City University of Hong Kong
2024-2025 Semester A
CS3343 Software Engineering Practice

Analysis and Design Report
Group 38
Hong Kong Journey Planner

Conducted by:

Name	**Position**
Fan Tianrui	Project Manager
Wang Fan	Assistant Project Manager
GAO Nanjie	Developing Analyst
CHEUNG Lok Yi	Testing Engineer
LIU Hengche	Program Developer

Department of Computer Science
香港城市大學
City University of Hong Kong

Dec 4, 2024

Table of content

1. Introduction

Hong Kong has lots from historical sites to restaurants to unforgettable scenic spots, attracting large numbers of visitors all over the world. However, due to the crowded nature of the city and the massive activities involved, many first-time visitors usually find travel planning challenging. This project simplifies this process by developing a system that will suit the needs of visitors, making it easier for visitors to plan their visit to Hong Kong. The application will generate optimized routes showing recommended locations and activities based on user preferences and specific dates. The app will enhance the travel experience by offering personalized route choices to help visitors make the most of their time in Hong Kong. It enhances satisfaction but also encourages exploration of the various landmarks that the city has to offer.

2. Use Case Diagram and Specification

2.1 Client-User Use Case

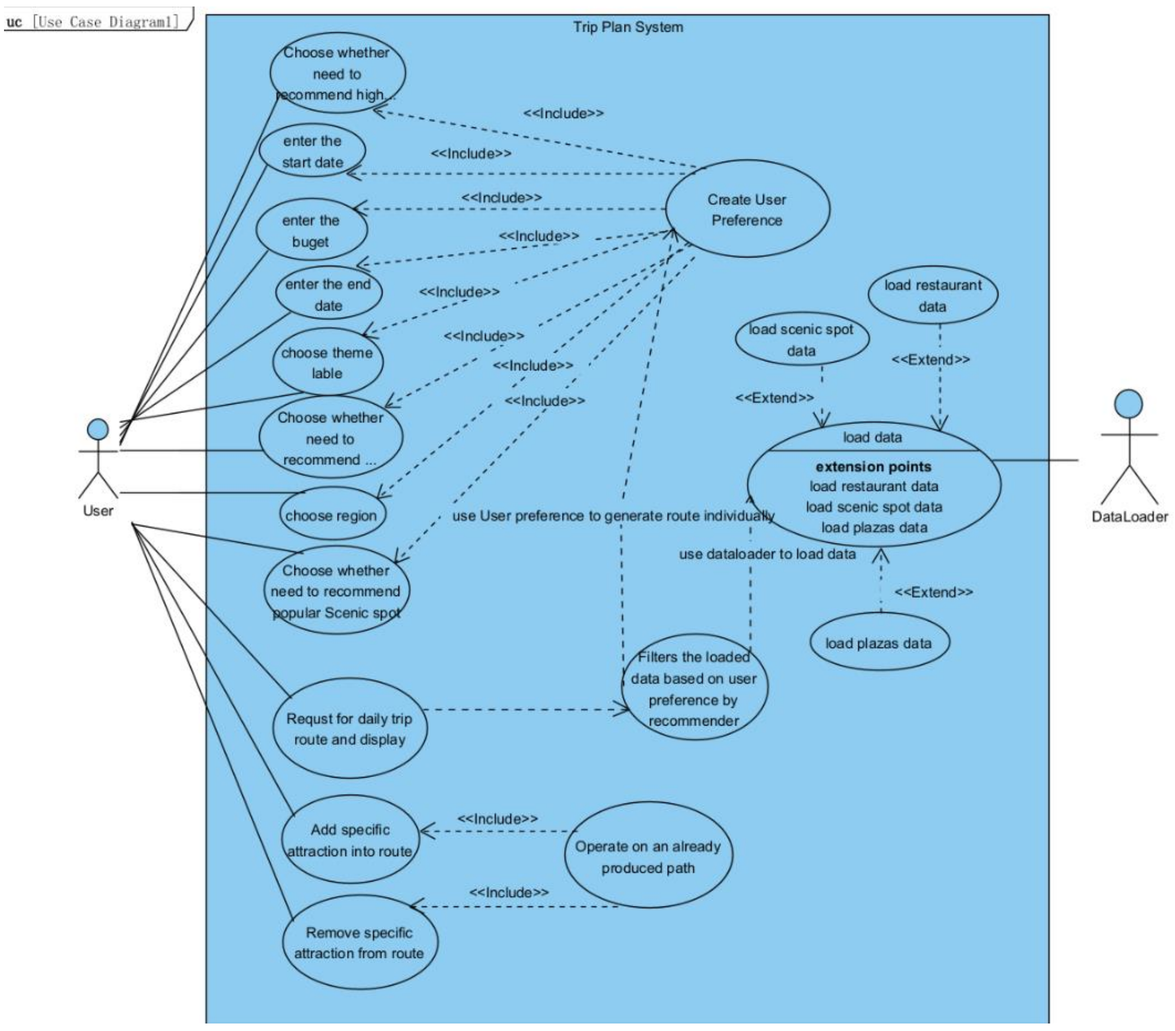

uc [Use Case Diagram1]
Trip Plan System
Choose whether need to recommend high...
enter the start date
enter the buget
enter the end date
choose theme lable
Choose whether need to recommend ...
choose region
Choose whether need to recommend popular Scenic spot
Requst for daily trip route and display
Add specific attraction into route
Remove specific attraction from route
Create User Preference
<<Include>>
load restaurant data
load scenic spot data
<<Extend>>
load data
extension points
load restaurant data
load scenic spot data
load plazas data
load plazas data
use User preference to generate route individually
use dataloader to load data
Filters the loaded data based on user preference by recommender
Operate on an already produced path
User
DataLoader

Figure 2.1 Client-User user case diagram

Use case diagrams can illustrate the interactions between the user and the system, as well as the manner in which these interactions occur. The user use case has three primary modules: first, inputting various individual requirements to create a UserPreference; second, soliciting the development of recommended pathways; and third, executing instructions to add or delete certain attractions from the created paths.

2.1.1 Use Case Specification – Create UserPreference

Use Case Name	Create UserPreference	
Actor(s)	User	
Description	This use case is designed to allow the user to enter personal requirements and generate the UserPreference	
Typical course of events	Actor action	System Response
	Step 1: Choose whether need to recommend high rating plazas Step 2: enter the start dateenter the start date Step 3: enter the buget Step 4: enter the end date Step 5: choose theme lable Step 6: Choose whether need to recommend popular plazas Step 7: choose region Step 8: Choose whether need to recommend popular Scenic spot	Step 9: The system creates a UserPreference
Alternate Course	**Step 9 (User Preference Creation)**: If some of the user input is missing or invalid, the system can prompt for the missing information and provide a summary of the preferences before generating the final user profile.	
Precondition	The user has no Userpreference before.	
Postcondition	The User have a new Userpreference, and the system will act according to this	

2.1.2 Use Case Specification – Request for daily trip route and display

Use Case Name	Request for daily trip route and display	
Actor(s)	User	
Description	This use case describe the process of a User request for daily trip route, and display it.	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user request for route.	
		Step 2: The system calls Dataloader to read all the attractions.
		Step 3: The system calls Userpreference to filter all attractions and get the ones that match the requirements.
		Step 4: The system displays the attractions.
	Step 5: Choose the final attractions to visit.	
		Step 5: The system get the final attractions and generate the route using the algorithm.
Alternate Course	**Step 1 (Initiation)**: If the user does not specify any preferences, the system can prompt them with default options or a list of popular attractions to choose from. **Step 2 (Data Loading)**: If Dataloader encounters missing or outdated data for attractions, it will attempt to retrieve the latest information from a backup source or display a notification to the user about unavailable data.	
Precondition	The users have already created their own Userpreference	
Postcondition	The User gets the wanted rentable.	

2.1.3 Use Case Specification - Add or Remove the specific attraction

Use Case Name	Add or Remove the specific attraction	
Actor(s)	User	
Description	This use case describe the process of using commands to add or remove the specific attraction	
Typical course of events	Actor action	System Response
	Step 1: This use case is initiated when the user apply the commands Step 2: The user select a specific attraction in the route. Step 3: Choose to add or remove	Step5: The system regenerate a route according to user's choise
Alternate Course	**Step 3 (Add/Remove)**: If the user's request to add or remove an attraction is invalid (e.g., the attraction is not part of the route), the system can provide an error message or suggest alternative actions, like selecting another attraction or adjusting the route.	
Precondition	The system has already generated aroute	
Postcondition	The User get a new route.	

