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

# Test 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 |



**Dec 4, 2024**

# Table of Content

# 1.Hierarchy Diagram

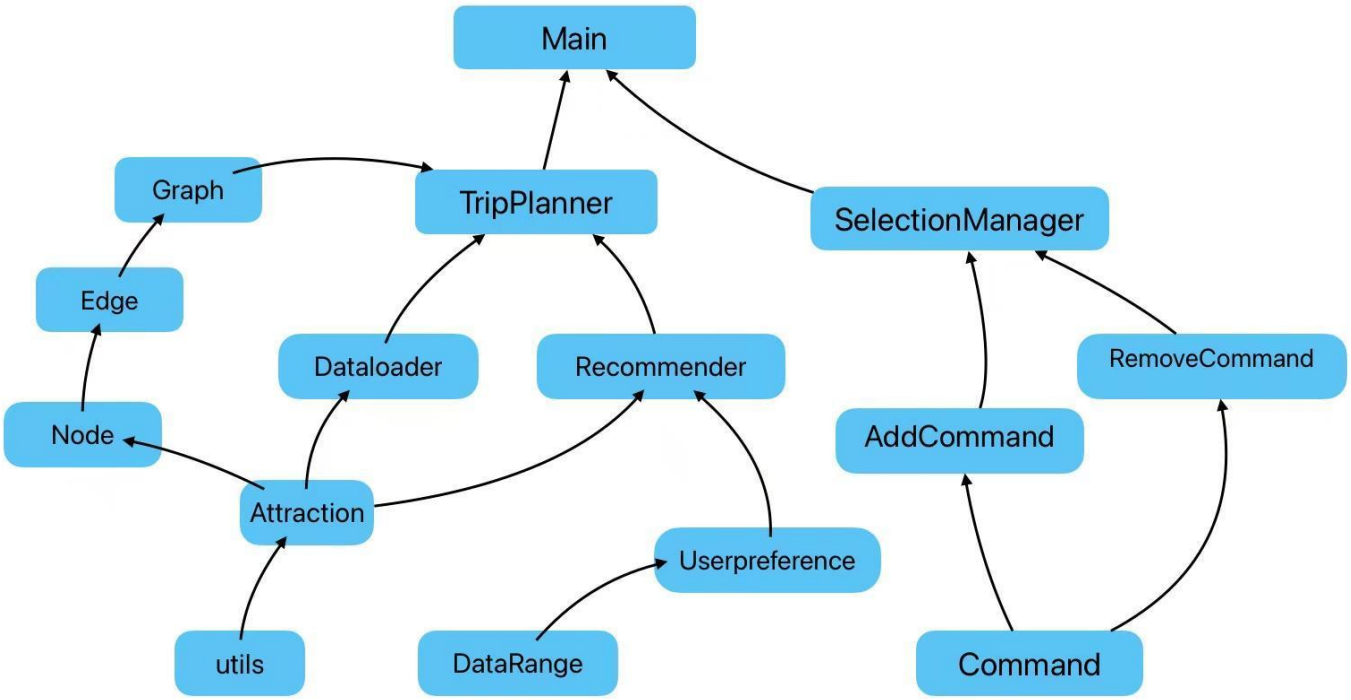



Figure 1. Overall hierarchy diagram

# 2. Testing Strategy

I Here's the revised text focusing on bottom-up testing strategy:
In this project, given our program's large and complex structure, we adopted a bottom-up testing strategy to effectively verify our system components.
For components below IO level, we wrote test cases using bottom-up methodology. This approach proved beneficial for several reasons:

Reduced need for test stubs and drivers despite frequent interdependencies
Enabled parallel testing across team members
Allowed early bug detection and fixes during initial coding stages
Facilitated independent testing of lower-level modules before integration

For components above IO level, we continued with the bottom-up approach despite their simpler structure. While testing these components required more coordination in terms of integration, it maintained consistency with our overall testing strategy and ensured thorough validation of all system interfaces.
This systematic bottom-up testing approach helped us maintain code quality while efficiently managing the testing timeline.

# 3.Testing Process

Based on the architecture diagram shown in the image, here's the revised testing process description:

