    }
    public boolean isBonusApplied() {
        return bonusApplied;
    }
    public void setBonusApplied(boolean bonusApplied) {
        this.bonusApplied = bonusApplied;
    }
    public List<String> getDates() {
        return dates;
    }
    public List<String> getMyTrips() {
        return myTrips;
    }
}
```

Further criteria

Application of design patterns

The act of implementing standardized solutions to commonly occurring design challenges.

Factory pattern

A design pattern providing an interface for object creation but defers the instantiation process to subclasses. Used to create different trips and packages based on the user type and his budget.

```
public class TripFactory {
    public ResultHolder<?> createTrip(User user) {
        String userType = user.getUserType().getUserTypeDescription();

        Trip userTrip = switch (userType) {
            case "Adventure seeker" ->
                user.getBudget() > 1000 ? new LuxuryAdventureSeekerTrip(user) :
new AdventureSeekerTrip(user);
            case "Culture explorer" ->
                user.getBudget() > 1000 ? new LuxuryCultureExplorerTrip(user) :
new CultureExplorerTrip(user);
            case "Nature lover" ->
                user.getBudget() > 1000 ? new LuxuryNatureLoverTrip(user) : new
NatureLoverTrip(user);
            case "Foodie" -> user.getBudget() > 1000 ? new LuxuryFoodieTrip(user) :
new FoodieTrip(user);
            default -> null;
        };

        if (userTrip != null) {
            return new ResultHolder<>(userTrip);
        } else {
            return new ResultHolder<>("Unable to create a trip for this user
type.");
        }
    }
}

private ResultHolder<?> generateTrip() {
    return tripFactory.createTrip(user);
}
```

```
public class PackageFactory {
    public Optional<Package> createPackage(User user) {
        String userType = user.getUserType().getUserTypeDescription();
        Package userPackage = switch (userType) {
            case "Adventure seeker" -> new AdventureSeekerPackage(user);
            case "Culture explorer" -> new CultureExplorerPackage(user);
            case "Nature lover" -> new NatureLoverPackage(user);
            case "Foodie" -> new FoodiePackage(user);
            default -> null;
        };

        return Optional.ofNullable(userPackage);
    }
}

private Optional<Package> generatePackage() {
    return packageFactory.createPackage(user);
}
```

Strategy pattern

A design pattern that enables a strategy or an algorithm's behavior to be selected at runtime. Used to add discount on trips for student or senior.

```
public interface DiscountStrategy extends Serializable {
    double applyDiscount(double originalPrice);
}
```

```

public class SeniorDiscountStrategy implements DiscountStrategy {
    public double applyDiscount(double originalPrice) {
        return originalPrice * 0.7;
    }
}

public class StudentDiscountStrategy implements DiscountStrategy {
    public double applyDiscount(double originalPrice) {
        return originalPrice * 0.8;
    }
}

User class
private DiscountStrategy discountStrategy;
public double applyDiscount(double originalPrice) {
    if (discountStrategy != null) {
        return discountStrategy.applyDiscount(originalPrice);
    }
    return originalPrice;
}

Senior class
public Senior(User user) {
    super(user);
    setBonus(new SeniorBonus());
    applyBonus();
    discountStrategy = new SeniorDiscountStrategy();
    userChanged();
}

@Override
public double applyDiscount(double price) {
    return discountStrategy.applyDiscount(price);
}

```

Decorator pattern

A design pattern used to add new functionality to an existing object without altering its structure. Used when user creates new item to the backpack.

Backpack
Sandwich
Water
Local cuisine guide
Protein bar
Apple (personalized)

```

public class Item {
    private final String description;
    private final int price;
    public Item(String description, int price) {
        this.description = description;
        this.price = price;
    }
    public String getDescription() {
        return description;
    }
    public int getPrice() {
        return price;
    }
}

public abstract class ItemDecorator {
    protected Item item;
    public ItemDecorator(Item item) {
        this.item = item;
    }
    public abstract String getDescription();
    public abstract int getPrice();
}

```

```

public class PersonalizedItem extends ItemDecorator {
    public PersonalizedItem(Item item) {
        super(item);
    }
    public String getDescription() {
        return item.getDescription() + " (personalized)";
    }
    public int getPrice() {
        return item.getPrice();
    }
}

private void handleConfirm() throws IOException {
    ...
    Item baseItem = new Item(itemDescription, price);
    ItemDecorator item = new PersonalizedItem(baseItem);
    ...
}

```

Observer pattern

A design pattern where an object maintains a list of its dependents and notifies them of any changes. Used to inform when user changed to student or senior user during registration.