3. Class Diagram

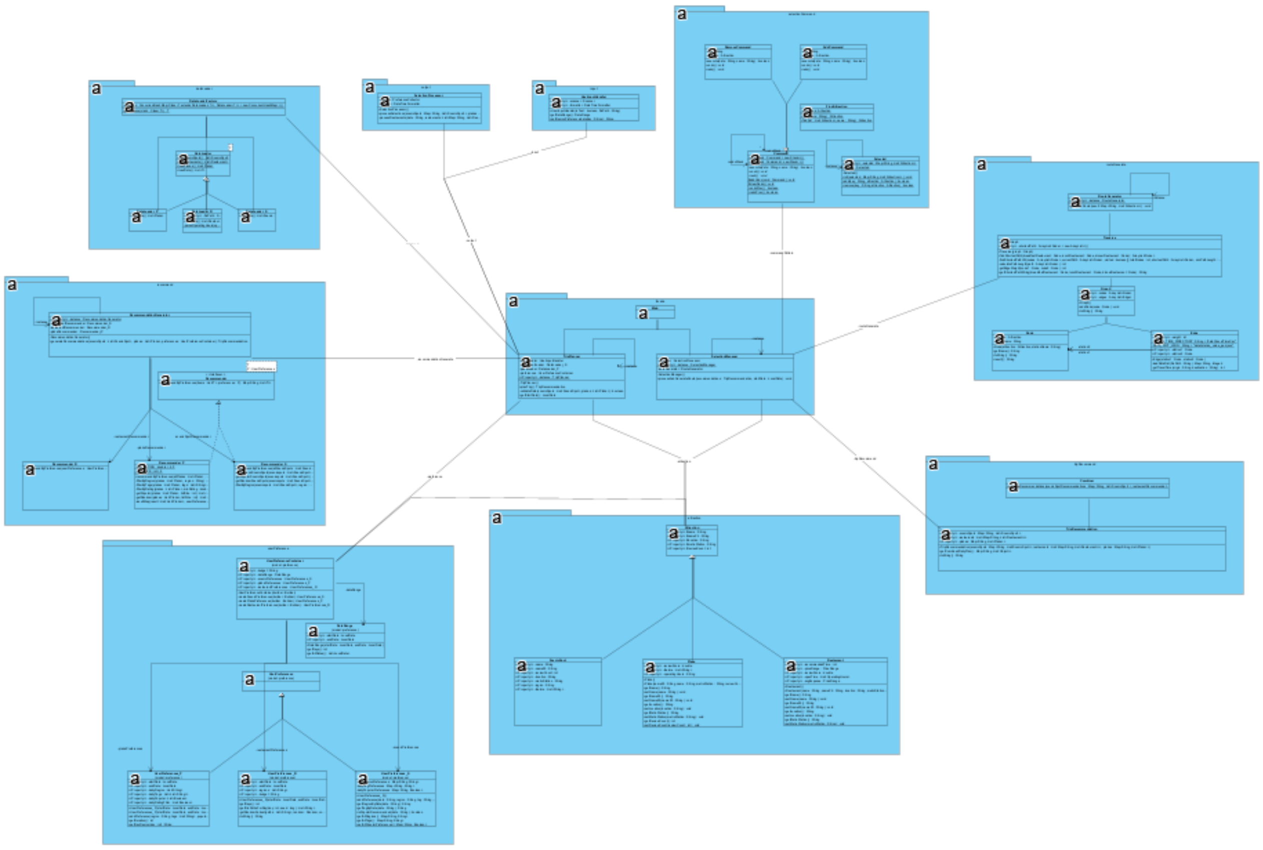


Figure 3 Class Diagram

4. Design principle

The picture shown above illustrates the overall landscape of the class diagram, where we have implemented several design patterns and principles in our project, some of which we will discuss in detail. The SOLID design principles are applied throughout the entire project, encompassing the single responsibility principle, open-closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion principle.

4.1 Open-closed principle

The open-closed principle (OCP) states that software entities should be open for extension but closed for modification, a concept we fully embrace by allowing the behaviors of an entity to be extended without altering its source code.

This principle is clearly demonstrated in our selectionCommand module, where we implemented the command pattern. By encapsulating the method calls of addCommand and removeCommand into objects and managing them through the invoker, we ensure that new commands can be added without modifying the original ones, thus embodying the essence of the OCP.

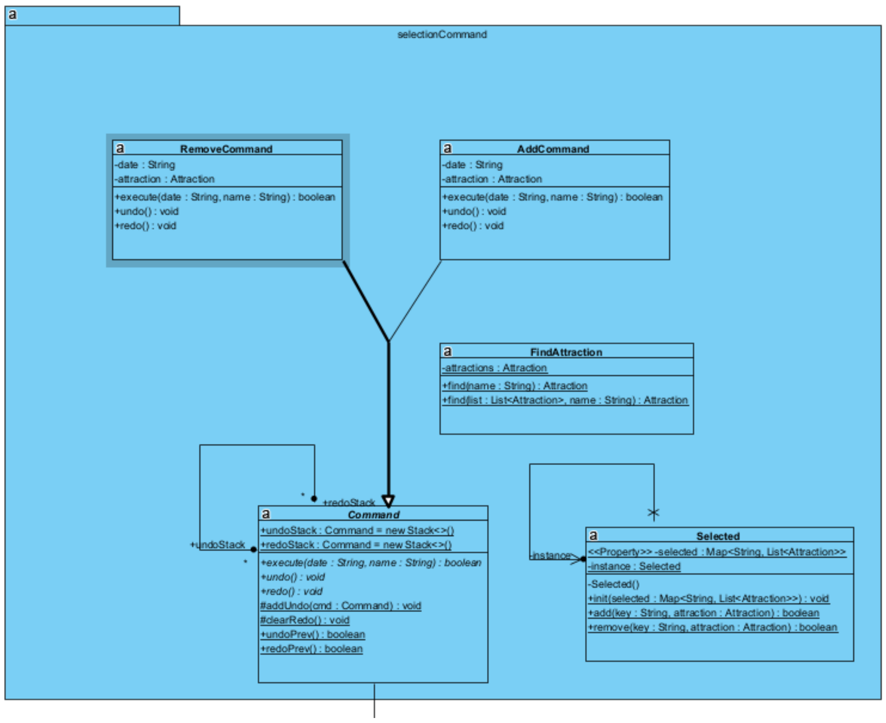

selectionCommand
RemoveCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
AddCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
FindAttraction
-attractions : Attraction
+find(name : String) : Attraction
+find(list : List<Attraction>, name : String) : Attraction
Command
+undoStack : Command = new Stack<>()
+redoStack : Command = new Stack<>()
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
#addUndo(cmd : Command) : void
#clearRedo() : void
+undoPrev() : boolean
+redoPrev() : boolean
+redoStack
+undoStack
*
*
Selected
<<Property>> -selected : Map<String, List<Attraction>>
-instance : Selected
-Selected()
+init(selected : Map<String, List<Attraction>>) : void
+add(key : String, attraction : Attraction) : boolean
+remove(key : String, attraction : Attraction) : boolean
-instance

Figure 4.1 Open-closed principle

4.2 Liskov substitution principle

The Liskov substitution principle (LSP) defines strong behavioral subtyping, stating that instances of a superclass and its subclasses must be interchangeable without disrupting the program. This principle ensures that subclasses extend the base class without altering its behavior and do not inherit non-existent features from the superclass during real interactions.

An example of this can be seen in our DataLoaderFactory class, which utilizes a factory method to create instances of DataLoader subclasses. The factory maintains a static mapping of singleton instances using a ConcurrentHashMap, allowing for the creation of subclass instances via reflection. This design adheres to the LSP, as any DataLoader subclass can be substituted without affecting the functionality of the factory or the broader application.