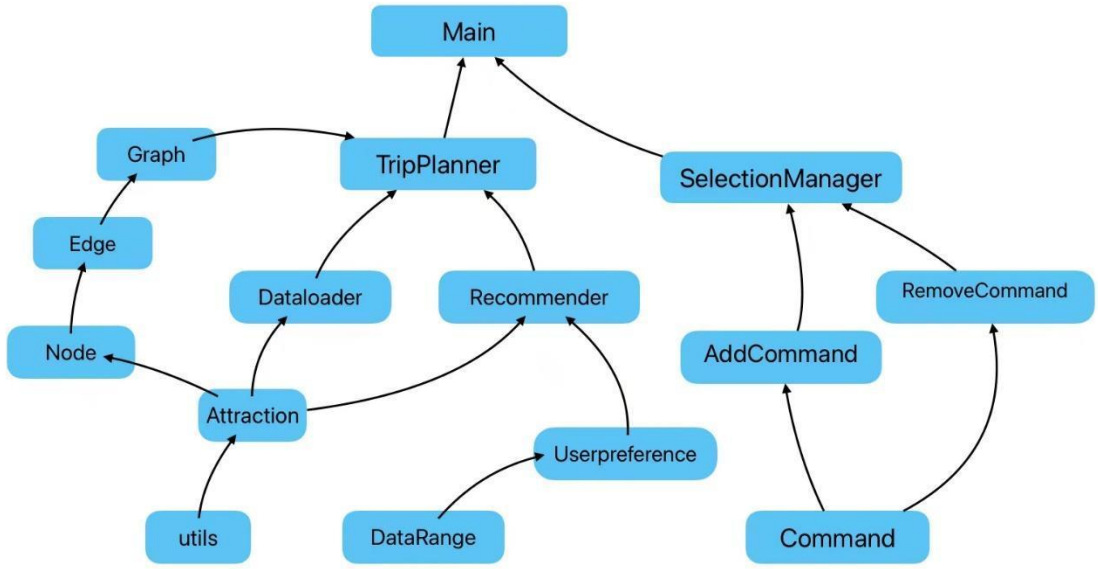



Figure 2 Hierarchy diagram

First, we conducted unit tests on core components:

- Node.java
- Edge.java
- Graph.java
- Attraction.java
- Utils.java

Integration testing proceeded as follows:

Graph Component Integration:

- Node + Edge + Graph

TripPlanner Integration Chain:

- Graph + TripPlanner
- Dataloader + TripPlanner
- Recommender + TripPlanner
- Userpreference + Recommender
- DataRange + Userpreference

Command Management Integration:

- Command + AddCommand + RemoveCommand + SelectionManager

Final Integration Phases:

- TripPlanner + Main
- SelectionManager + Main

System Test:

- Complete end-to-end testing of the entire application flow

This revision aligns with the component relationships and dependencies shown in the architectural diagram, focusing on the key components like TripPlanner, Graph, and SelectionManager, along with their associated subcomponents.

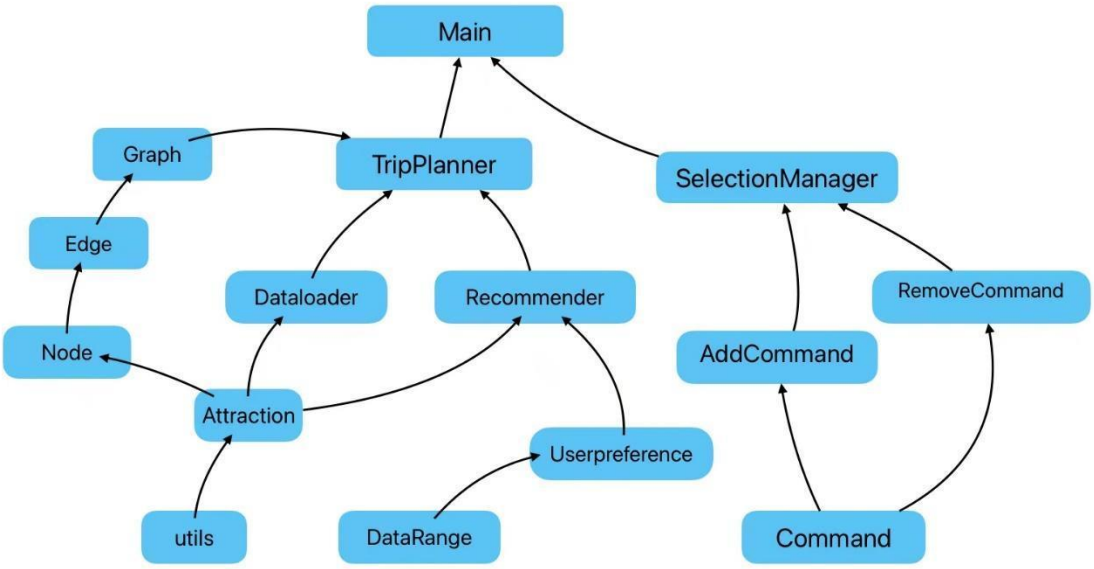
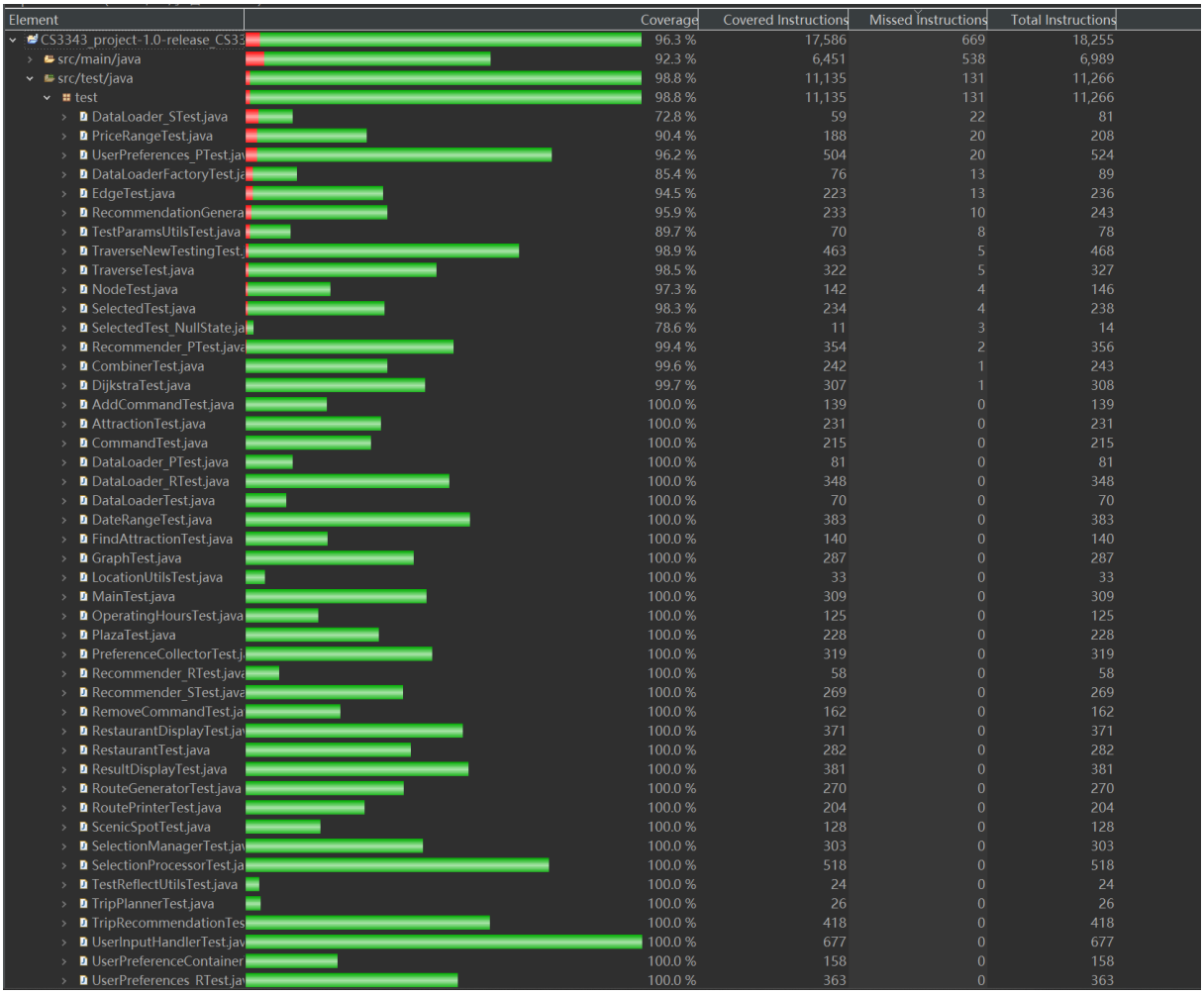