```

private List<UserObserver> observers = new ArrayList<>();

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

public void userChanged() {
    notifyObservers();
}

private void notifyObservers() {
    for (UserObserver observer : observers) {
        observer.onUserUpdated(this);
    }
}

public interface UserObserver extends Serializable {
    void onUserUpdated(User user);
}

public static class UserNotification implements UserObserver {
    public void onUserUpdated(User user) {
        if (user instanceof Student) {
            System.out.println("New student user created: " + user.getUsername());
        } else if (user instanceof Senior) {
            System.out.println("New senior user created: " + user.getUsername());
        }
    }
}

private void handleConfirm() {
    User.UserObserver userNotification = new
        User.UserNotification();
    user.addObserver(userNotification);
    ...}

```

Handling exceptional states using own exceptions

The process of managing error situations through custom exceptions defined by the developer. Used during registration when user checks that he is student/senior/both and set age out of this range. The message will be then displayed.

```
public class RegisterController {
    private void handleConfirm() {
        // exception for choosing both student and senior
        try {
            if (isStudent && isSenior) {
                throw new YouCantBeStudentAndSenior("You can't be a student and
senior \n at the same time.");
            }
        } catch (YouCantBeStudentAndSenior e) {
            outputLabel.setText(e.getMessage());
            return;
        }

        // exception for choosing a student
        try {
            if (isStudent) {
                if (age > 26) {
                    throw new YouCantBeStudent("You can't be a student if you are
older than 26.");
                }
                user = new Student(user);
            }
        } catch (YouCantBeStudent e) {
            outputLabel.setText(e.getMessage());
            return;
        }

        // exception for choosing senior
        try {
            if (isSenior) {
                if (age < 60) {
                    throw new YouCantBeSenior("You can't be a senior if you are
younger than 60.");
                }
                user = new Senior(user);
            }
        } catch (YouCantBeSenior e) {
            outputLabel.setText(e.getMessage());
            return;
        }
    }
}

public class YouCantBeSenior extends IOException {
    public YouCantBeSenior(String message) {
        super(message);
    }
}

public class YouCantBeStudent extends Exception {
    public YouCantBeStudent(String message) {
        super(message);
    }
}

public class YouCantBeStudentAndSenior extends Exception {
    public YouCantBeStudentAndSenior(String message) {
        super(message);
    }
}
```

Providing a graphical user interface separated from application logic and with at least part of the event handlers created manually

Creating a visual way for users to interact with a system, separated from the underlying application logic.

In my project each GUI view has a “Controller” associated with it.

```
@FXML
public void initialize() {
    newTripButton.setOnAction(event -> {
        try {
            openCarAnimationWindow();
            addSelectedTrip();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    });

    editTripsButton.setOnAction(event -> {
        try {
            openEditTripsWindow();
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
    });
}

private void openCarAnimationWindow() throws IOException {
    FXMLLoader loader = new FXMLLoader(getClass().getResource("car.fxml"));
    Parent root = loader.load();

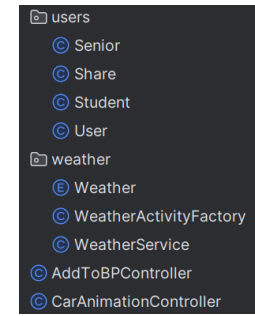
    Stage animationStage = new Stage();

    animationStage.setTitle("Car Animation");
    animationStage.setScene(new Scene(root));
    animationStage.showAndWait();
}

private void openEditTripsWindow() throws IOException {
    FXMLLoader loader = new FXMLLoader(getClass().getResource("edit_trips.fxml"));
    Parent editTripsRoot = loader.load();
    EditTripsController editTripsController = loader.getController();
    editTripsController.set(user);

    Stage stage = new Stage();
    stage.setTitle("Edit Trips");
    stage.setScene(new Scene(editTripsRoot));
    stage.show();

    Stage homeStage = (Stage) editTripsButton.getScene().getWindow();
    homeStage.close();
}
```



Explicit use of multithreading

Deliberately coding for multiple threads, or separate paths of execution, within a program. Creating and displaying weather is done on another thread than main program.