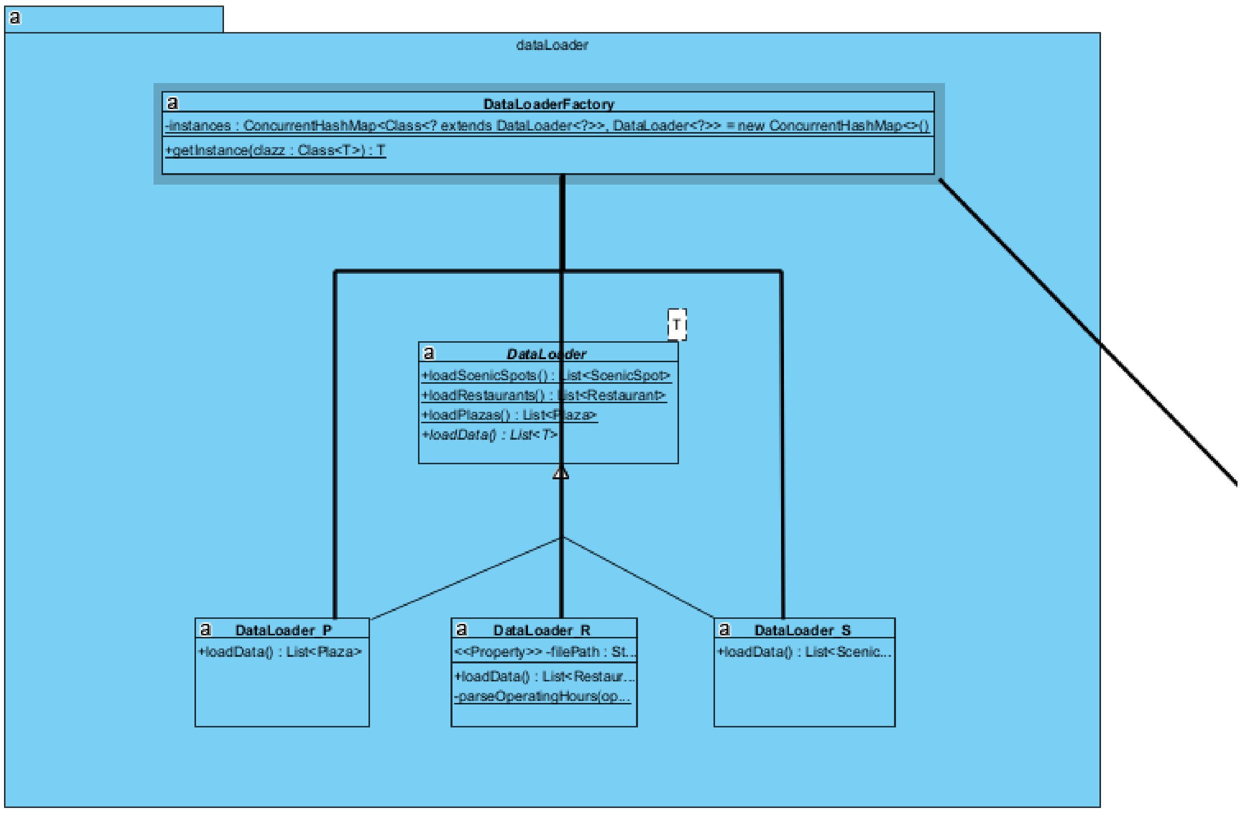

dataLoader
DataLoaderFactory
-instances : ConcurrentHashMap<Class<? extends DataLoader<?>>, DataLoader<?>> = new ConcurrentHashMap<>()
+getInstance(clazz : Class<T>) : T
T
DataLoader
+loadScenicSpots() : List<ScenicSpot>
+loadRestaurants() : List<Restaurant>
+loadPlazas() : List<Plaza>
+loadData() : List<T>
DataLoader_P
+loadData() : List<Plaza>
DataLoader_R
<<Property>> -filePath : St...
+loadData() : List<Restaur...
-parseOperatingHours(op...
DataLoader_S
+loadData() : List<Scenic...

Figure 4.2 Liskov substitution principle

4.3 Single Responsibility principle

The app entry point, represented by the Main class, demonstrates the facade pattern by providing a simplified interface for subsystems like TripPlanner and SelectionManager to interact seamlessly. This design not only simplifies debugging but also allows for future enhancements. Additionally, the interaction among these subsystems exemplifies the single responsibility principle (SRP), which states that every class, module, or function should have one distinct responsibility or purpose. As commonly defined, “every class should have only one reason to change.” In the Main class, we initialize the TripPlanner and SelectionManager, retrieve trip recommendations, and manage selections, showcasing how each subsystem focuses on its specific role without overlapping responsibilities.

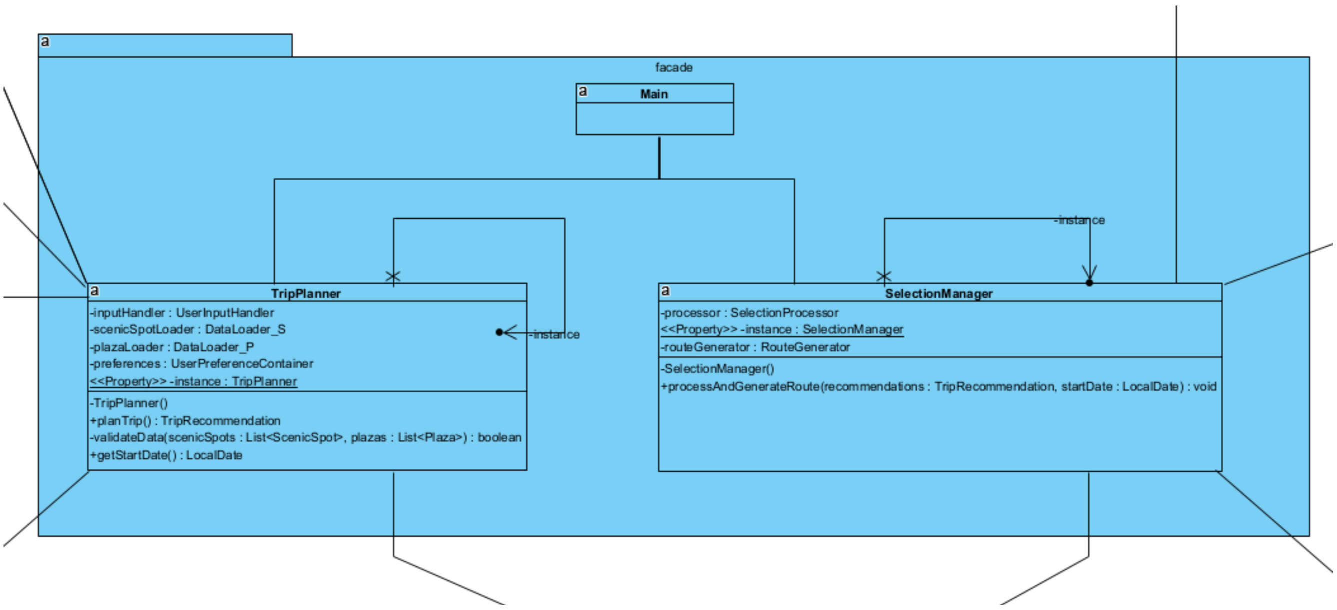

a
facade
a
Main
-instance
a
TripPlanner
-inputHandler : UserInputHandler
-scenicSpotLoader : DataLoader_S
-plazaLoader : DataLoader_P
-preferences : UserPreferenceContainer
<<Property>> -instance : TripPlanner
-TripPlanner()
+planTrip() : TripRecommendation
-validateData(scenicSpots : List<ScenicSpot>, plazas : List<Plaza>) : boolean
+getStartDate() : LocalDate
-instance
a
SelectionManager
-processor : SelectionProcessor
<<Property>> -instance : SelectionManager
-routeGenerator : RouteGenerator
-SelectionManager()
+processAndGenerateRoute(recommendations : TripRecommendation, startDate : LocalDate) : void

Figure 4.3 Single Responsibility principle

4.4 Dependency inversion principle

In object-oriented design, we emphasize two key principles: first, high-level modules should not depend on low-level modules; instead, both should rely on abstractions. Second, abstractions should not depend on details; rather, concrete implementations should depend on abstractions. These principles align with the Dependency Inversion Principle (DIP), which states that high-level modules should not depend on low-level modules but rather on abstractions.

An illustration of these principles can be seen in the UserPreferences_P class. This class extends the UserPreferences base class, focusing on user-specific preferences without directly depending on lower-level details. It encapsulates attributes like startDate, endDate, and lists for daily regions, tags, and filters, while providing methods to manipulate these preferences. By adhering to these principles, including DIP, the design promotes flexibility and maintainability. Changes in concrete implementations, such as adding new user preference types, can occur without affecting higher-level modules, fostering a more robust and adaptable architecture.