Figure 2 Database hierarchy diagram

# 4.Test Coverage Analysis

**Overall Coverage Status**

- Project Overall Coverage: 96.3% (17,586/18,255 instructions)
- Total Missing Coverage: 669 instructions



| Element | Coverage | Covered Instructions | Missed Instructions | Total Instructions |
|---|---|---|---|---|
| CS3343_project-1.0-release_CS33 | 96.3 % | 17,586 | 669 | 18,255 |
| src/main/java | 92.3 % | 6,451 | 538 | 6,989 |
| src/test/java | 98.8 % | 11,135 | 131 | 11,266 |
| test | 98.8 % | 11,135 | 131 | 11,266 |
| DataLoader_STest.java | 72.8 % | 59 | 22 | 81 |
| PriceRangeTest.java | 90.4 % | 188 | 20 | 208 |
| UserPreferences_PTest.jav | 96.2 % | 504 | 20 | 524 |
| DataLoaderFactoryTest.ja | 85.4 % | 76 | 13 | 89 |
| EdgeTest.java | 94.5 % | 223 | 13 | 236 |
| RecommendationGenera | 95.9 % | 233 | 10 | 243 |
| TestParamsUtilsTest.java | 89.7 % | 70 | 8 | 78 |
| TraverseNewTestingTest_ | 98.9 % | 463 | 5 | 468 |
| TraverseTest.java | 98.5 % | 322 | 5 | 327 |
| NodeTest.java | 97.3 % | 142 | 4 | 146 |
| SelectedTest.java | 98.3 % | 234 | 4 | 238 |
| SelectedTest_NullState.ja | 78.6 % | 11 | 3 | 14 |
| Recommender_PTest.java | 99.4 % | 354 | 2 | 356 |
| CombinerTest.java | 99.6 % | 242 | 1 | 243 |
| DijkstraTest.java | 99.7 % | 307 | 1 | 308 |
| AddCommandTest.java | 100.0 % | 139 | 0 | 139 |
| AttractionTest.java | 100.0 % | 231 | 0 | 231 |
| CommandTest.java | 100.0 % | 215 | 0 | 215 |
| DataLoader_PTest.java | 100.0 % | 81 | 0 | 81 |
| DataLoader_RTest.java | 100.0 % | 348 | 0 | 348 |
| DataLoaderTest.java | 100.0 % | 70 | 0 | 70 |
| DateRangeTest.java | 100.0 % | 383 | 0 | 383 |
| FindAttractionTest.java | 100.0 % | 140 | 0 | 140 |
| GraphTest.java | 100.0 % | 287 | 0 | 287 |
| LocationUtilsTest.java | 100.0 % | 33 | 0 | 33 |
| MainTest.java | 100.0 % | 309 | 0 | 309 |
| OperatingHoursTest.java | 100.0 % | 125 | 0 | 125 |
| PlazaTest.java | 100.0 % | 228 | 0 | 228 |
| PreferenceCollectorTest.j | 100.0 % | 319 | 0 | 319 |
| Recommender_RTest.java | 100.0 % | 58 | 0 | 58 |
| Recommender_STest.java | 100.0 % | 269 | 0 | 269 |
| RemoveCommandTest.ja | 100.0 % | 162 | 0 | 162 |
| RestaurantDisplayTest.jav | 100.0 % | 371 | 0 | 371 |
| RestaurantTest.java | 100.0 % | 282 | 0 | 282 |
| ResultDisplayTest.java | 100.0 % | 381 | 0 | 381 |
| RouteGeneratorTest.java | 100.0 % | 270 | 0 | 270 |
| RoutePrinterTest.java | 100.0 % | 204 | 0 | 204 |
| ScenicSpotTest.java | 100.0 % | 128 | 0 | 128 |
| SelectionManagerTest.jav | 100.0 % | 303 | 0 | 303 |
| SelectionProcessorTest.ja | 100.0 % | 518 | 0 | 518 |
| TestReflectUtilsTest.java | 100.0 % | 24 | 0 | 24 |
| TripPlannerTest.java | 100.0 % | 26 | 0 | 26 |
| TripRecommendationTes | 100.0 % | 418 | 0 | 418 |
| UserInputHandlerTest.jav | 100.0 % | 677 | 0 | 677 |
| UserPreferenceContainer | 100.0 % | 158 | 0 | 158 |
| UserPreferences_RTest.jav | 100.0 % | 363 | 0 | 363 |