```
public class HomeController {
    ...
    private void fetchWeather() {
        Task<Weather> weatherTask = new Task<>() {
            protected Weather call() {
                weatherService = new WeatherService();
            }
        };
        ...
    }
}
```

```

    };
    ...
    Thread weatherThread = new Thread(weatherTask);
    weatherThread.setDaemon(true);
    weatherThread.start();
}

public class WeatherService {
    public Weather getWeather() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return Weather.getRandomWeather();
    }
}

```

Using generics in own classes

Implementing generics to create classes that can work with different data types. Use to store information about usertype during registration.

```

public class HomeController {
    private void fillComboBox() {
        ResultHolder<?> tripResult = generateTrip();
        ...

        private ResultHolder<?> generateTrip() {
            return tripFactory.createTrip(user);
        }
}

public record ResultHolder<T>(T result) {
    public ResultHolder {
    }
    public T result() {
        return result;
    }
}

```

Explicit use of RTTI

Deliberate use of Run-Time Type Information, a mechanism for finding out the type of an object during runtime.

```

public Package(User user) {
    if (user instanceof Student) {
        essentials.add("Movie tickets (student benefit)");
    }
}

public Trip(User user) {
    if (user instanceof Student) {
        places.add("Cinema");
        price.add(0);
    }
}

```

When user is student then he will have new option for the trip and also new item in his backpack.

```

public static class UserNotification implements UserObserver {
    public void onUserUpdated(User user) {
        if (user instanceof Student) {
            System.out.println("New student user created: " + user.getUsername());
        } else if (user instanceof Senior) {
            System.out.println("New senior user created: " + user.getUsername());
        }
    }
}

```

If user is student/senior then appropriate message will be displayed.

```
public void setBenefitLabel() {
    String Status = "None";
    if (user instanceof Student) {
        Status = "Student discount 20%";
    }

    if (user instanceof Senior) {
        Status = "Senior discount 30%";
    }
    benefitLabel.setText(Status);
}
```

Setting different labels for student/senior/normal user.

Using nested classes and interfaces

The act of defining a class or an interface inside another class or interface.

```
public class User implements Serializable {
    ...
    public interface Bonus extends Serializable {
        ...
    }

    public interface UserObserver extends Serializable {
        ...
    }

    public static class StudentBonus implements Bonus {
        ...
    }

    public static class SeniorBonus implements Bonus {
        ...
    }

    public static class UserNotification implements UserObserver {
        ...
    }

    public static class UserType implements Serializable {
        ...
    }

    public static class AdventureSeeker extends UserType {
        ...
    }

    public static class CultureExplorer extends UserType {
        ...
    }
}
```

Using lambda expressions or method reference

Implementing anonymous functions or referring to methods directly, often for more concise and functional-style code.

```
public void applyDiscount() {
    price.replaceAll(originalPrice -> (int) user.applyDiscount(originalPrice));
}
```

This lambda expression takes each original price in the price list and applies a discount to it using the applyDiscount method of the user object.


```
public List<String> getAffordablePlaces() {
    int userBudget = user.getBudget();
    return places.stream().filter(place -> {
        int index = places.indexOf(place);
        return price.get(index) <= userBudget;
    }).collect(Collectors.toList());
}
```

This lambda expression is used to filter places based on a condition. For each place in the places list, it checks whether the price of that place is less than or equal to the user's budget.

```
private void setBpInfoLabel() {
    Optional<Package> userPackage = generatePackage();
    userPackage.ifPresent(pkg -> {
        if (!user.isBackpackInitialized()) {
            pkg.prepareBackpack();
            user.setBackpackInitialized(true);
        }
        bpInfoLabel.setText(pkg.getPackageDetails());
    });
}
```

This lambda expression is executed if the userPackage is present. It prepares the backpack, sets the backpack as initialized for the user, and sets the package details to a label.

```
private void fetchWeather() {
    ...
    weatherTask.setOnSucceeded(event -> {
        Weather weather = (Weather) event.getSource().getValue();
        weatherLabel.setText("Current weather: " + weather);
        fillActivityComboBox();
    });
    ...
}
```

This lambda expression is used to handle the action when the weatherTask has succeeded. It fetches the weather data from the task's event source, sets the weather information to a label, and fills the activity combo box.

Using default method implementation in interfaces

Providing default behavior for a method in an interface.