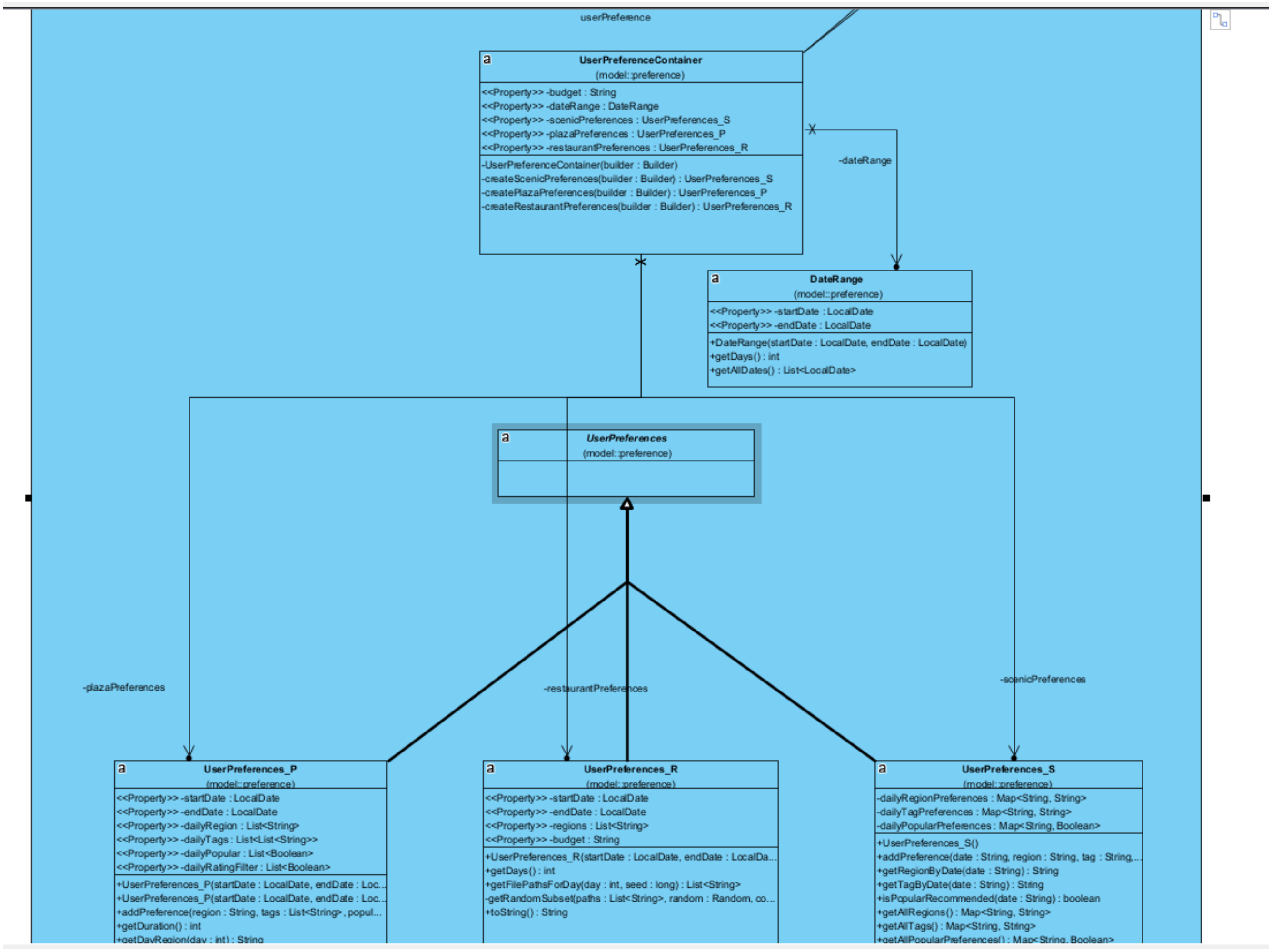

userPreference
UserPreferenceContainer
(model::preference)
<<Property>> -budget : String
<<Property>> -dateRange : DateRange
<<Property>> -scenicPreferences : UserPreferences_S
<<Property>> -plazaPreferences : UserPreferences_P
<<Property>> -restaurantPreferences : UserPreferences_R
-UserPreferenceContainer(builder : Builder)
-createScenicPreferences(builder : Builder) : UserPreferences_S
-createPlazaPreferences(builder : Builder) : UserPreferences_P
-createRestaurantPreferences(builder : Builder) : UserPreferences_R
-dateRange
DateRange
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
+DateRange(startDate : LocalDate, endDate : LocalDate)
+getDays() : int
+getAllDates() : List<LocalDate>
UserPreferences
(model::preference)
-plazaPreferences
-restaurantPreferences
-scenicPreferences
UserPreferences_P
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
<<Property>> -dailyRegion : List<String>
<<Property>> -dailyTags : List<List<String>>
<<Property>> -dailyPopular : List<Boolean>
<<Property>> -dailyRatingFilter : List<Boolean>
+UserPreferences_P(startDate : LocalDate, endDate : Loc...
+UserPreferences_P(startDate : LocalDate, endDate : Loc...
+addPreference(region : String, tags : List<String>, popul...
+getDuration() : int
+getDayRegion(day : int) : String
UserPreferences_R
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
<<Property>> -regions : List<String>
<<Property>> -budget : String
+UserPreferences_R(startDate : LocalDate, endDate : LocalDa...
+getDays() : int
+getFilePathsForDay(day : int, seed : long) : List<String>
-getRandomSubset(paths : List<String>, random : Random, co...
+toString() : String
UserPreferences_S
(model::preference)
-dailyRegionPreferences : Map<String, String>
-dailyTagPreferences : Map<String, String>
-dailyPopularPreferences : Map<String, Boolean>
+UserPreferences_S()
+addPreference(date : String, region : String, tag : String,...
+getRegionByDate(date : String) : String
+getTagByDate(date : String) : String
+isPopularRecommended(date : String) : boolean
+getAllRegions() : Map<String, String>
+getAllTags() : Map<String, String>
+getAllPopularPreferences() : Map<String, Boolean>

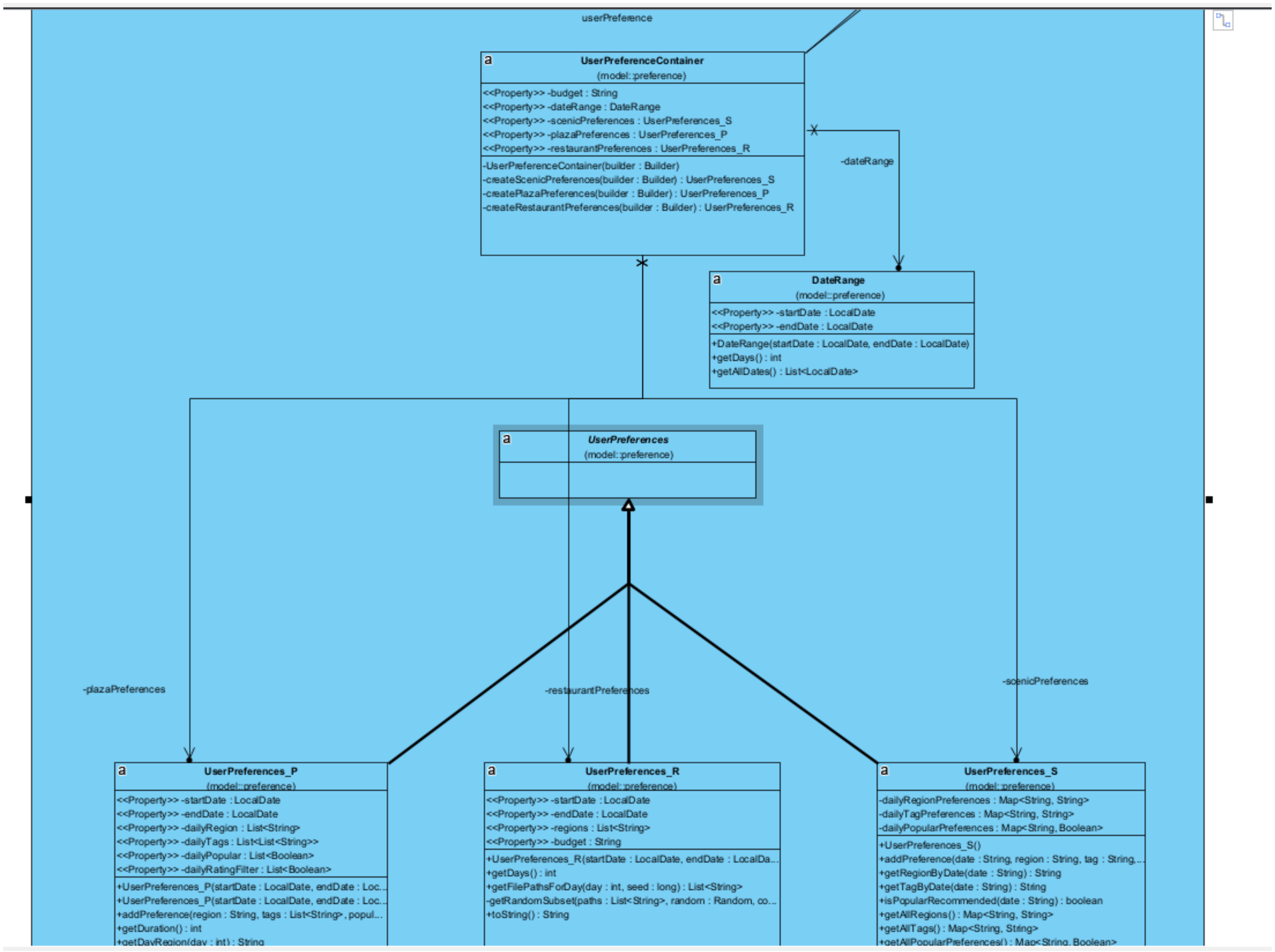
Figure 4.4 Dependency inversion principle

5. Design pattern

5.1 Builder pattern

The UserPreferenceContainer class implements the Builder pattern by using a nested Builder class to facilitate the construction of complex UserPreferenceContainer instances. The Builder provides methods for setting various properties, allowing for a fluent interface that supports method chaining. This approach encapsulates the logic required to create different user preference types (UserPreferences_S, UserPreferences_P, UserPreferences_R) while keeping the main class focused on its core responsibilities.

By using this pattern, the construction process becomes more flexible and readable, enabling users to create fully initialized objects with optional parameters without the need for multiple constructors.



userPreference
UserPreferenceContainer
(model::preference)
<<Property>> -budget : String
<<Property>> -dateRange : DateRange
<<Property>> -scenicPreferences : UserPreferences_S
<<Property>> -plazaPreferences : UserPreferences_P
<<Property>> -restaurantPreferences : UserPreferences_R
-UserPreferenceContainer(builder : Builder)
-createScenicPreferences(builder : Builder) : UserPreferences_S
-createPlazaPreferences(builder : Builder) : UserPreferences_P
-createRestaurantPreferences(builder : Builder) : UserPreferences_R
-dateRange
DateRange
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
+DateRange(startDate : LocalDate, endDate : LocalDate)
+getDays() : int
+getAllDates() : List<LocalDate>
UserPreferences
(model::preference)
-plazaPreferences
-restaurantPreferences
-scenicPreferences
UserPreferences_P
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
<<Property>> -dailyRegion : List<String>
<<Property>> -dailyTags : List<List<String>>
<<Property>> -dailyPopular : List<Boolean>
<<Property>> -dailyRatingFilter : List<Boolean>
+UserPreferences_P(startDate : LocalDate, endDate : Loc...
+UserPreferences_P(startDate : LocalDate, endDate : Loc...
+addPreference(region : String, tags : List<String>, popul...
+getDuration() : int
+getDayRegion(day : int) : String
UserPreferences_R
(model::preference)
<<Property>> -startDate : LocalDate
<<Property>> -endDate : LocalDate
<<Property>> -regions : List<String>
<<Property>> -budget : String
+UserPreferences_R(startDate : LocalDate, endDate : LocalDa...
+getDays() : int
+getFilePathsForDay(day : int, seed : long) : List<String>
-getRandomSubset(paths : List<String>, random : Random, co...
+toString() : String
UserPreferences_S
(model::preference)
-dailyRegionPreferences : Map<String, String>
-dailyTagPreferences : Map<String, String>
-dailyPopularPreferences : Map<String, Boolean>
+UserPreferences_S()
+addPreference(date : String, region : String, tag : String,...
+getRegionByDate(date : String) : String
+getTagByDate(date : String) : String
+isPopularRecommended(date : String) : boolean
+getAllRegions() : Map<String, String>
+getAllTags() : Map<String, String>
+getAllPopularPreferences() : Map<String, Boolean>

Figure 5.1 Builder Pattern

5.2 Command pattern

The command pattern is employed in our SelectionManager class to encapsulate method calls for adding and removing selections. By implementing this pattern, we create command objects—specifically AddCommand and RemoveCommand—that encapsulate the details of each operation.