**Module-Level Analysis**

High Coverage Modules (>95%)

- src/test/java: 98.8% (11,135/11,266 instructions)
- Multiple test files with 100% coverage, including:
  - AttractionTest.java
  - CommandTest.java
  - DateRangeTest.java
  - RestaurantTest.java
  - ResultDisplayTest.java
  - UserInputHandlerTest.java

Notable Coverage Areas:

- TripRecommendationTest: 100% (418/418)
- SelectionProcessorTest: 100% (518/518)
- UserInputHandlerTest: 100% (677/677)
- RestaurantDisplayTest: 100% (371/371)

Lower Coverage Explanation
DataLoader_STest.java (72.8%):

- Contains complex data loading operations
- Includes error handling for file I/O operations
- Features edge cases for malformed data handling

SelectedTest_NullState.java (78.6%):

- Handles null state scenarios and edge cases
- Contains defensive programming logic
- Includes complex state validation checks

DataLoaderFactoryTest.java (85.4%):

- Involves factory pattern implementation testing
- Contains dependency injection scenarios
- Features multiple configuration combinations

The main source code (src/main/java) shows slightly lower coverage at 92.3% (6,451/6,989):

- Contains core business logic implementation
- Includes complex error handling routines
- Features extensive integration points with external systems

# 5. Code Refactoring

In the code refactoring section, three main methods are used: Rename and Merge Duplicate Code, Extract Method, and Replace Conditional with Polymorphism, while the refactoring is mainly for Attraction including three subclasses: Plaza, ScenicSpot, and Restaurant, and also for three dataloaders and three recommenders.Next, the three methods are explained one by one with examples.

**Rename and Merge Duplicate Code**

This method is to rename variables, methods, classes, etc. to make their names more consistent with their actual meaning and to enhance the self-interpretability of the code. Additionally, to extract and consolidate duplicate code segments to reduce code redundancy. Following is the example, we begin with three distinct sorts of attractions, which are independent yet share certain common qualities. We constructed a base class to encapsulate shared properties, while the subclasses implement their specific functionalities.

**Before:**

```java
import java.util.List;

public class ScenicSpot {
    private String name;
    private String namezh;
    private int reviewCount;
    private String location;
    private String metroStation;
    private String region;
    private List<String> feature;

    // Getters and Setters
```

```java
import java.util.List;
import utils.LocationUtils;

public class Plaza {
    private String name;
    private String nameZh;
    private String location;
    private String metroStation;
    private int reviewCount;
    private double reviewScore;
    private List<String> feature;
    private OpeningHours openingHours;

    public Plaza() {}
    //Getter & Setter
```

```java
import java.time.LocalTime;
import java.util.List;
import utils.PriceRange;
import utils.OperatingHours;

public class Restaurant {
    private String name;
    private String nameZh;
    private String location;
    private String metroStation;
    private int reviewCount;
    private int recommendedTime;
    private PriceRange priceRange;
    private double reviewScore;
    private List<OperatingHours> openTime;
    private PriceRange avgExpense;

//Getter&Setter
```