```
public interface TripExtras {
    default void addExtraOptions(List<String> places, List<Integer> prices) {
        places.add("LSD trip");
        prices.add(30);
    }
}

public abstract class Trip implements TripExtras {
    @Override
    public void addExtraOptions(List<String> places, List<Integer> prices) {
        places.add("City tour");
        prices.add(10);

        places.add("Boat trip");
        prices.add(22);

        places.add("Beach trip");
        prices.add(5);

        places.add("Ski trip");
        prices.add(60);
    }
}
```

Using serialization

Converting an object's state into a byte stream for storage or transmission. Serializing user object and shared trips object.

```
public class SerializationUtils {
    public static void serializeUser(User user) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(user.getUsername() + ".ser"))) {
            oos.writeObject(user);
        }
    }

    public static User deserializeUser(String username) throws IOException,
ClassNotFoundException {
        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(username + ".ser"))) {
            return (User) ois.readObject();
        }
    }

    public static void serializeShare(Share share) throws IOException {
        try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("shared_trips.ser"))) {
            oos.writeObject(share);
        }
    }

    public static Share deserializeShare() throws IOException,
ClassNotFoundException {
        File file = new File("shared_trips.ser");
        if (!file.exists()) {
            return new Share();
        }

        try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(file))) {
            return (Share) ois.readObject();
        }
    }
}

public class User implements Serializable {
}

private void handleLogout() throws IOException {
    System.out.println(user.GoodbyeMessage());
    try {
        SerializationUtils.serializeUser(user);
    } catch (IOException e) {
        throw new RuntimeException("Error: Unable to save user data.", e);
    }
}
```

Major program versions

1. Version- Commits on Mar 30, 2023

Kickstarted the project by building the core user login and registration system along with their associated views.

2. Version- Commits on Mar 31, 2023

Implemented a detailed registration process where users share their age, budget, and travel type. Post-registration, users are navigated to a customized home screen featuring tailored trips and a pre-populated backpack.

3. Version- Commits on Apr 4, 2023

Introduced a 'Student' user category with additional trip options, specialized backpack items, and all-inclusive discounts on their adventures.

4. Version- Commits on May 2, 2023

Rolled out a 'Senior' user category with exclusive bonuses and discounts. Enhanced app functionality with more OOP patterns, lambda expressions, and multithreading. Luxury trip options now available for users with a budget exceeding 1000. Also incorporated real-time weather updates for activity selection.

5. Version- Commits on May 9, 2023

Empowered users with the ability to schedule trips, add custom items to their backpack, and modify trip details like changing dates, deleting trips, or adding nearby activities.

6. Version- Commits on May 12, 2023

Made trips shareable to promote social interaction. Enhanced code architecture with the application of polymorphism, implicit implementation, and nested classes. Added comprehensive Javadoc comments and generated a UML diagram. Undertook a rigorous code cleanup to eliminate redundancy and improve readability.

Conclusion

In conclusion, my project on Trip Planning has been a challenging and rewarding journey. I am proud to have achieved my objective in creating a user-friendly application that effectively simplifies the intricate process of trip planning. This app uniquely tailors experiences to individual users, making it a reliable and personal tool for all travel enthusiasts.

Throughout this project, I have successfully applied various fundamental principles of Object-Oriented Programming (OOP), including inheritance, polymorphism, encapsulation, and aggregation. These principles ensuring a well-structured and efficiently organized code.

Furthermore, I have managed to implement numerous design patterns in my application. The application handles exceptions using customized exceptions and uses generics in its classes. The incorporation of Runtime Type Information (RTTI) and multithreading, alongside the application of nested classes and interfaces, has further bolstered the robustness and versatility of the program.

The application also utilizes lambda expressions, method references, and default method implementations in interfaces. Although I did not incorporate aspect-oriented programming, I have successfully implemented serialization in the application.

Overall, I am pleased with the outcome of my project. This journey has not only helped me solidify my understanding of OOP concepts but has also allowed me to see their practical application in real-world software development. I am excited about the potential impact of this application in making trip planning more accessible and personalized for users.