This allows clients to issue commands without needing to understand the underlying logic. The SelectionManager acts as the invoker, managing the execution of these commands. This design not only simplifies the process of issuing different requests but also enables features like undo and redo functionality, as commands can be stored and referenced in a queue. Overall, the command pattern enhances flexibility and maintainability in handling user selections and interactions within the application.

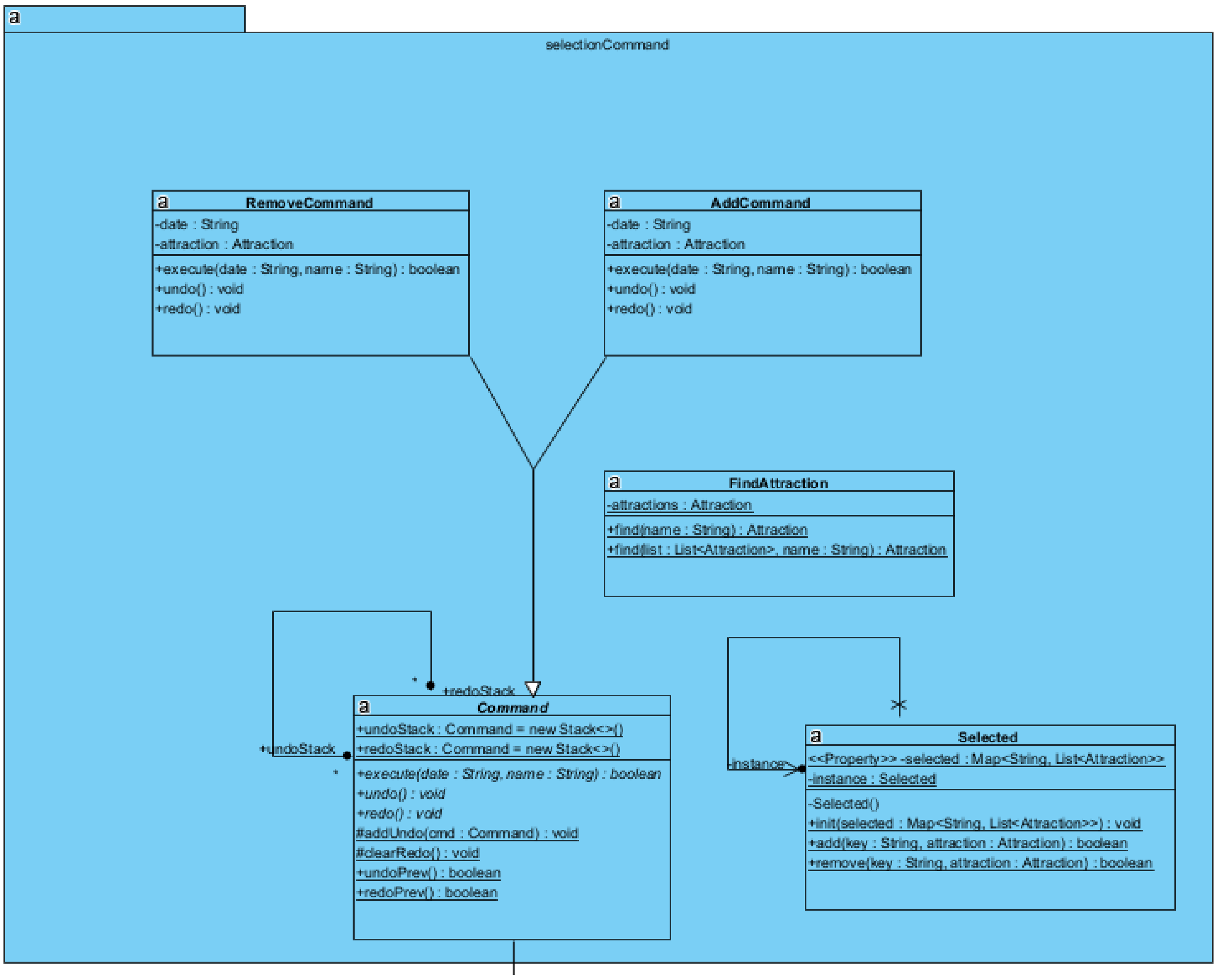
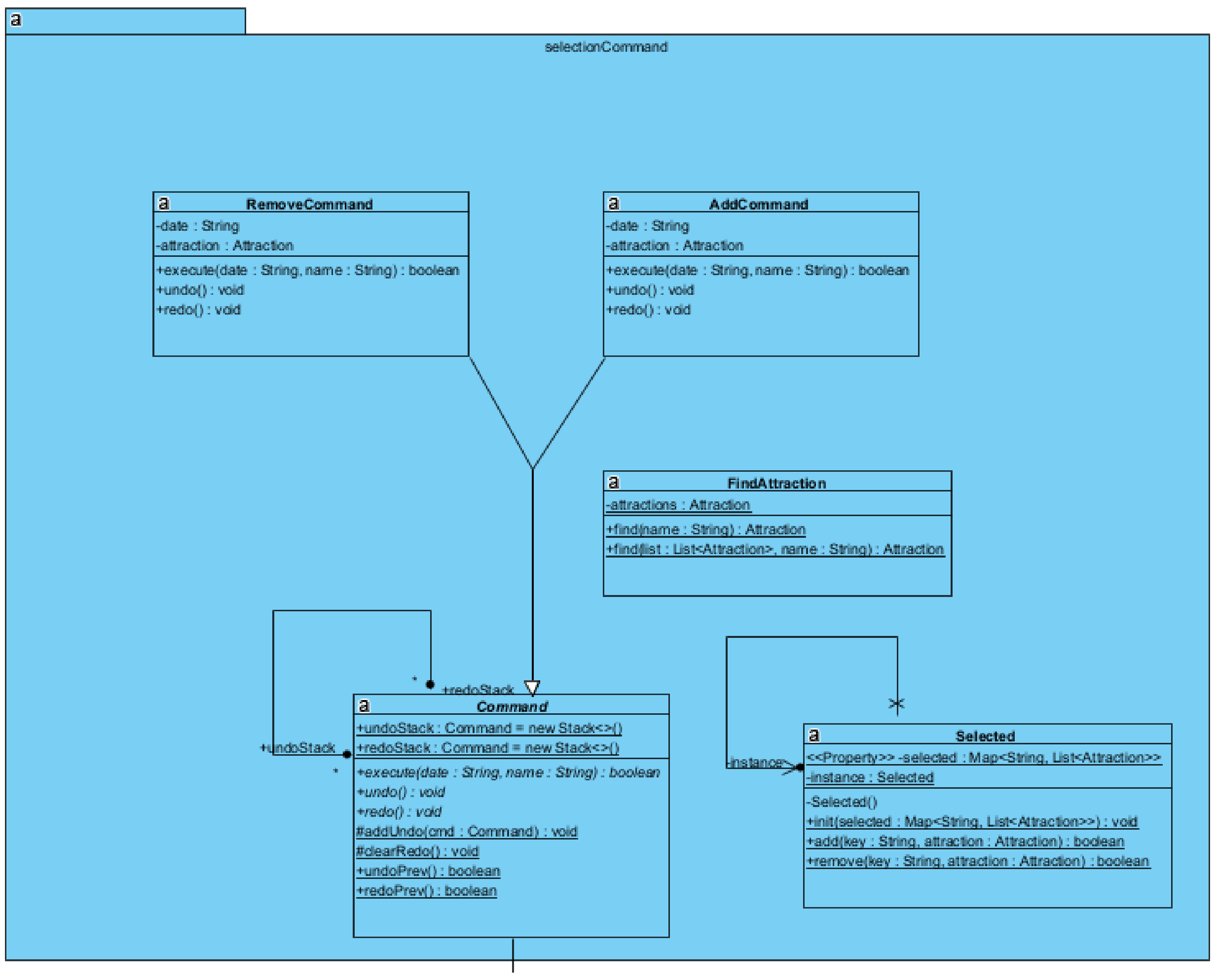

selectionCommand
RemoveCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
AddCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
FindAttraction
-attractions : Attraction
+find(name : String) : Attraction
+find(list : List<Attraction>, name : String) : Attraction
*
+redoStack
+undoStack
*
Command
+undoStack : Command = new Stack<>()
+redoStack : Command = new Stack<>()
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
#addUndo(cmd : Command) : void
#clearRedo() : void
+undoPrev() : boolean
+redoPrev() : boolean
-instance
Selected
<<Property>> -selected : Map<String, List<Attraction>>
-instance : Selected
-Selected()
+init(selected : Map<String, List<Attraction>>) : void
+add(key : String, attraction : Attraction) : boolean
+remove(key : String, attraction : Attraction) : boolean

Figure 5.2 Command Pattern

5.3 Singleton pattern

The Singleton pattern is implemented in the SelectionManager class to ensure that only one instance of this class exists throughout the application. This is crucial for managing user selections and maintaining a consistent state. The singleton instance is created lazily, meaning it is only instantiated when the getInstance method is called for the first time. This is achieved using double-checked locking to ensure thread safety in a multi-threaded environment.