Figure 3 Sample code before refactoring

**After:**

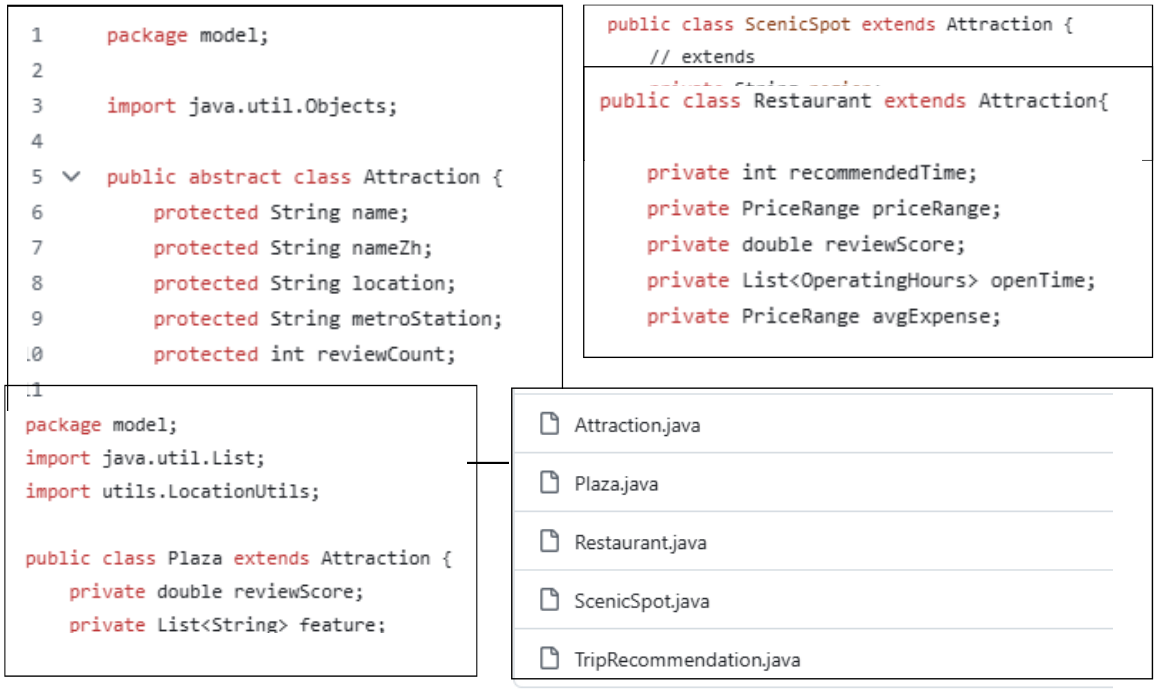



Figure 4. Sample code after refactoring and the catalogs

## Extract Method

The second method is Extracting methods, which is used to encapsulate certain functions that can be separated to improve code readability and reusability. In this example, innitially, We consolidated the three data-reading functions into a single class and employed criteria to choose which function to utilize. Subsequent to refactoring, we encapsulate the three methods for reading distinct files into three related subclasses.

**Before:**

```
public class DataLoader {
    public List<ScenicSpot> loadScenicSpots(String filePath) {
        return loadData(filePath, type:"ScenicSpot");
    }
    public List<Restaurant> loadRestaurants(String filePath) {
        return loadData(filePath, type:"Restaurant");
    }
    public List<Plaza> loadPlazas(String filePath) {
        return loadData(filePath, type:"Plaza");
    }
    private <T> List<T> loadData(String filePath, String type) {
        List<T> data = new ArrayList<>();
        ObjectMapper objectMapper = new ObjectMapper();

        try {
            JsonNode root = objectMapper.readTree(new File(filePath));
            for (JsonNode node : root) {
                try {
                    if ("ScenicSpot".equals(type)) {
                        data.add((T) parseScenicSpot(node));
                    } else if ("Restaurant".equals(type)) {
                        data.add((T) parseRestaurant(node));
                    } else if ("Plaza".equals(type)) {
                        data.add((T) parsePlaza(node));
                    }
                } catch (Exception e) {
                    System.err.println("Error processing data: " + e.getMessage());
                    continue;
                }
            }
        } catch (IOException e) {
            System.err.println("Error reading file: " + e.getMessage());
        }
        return data;
```

```
private ScenicSpot parseScenicSpot(JsonNode node) {
    String name = node.get("name").asText();
    String location = node.get("location").asText();
    String description = node.get("description").asText();
    return new ScenicSpot(name, location, description);
}
```

Figure 5. Sample code before refactoring

**After:**

```
public class DataLoader_S extends DataLoader<ScenicSpot> {
    private String filePath = "data\\scenicspots.json";

    public List<ScenicSpot> loadData() {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            // 读取JSON文件并转换成List
            return objectMapper.readValue(new File(filePath),
                    objectMapper.getTypeFactory().constructCollectionType(List.class, ScenicSpot.class));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }

    public void setFilePath(String filePath) {
        this.filePath = filePath;
    }
}
```

```
public List<Restaurant> loadData() {

    List<Restaurant> attractions = new ArrayList<>();
    ObjectMapper objectMapper = new ObjectMapper();
```

```java
public class DataLoader_S extends DataLoader<ScenicSpot> {
    private String filePath = "data\\scenicspots.json";

    public List<ScenicSpot> loadData() {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            // 读取JSON文件并转换成List
            return objectMapper.readValue(new File(filePath),
                    objectMapper.getTypeFactory().constructCollectionType(List.class, ScenicSpot.class));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

Figure 6. Sample code after refactoring

**Replace Condition with Polymorphism**

The third method is replacing conditional with polymorphism, which is used to replace some complex conditional expressions with polymorphism. For example, there are three functions are encapsulated in three sub-classes, but the names of the functions that load the data are different. So an base-class is created with an abstract function, and implement the function in different ways in sub-classes.Consequently, while subsequently loading data, you just need to utilize the same name and provide different class arguments to invoke several loaddata procedures.

**Before:**

```
public List<ScenicSpot> loadData(String filePath) {
    public List<ScenicSpot> loadData() {
        ObjectMapper objectMapper = new ObjectMapper();
        String filePath = "E:\\github\\CS3343_project\\data\\scenicspots.json";
        try {
            // 读取JSON文件并转换成List
            return objectMapper.readValue(new File(filePath),

    }
}
}
```

```
public static List<Plaza> loadPlaza(String filePath) {
    ObjectMapper objectMapper = new ObjectMapper();
    try {
        return objectMapper.readValue(new File(filePath),
                objectMapper.getTypeFactory().constructCollectionType(List.class, ScenicSpot.class));
    } catch (IOException e) {
        e.printStackTrace();
    }
    return null;
}
```