By using the Singleton pattern, we reduce the overhead of creating multiple instances of SelectionManager, which could lead to duplicated efforts in managing selections and controlling application flow. This pattern is beneficial in various parts of the design, such as with ClientSearcher, RentableAllocator, and RecordManager, where a single point of access is essential for maintaining consistent data and behavior across the application. Overall, the Singleton pattern enhances resource efficiency, simplifies access management, and ensures that the state is shared correctly among different components.



selectionCommand
RemoveCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
AddCommand
-date : String
-attraction : Attraction
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
FindAttraction
-attractions : Attraction
+find(name : String) : Attraction
+find(list : List<Attraction>, name : String) : Attraction
Command
+undoStack : Command = new Stack<>()
+redoStack : Command = new Stack<>()
+execute(date : String, name : String) : boolean
+undo() : void
+redo() : void
#addUndo(cmd : Command) : void
#clearRedo() : void
+undoPrev() : boolean
+redoPrev() : boolean
+redoStack
+undoStack
*
*
Selected
<<Property>> -selected : Map<String, List<Attraction>>
-instance : Selected
-Selected()
+init(selected : Map<String, List<Attraction>>) : void
+add(key : String, attraction : Attraction) : boolean
+remove(key : String, attraction : Attraction) : boolean
-instance

Figure 5.3 Singleton pattern

5.4 Facade pattern

The Main class exemplifies the Facade pattern by providing a simplified interface for interacting with complex subsystems, specifically the TripPlanner and SelectionManager. This design allows users to initiate the trip planning process without needing to understand the intricate details of how these subsystems operate. By calling methods like planTrip() and initSelections(), the Main class acts as a front-facing interface that abstracts the complexity of the underlying code.

This approach not only streamlines the interaction with these subsystems, making the application easier to use and debug, but it also enhances maintainability. Future improvements or modifications to the underlying systems can be made without affecting the overall interface presented to the user. In essence, the Facade pattern simplifies the user experience and promotes a cleaner, more organized code structure.

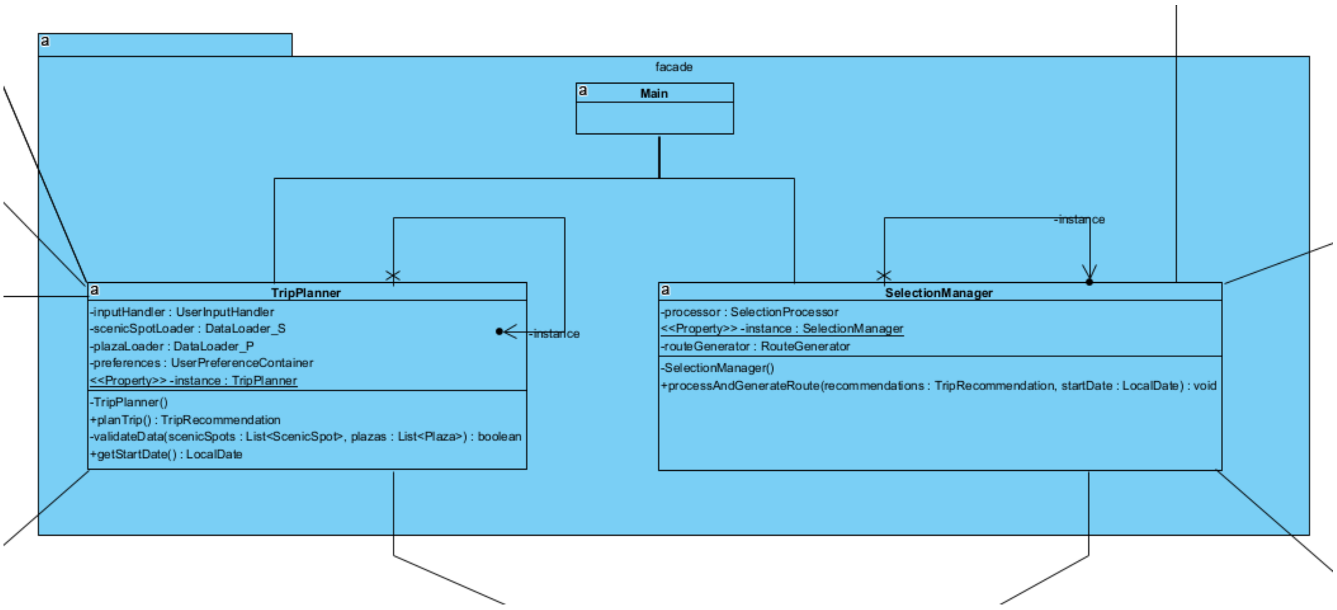

facade
Main
TripPlanner
-inputHandler : UserInputHandler
-scenicSpotLoader : DataLoader_S
-plazaLoader : DataLoader_P
-preferences : UserPreferenceContainer
<<Property>> -instance : TripPlanner
-TripPlanner()
+planTrip() : TripRecommendation
-validateData(scenicSpots : List<ScenicSpot>, plazas : List<Plaza>) : boolean
+getStartDate() : LocalDate
-instance
SelectionManager
-processor : SelectionProcessor
<<Property>> -instance : SelectionManager
-routeGenerator : RouteGenerator
-SelectionManager()
+processAndGenerateRoute(recommendations : TripRecommendation, startDate : LocalDate) : void
-instance

Figure 5.4 Façade pattern

6. Sequence Diagram

6.1 Collect Preference

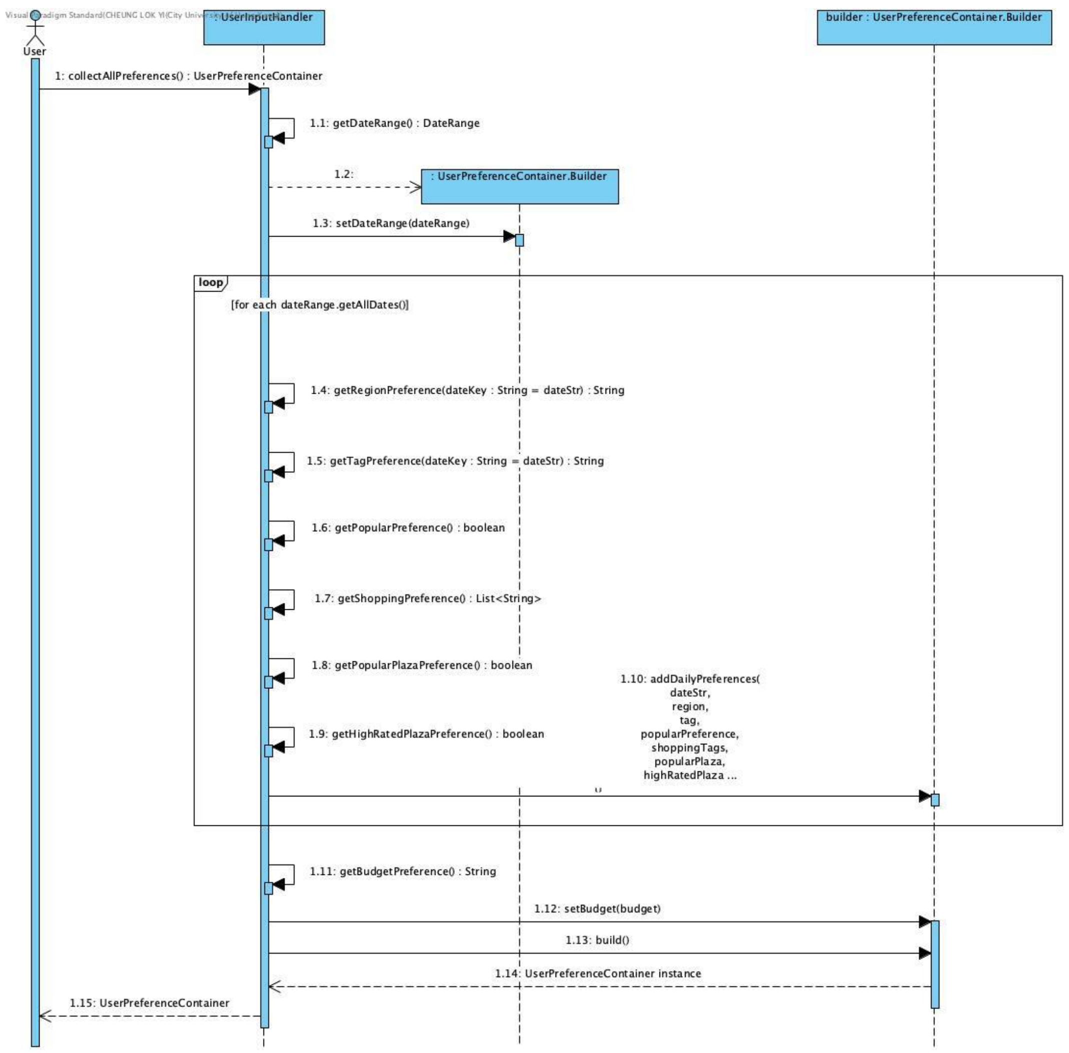
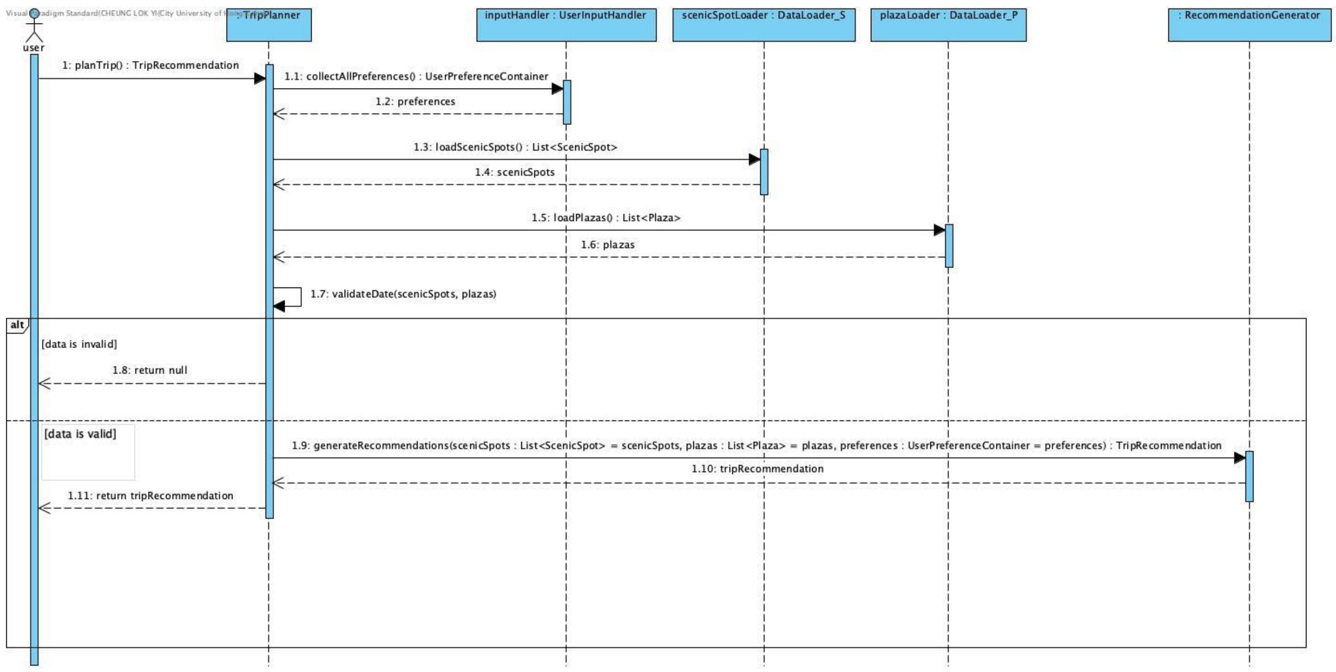