```
public List<Restaurant> loadRestuarant(String filePath) {
    List<Restaurant> attractions = new ArrayList<>();
    try {
        int reviewCount = node.has("reviewCount") ? node.get("reviewCount").asInt() : 0; // 获取评论数量
        double reviewScore = node.has("reviewScore") ? node.get("reviewScore").asDouble() : 0.0; // 获取评分
        List<OperatingHours> openTime = parseOperatingHours(node.get("openTime").asText()); // 获取开放时间
        attractions.add(new Restaurant(name, nameZh, location, metroStation, recommendedTime, avgExpense, reviewCount, reviewScore, openTime));
        attractions.add(new Restaurant(name, nameZh, location, metroStation, recommendedTime, avgExpense,
                reviewCount, reviewScore, openTime));
    } catch (Exception e) {
        // 如果处理单个景点数据出错，跳过该景点继续处理下一个
        System.err.println("Error processing attraction: " + e.getMessage());
    }
    return attractions;
}
```

**After:**

```
public abstract class DataLoader<T> {

    // 公共方法：加载 ScenicSpot
    public static List<ScenicSpot> loadScenicSpots() {
        DataLoader<ScenicSpot> loader = DataLoaderFactory.getInstance(DataLoader_S.class);
        return loader.loadData();
    }

    // 公共方法：加载 Restaurant
    public static List<Restaurant> loadRestaurants() {
        DataLoader<Restaurant> loader = DataLoaderFactory.getInstance(DataLoader_R.class);
        return loader.loadData();
    }

    // 公共方法：加载 Plaza
    public static List<Plaza> loadPlazas() {
        DataLoader<Plaza> loader = DataLoaderFactory.getInstance(DataLoader_P.class);
        return loader.loadData();
    }

    // 抽象方法：子类实现具体加载逻辑
    public abstract List<T> loadData();
}
```

```java
public class DataLoader_S extends DataLoader<ScenicSpot> {
    private String filePath = "data\\scenicspots.json";

    public List<ScenicSpot> loadData() {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            // 读取JSON文件并转换成List
            return objectMapper.readValue(new File(filePath),
                    objectMapper.getTypeFactory().constructCollectionType(List.class, ScenicSpot.class));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

```java
public List<Restaurant> loadData() {

    List<Restaurant> attractions = new ArrayList<>();
    ObjectMapper objectMapper = new ObjectMapper();

    try {
        JsonNode root = objectMapper.readTree(new File(filePath));
        for (JsonNode node : root) {
            try {
                String name = node.get("name").asText(); // 获取景点名称
                String nameZh = node.get("nameZh").asText(); // 获取景点的中文名称
                String location = node.get("location").asText(); // 获取景点的位置
                String metroStation = node.get("metroStation").asText(); // 获取地铁站信息
                int recommendedTime = node.has("recommendedTime") ? node.get("recommendedTime").asInt() : 0; // 获取推荐游玩时间
                String avgExpense = node.get("avgExpense").asText(); // 直接获取原始价格字符串
                int reviewCount = node.has("reviewCount") ? node.get("reviewCount").asInt() : 0; // 获取评论数量
                double reviewScore = node.has("reviewScore") ? node.get("reviewScore").asDouble() : 0.0; // 获取评分
                List<OperatingHours> openTime = parseOperatingHours(node.get("openTime").asText()); // 获取开放时间
                attractions.add(new Restaurant(name, nameZh, location, metroStation, recommendedTime, avgExpense,
                        reviewCount, reviewScore, openTime));
            } catch (Exception e) {
                // 如果处理单个景点数据出错，跳过该景点继续处理下一个
                System.err.println("Error processing attraction: " + e.getMessage());
                continue;
            }
        }
    } catch (IOException e) {
        System.err.println("Error reading file: " + e.getMessage());
    }

    return attractions;
}
```

```java
public class DataLoader_P extends DataLoader<Plaza> {
    private String filePath = "data\\shopping.json";
    public List<Plaza> loadData() {
        ObjectMapper objectMapper = new ObjectMapper();
        try {
            return objectMapper.readValue(new File(filePath),
                    objectMapper.getTypeFactory().constructCollectionType(List.class, Plaza.class));
        } catch (IOException e) {
            e.printStackTrace();
        }
        return null;
    }
```

# 6. Test Cases

## AddCommandTest.java

| **Purpose:** To test whether the basic methods inside AddCommand.java can function well for adding attractions to selected dates | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testAddSuccess** | Test AddCommand.execute() with valid date "2024-01-01" and valid attraction name | Selected list for "2024-01-01" contains 3 attractions after adding | As expected |
| T002 | **testAddFail1** | Test ClientStaff.getMaxBorrowedCount() | False | As expected |
| T003 | **testAddFail2** | Test ClientStaff.getEmailt() | False | As expected |

# TestAttraction.java

| **Purpose:** To test whether the basic methods inside Attraction.java including getters, setters and equals comparisons can function well | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **test_01** | Test all getter and setter methods with sample attraction data | true | As expected |
| T002 | **test_02** | Test equals() with identical and different attraction objects | false | As expected |
| T003 | **test_03** | Test equals() with null object | false | As expected |
| T004 | **test_04** | Test equals() with object of different class | Same attraction returns true, different attraction returns false | As expected |
| T005 | **test_05** | Test equals() with attractions having only one different field (metro station, location, Chinese name, English name) | false for all partial matches | As expected |

# TestCombiner.java

| **Purpose:** To test whether the recommendation combining functionality in Combiner.java works correctly | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **test_01** | Test combining recommendations with valid data for scenic spots, restaurants and plazas | Combined list contains 5 items (2 restaurants + 1 spot + 1 plaza) with correct object types | As expected |
| T002 | **test_02** | Test combining with empty maps and lists | Empty but non-null result map | As expected |
| T003 | **test_03** | Test combining with null restaurant recommendations | Result contains at least 2 items (spot and plaza), other recommendations remain normal | As expected |

# TestCommand.java

**Purpose:** To test the command execution, undo, and redo functionality in the selection system

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | test_01 | Test adding attraction with AddCommand | Added successfully, list size becomes 3 | As expected |
| T002 | test_02 | Test undoing previous add operation | Undo successful, list size returns to 2 | As expected |
| T003 | test_03 | Test undoing when no operations left | Returns false | As expected |
| T004 | test_04 | Test redoing previously undone add | Redo successful, list size back to 3 | As expected |
| T005 | test_05 | Test redoing when no redos available | Returns false | As expected |
| T006 | test_06 | Test removing attraction with RemoveCommand | Removed successfully, list size becomes 2 | As expected |
| T007 | test_07 | Test redo after new command execution | Returns false (redo stack cleared) | As expected |

| | | | | |
|---|---|---|---|---|
| T008 | **test_08** | Test undoing remove command | Undo successful, list size back to 3 | As expected |
| T009 | **test_08** | Test redoing remove command | Redo successful, list size becomes 2 | As expected |

# TestDataLoaderFactory.java

| **Purpose:** To test the singleton pattern implementation and instance creation in DataLoaderFactory | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | test_01 | Get DataLoader_S instance twice | Both references point to same non-null instance | As expected |
| T002 | test_02 | Get DataLoader_P instance twice | Both references point to same non-null instance | As expected |
| T003 | test_03 | Get instances of DataLoader_S and DataLoader_P | Different non-null instances returned) | As expected |
| T004 | test_04 | Try to get instance of invalid DataLoader | RuntimeException with appropriate error message | As expected |

| | | | | |
|---|---|---|---|---|
| T005 | **test_04** | Create new DataLoaderFactory instance | Non-null factory instance | As expected |

`

# TestDataLoader.java

| **Purpose**: To test data loading functionality for scenic spots and plazas | | | | |
|---|---|---|---|---|
| Test case ID | **Test Name** | Input Description | Expected Result | Actual Output |
| T001 | **test_01** | Load scenic spots data | Non-null list with valid spot names and locations | As expected |
| T002 | **test_02** | Load plaza data | Non-null list with valid plaza names, locations, and operating hours | As expected |
| T003 | **test_03** | r Test exception handling for data loading | No exceptions thrown when loading either type of data | As expected |

# TestDataLoader_P.java

| **Purpose:** To test the plaza data loading functionality including JSON parsing and file handling l | | | | |
|---|---|---|---|---|
| Test case ID | **Test Name** | Input Description | Expected Result | Actual Output |
| T001 | **test_01** | Load valid JSON data with plaza information from temporary file | Successfully parsed plaza object with correct field values | As expected |
| T002 | **test_02** | Attempt to load data from non-existent file | Returns null | As expected |

# TestDataLoader_R.java

| **Purpose:** To test the restaurant data loading functionality including JSON parsing, field validation, and error handlin | | | | |
|---|---|---|---|---|
| Test case ID | **Test Name** | Input Description | Expected Result | Actual Output |
| T001 | **test_01** | pay rentables with full amount | "[Payment]\n" + String.format(">%-5s\t$%.2f\n", "BOX2333", allRentables[0].getPrice()) + String.format("\ndiscount: %.0f percent off\n", (1-allClients[0].getDiscount()) * 100) + String.format("total: $%.2f\n", allRentables[0].getPrice() * allClients[0].getDiscount()) | As expected |
| T002 | **test_02** | pay rentables with discounts | "[Payment]\n" + String.format("\ndiscount: %.0f percent off\n", (1-allClients[0].getDiscount()) * 100) + String.format("total: $%.2f\n", 0.0) | As expected |
| T003 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |
| T004 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |

| | | | | |
|---|---|---|---|---|
| T005 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |
| T006 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |
| T007 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |
| T008 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |
| T009 | **test_03** | test undo and redo function | undo and then redo paying the bags | As expected |

# TestDataLoader_S.java

| **Purpose:** To test whether the DataLoader_S class can properly load and handle scenic spot data from JSON files | | | | |
|---|---|---|---|---|
| Test case ID | **Test Name** | Input Description | Expected Result | Actual Output |
| T001 | **test_01** | DataLoader_S.loadData() with valid JSON file | "[Checkin list]\n" | As expected |
| T002 | **test_02** | return rentable with correct rentable ID | "[Checkin list]\n" + String.format("> %s\n", "BOX9526") | As expected |