Visual Paradigm Standard(CHEUNG LOK YI(City University
User
: UserInputHandler
builder : UserPreferenceContainer.Builder
1: collectAllPreferences() : UserPreferenceContainer
1.1: getDateRange() : DateRange
1.2:
: UserPreferenceContainer.Builder
1.3: setDateRange(dateRange)
loop
[for each dateRange.getAllDates()]
1.4: getRegionPreference(dateKey : String = dateStr) : String
1.5: getTagPreference(dateKey : String = dateStr) : String
1.6: getPopularPreference() : boolean
1.7: getShoppingPreference() : List<String>
1.8: getPopularPlazaPreference() : boolean
1.9: getHighRatedPlazaPreference() : boolean
1.10: addDailyPreferences(dateStr, region, tag, popularPreference, shoppingTags, popularPlaza, highRatedPlaza ...
1.11: getBudgetPreference() : String
1.12: setBudget(budget)
1.13: build()
1.14: UserPreferenceContainer instance
1.15: UserPreferenceContainer

Figure 6.1 Collect Preference

The collectAllPreferences() method gathers all user preferences and generates a customized trip. The builder is used to build a container, it allows us to incrementally set user preferences in an organized way. The method loops through all the dates in the specific range and collects user preferences for each day. Then, these preferences are added to the builder through a series of method calls. Finally, the build() method will create a fully populated container object and return to the user.

6.2 Trip Planner

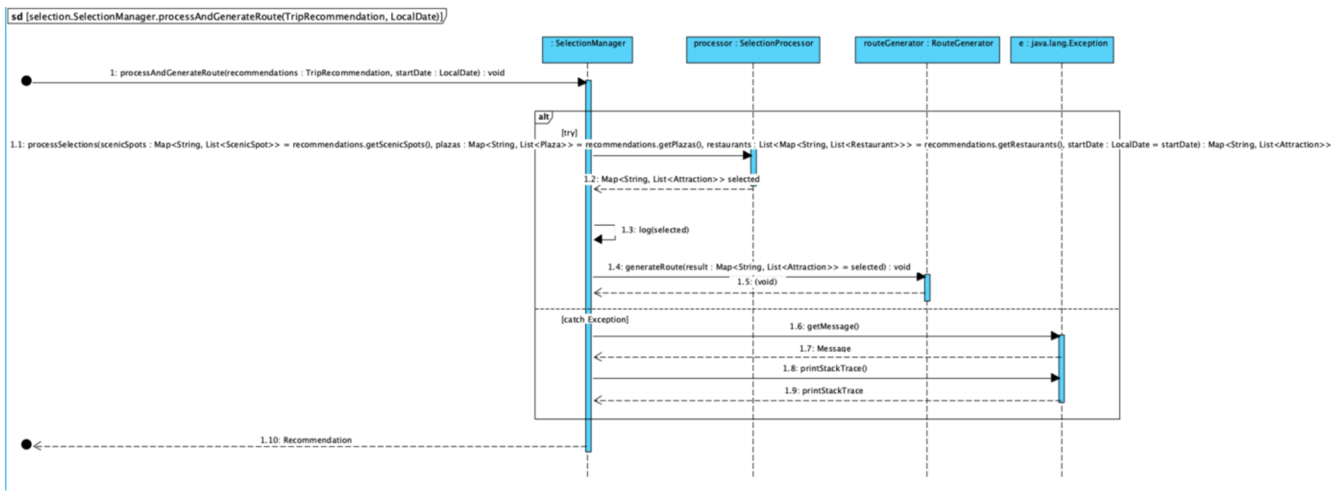


Visual Paradigm Standard(CHEUNG LOK YI(City University of Hong
user
: TripPlanner
inputHandler : UserInputHandler
scenicSpotLoader : DataLoader_S
plazaLoader : DataLoader_P
: RecommendationGenerator
1: planTrip() : TripRecommendation
1.1: collectAllPreferences() : UserPreferenceContainer
1.2: preferences
1.3: loadScenicSpots() : List<ScenicSpot>
1.4: scenicSpots
1.5: loadPlazas() : List<Plaza>
1.6: plazas
1.7: validateDate(scenicSpots, plazas)
alt
[data is invalid]
1.8: return null
[data is valid]
1.9: generateRecommendations(scenicSpots : List<ScenicSpot> = scenicSpots, plazas : List<Plaza> = plazas, preferences : UserPreferenceContainer = preferences) : TripRecommendation
1.10: tripRecommendation
1.11: return tripRecommendation

Figure 6.2 Trip Planner

The user will input the date range, preference districts, budget, ect. Then, the user triggers the planTrip() method on the controller class. The collectAllPreferences() method on InputHandler to retrieve preferences. The loadScenicSnΩpots() method on ScenicSpotLoader to load scenic spots. The loadPlazas() method on PlazaLoader to load plazas. The controller calls the validateData() method to ensure data integrity. The controller then calls the generateRecommendations() method on the singleton instance of RecommendationGenerator. Finally, the method returns a TripRecommendation object.

6.3 Route Generation



sd [selection.SelectionManager.processAndGenerateRoute(TripRecommendation, LocalDate)]
: SelectionManager
processor : SelectionProcessor
routeGenerator : RouteGenerator
e : java.lang.Exception
1: processAndGenerateRoute(recommendations : TripRecommendation, startDate : LocalDate) : void
alt
[try]
1.1: processSelections(scenicSpots : Map<String, List<ScenicSpot>> = recommendations.getScenicSpots(), plazas : Map<String, List<Plaza>> = recommendations.getPlazas(), restaurants : List<Map<String, List<Restaurant>>> = recommendations.getRestaurants(), startDate : LocalDate = startDate) : Map<String, List<Attraction>>
1.2: Map<String, List<Attraction>> selected
1.3: log(selected)
1.4: generateRoute(result : Map<String, List<Attraction>> = selected) : void
1.5: (void)
[catch Exception]
1.6: getMessage()
1.7: Message
1.8: printStackTrace()
1.9: printStackTrace
1.10: Recommendation

Figure 6.3 Route Generation

The processAndGenerateRoute() method is responsible for allowing users to select their preferred attractions and restaurants from a list of recommendations, and then generating an optimized route based on their selections. It ensures that the generated itinerary aligns with user preferences and scheduling. User triggers the processAndGenerateRoute() method on the controller class. The processSelections() method on Processor with scenic spots, plazas, restaurants, and the start date. It receives a Map of selected attractions and logs the selected attractions. The generateRoute() method on RouteGenerator to create the optimal route. If an exception occurs, the controller logs an error message and prints the stack trace.

6.4 Restaurant Recommendations

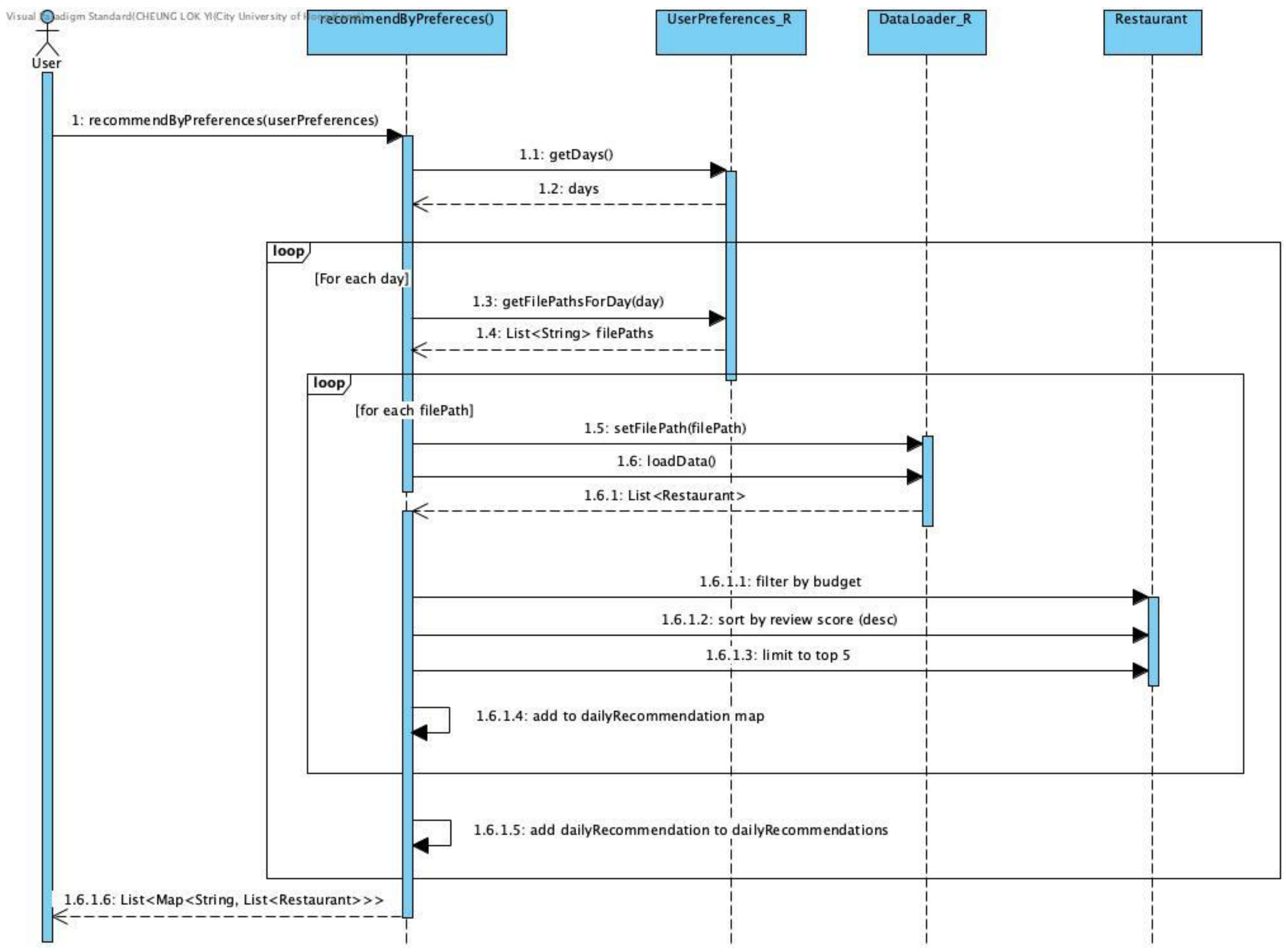
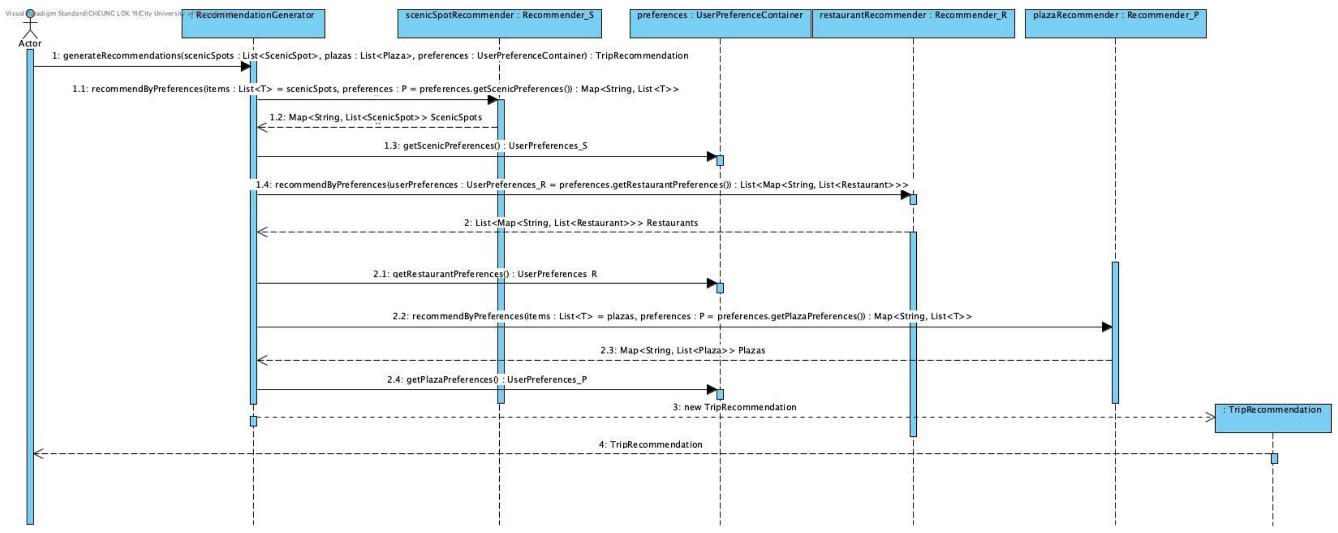

Visual Paradigm Standard(CHEUNG LOK YI(City University of
User
recommendByPreferences()
UserPreferences_R
DataLoader_R
Restaurant
1: recommendByPreferences(userPreferences)
1.1: getDays()
1.2: days
loop
[For each day]
1.3: getFilePathsForDay(day)
1.4: List<String> filePaths
loop
[for each filePath]
1.5: setFilePath(filePath)
1.6: loadData()
1.6.1: List<Restaurant>
1.6.1.1: filter by budget
1.6.1.2: sort by review score (desc)
1.6.1.3: limit to top 5
1.6.1.4: add to dailyRecommendation map
1.6.1.5: add dailyRecommendation to dailyRecommendations
1.6.1.6: List<Map<String, List<Restaurant>>>

Figure 6.4 Restaurant Recommendations

The recommendByPreferences() method provides restaurant recommendations that match the user's preferences and budget. For each day of the trip, it loads restaurant data from specified file paths, filters restaurants within the budget, and selects the top 5 based on review scores. The recommendations are organized into a list, with each day containing a map of file paths to the best restaurant options. This ensures users receive high-quality, budget-friendly dining suggestions for their trip. It is also similar design for plaza and scenic spots.

6.5 Recommendations Generation



Actor
: RecommendationGenerator
scenicSpotRecommender : Recommender_S
preferences : UserPreferenceContainer
restaurantRecommender : Recommender_R
plazaRecommender : Recommender_P
1: generateRecommendations(scenicSpots : List<ScenicSpot>, plazas : List<Plaza>, preferences : UserPreferenceContainer) : TripRecommendation
1.1: recommendByPreferences(items : List<T> = scenicSpots, preferences : P = preferences.getScenicPreferences()) : Map<String, List<T>>
1.2: Map<String, List<ScenicSpot>> ScenicSpots
1.3: getScenicPreferences() : UserPreferences_S
1.4: recommendByPreferences(userPreferences : UserPreferences_R = preferences.getRestaurantPreferences()) : List<Map<String, List<Restaurant>>>
2: List<Map<String, List<Restaurant>>> Restaurants
2.1: getRestaurantPreferences() : UserPreferences_R
2.2: recommendByPreferences(items : List<T> = plazas, preferences : P = preferences.getPlazaPreferences()) : Map<String, List<T>>
2.3: Map<String, List<Plaza>> Plazas
2.4: getPlazaPreferences() : UserPreferences_P
3: new TripRecommendation
: TripRecommendation
4: TripRecommendation

Figure 6.5 Recommendations Generation

The generateRecommendations() method begins with the user preferences as input. For each day, the method retrieves file paths for restaurant data and uses the DataLoader_R to load the data. Restaurants are then filtered based on budget and sorted by review scores. The top 5 restaurants for each file path are selected and compiled into daily recommendations. Finally, the method returns a list of these recommendations, organized by day. The diagram highlights the interaction between the method, DataLoader_R, and filtering logic to produce tailored results.