# TestDateRange.java

| **Purpose:** To test the functionality of DateRange class for handling date ranges, including creation, retrieval, and date calculation | | | | |
|---|---|---|---|---|
| Test case ID | **Test Name** | Input Description | Expected Result | Actual Output |
| T001 | **test_01** | DateRange(2024-03-01, 2024-03-05).getStartDate() | LocalDate of 2024-03-01 | As expected |
| T002 | **test_02** | DateRange (2024-03-01, 2024-03-05).getEndDate()” | LocalDate of 2024-03-05 | As expected |
| T003 | **test_03** | DateRange(2024-03-01, 2024-03-05).getDays() | 5 days | As expected |

| | | | | |
|---|---|---|---|---|
| T004 | **test_04** | DateRange (2024-03-01, 2024-03-01).getDays() | 1 day | As expected |
| T005 | **test_05** | DateRange( 2024-03-01, 2024-03-05).getAllDates() | List of 5 consecutive dates | As expected |
| T006 | **test_06** | DateRange (2024-03-01, 2024-03-01).getAllDates() | List with single date | As expected |
| T007 | **test_07** | DateRange( 2024-01-01, 2024-12-31).getDays() | 366 days (leap year) | As expected |
| T008 | **test_08** | Multiple calls to getAllDates() | Different instances with same content | As expected |

| | | | | |
|---|---|---|---|---|
| T009 | **test_09** | Modify returned getAllDates() list | Original DateRange unaffected | As expected |
| T010 | **test_10** | DateRange (2024-01-01, 2024-12-31).getAllDates() | List of 366 consecutive dates | As expected |
| T011 | **test_11** | DateRange( 2024-03-30, 2024-04-02).getAllDates() | List of 4 dates crossing month | As expected |
| T012 | **test_12** | DateRange (2024-12-30, 2025-01-02).getAllDates() | List of 4 dates crossing year | As expected |

# TestDijkstra.java

**Purpose:** To test the Dijkstra algorithm implementation for finding shortest paths between metro stations in a graph

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | test_01 | new Dijkstra(graph, "Central", "Tsim Sha Tsui") | Non-null Dijkstra object | As expected |
| T0602 | Test_02 | Dijkstra.findShortestPath() from Central to Tsim Sha Tsui | Valid path list starting with Central and ending with Tsim Sha Tsui" | As expected |
| T003 | Test_03 | Dijkstra with non-existent station names | Empty path list | As expected |
| T004 | test_04 | Dijkstra with same start and end station (Central) | Single-node path containing only start station | As expected |

| | | | | |
|---|---|---|---|---|
| T0605 | **test_05** | Dijkstra.getShortestPathString() from Central to Tsim Sha Tsui | String containing both stations and "=>" separator | As expected |

# TestEdge.java

**Purpose:** To test the Edge class functionality for managing connections between metro stations in the route graphl

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testConstructor | new Edge(station1, station2) | Non-null Edge object | As expected |
| T002 | testGetWeight | edge1.getWeight() | Non-negative weight value | As expected |
| T003 | testGetStations | edge1.getStation1(), edge1.getStation 2() | Match with input stations | As expected |
| T004 | testToString | edge1.toString() | "Central -> Kowloon Bay (weight mins)" | As expected |
| T005 | testWithNullStations | new Edge(null, station2), new Edge(station1, null) | NullPointerException / As expected | As expected |

# TestFindAttraction.java

| **Purpose:** To test the FindAttraction class's ability to search for attractions by name in various lists | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testFindNameZh** | FindAttraction.find(nameZh) | Returns matching attractionA | As expected |
| T002 | **testFindName** | FindAttraction.find(name) | Returns matching attraction | As expected |
| T003 | **testFindNull** | FindAttraction.find("not exist") | Returns null | As expected |
| T004 | **testFindInListSuccess** | FindAttraction.find(customList, existingName) | Returns matching attraction | As expected |

| | | | | |
|---|---|---|---|---|
| T005 | **testFindInListFail** | FindAttraction.find(customList, nonExistentName) | Returns null | As expected |
| T006 | **testFindInEmptyList** | FindAttraction.find(emptyList, name) | Returns null | As expected |
| T007 | **testFindNullName** | FindAttraction.find(null) | Returns null | As expected |

# TestGraph.java

**Purpose:** To test the Graph class functionality for managing metro station nodes and their connection

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testConstructor | new Graph() | Empty graph with no nodes or edges | As expected |
| T002 | testAddSingleNode | Add single node to graph | 1 node, 0 edges | As expected |
| T003 | testAddMultipleNodes | Add three nodes sequentially | 3 nodes, 3 edges with proper connections | As expected |
| T004 | testToString | Graph with two nodes toString() | String containing station names and "mins" | As expected |
| T005 | testGetNodesAndEdges | Graph with two nodes, getNodes() and getEdges() | 2 nodes, 1 edge | As expected |

# TestLocationUtils.java

**Purpose:** To test location string parsing and region identification functionality

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testConstructor | new LocationUtils() | Non-null LocationUtils object | As expected |
| T002 | testGetRegionByLocation_NullInput | getRegionByLocation(null) | null | As expected |
| T003 | testGetRegionByLocation_EmptyInput | getRegionByLocation("") | null | As expected |
| T004 | testGetRegionByLocation_SinglePart | getRegionByLocation("Central") | "Central" | As expected |
| T005 | testGetRegionByLocation_MultipleParts | getRegionByLocation("Causeway Bay, Hong Kong")"H | "Hong Kong" | As expected |
| T006 | testGetRegionByLocation_NewTerritories | getRegionByLocation("Tsuen Wan, N.T.") | "New Territories" | As expected |

# TestMain.java

**Purpose:** To test the main program flow, input handling, and overall system integration

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testMainFlow | Complete user input sequence with valid dates, regions, preferences | Program runs successfully with welcome message | As expected |
| T002 | testErrorHandling | Invalid input followed by valid input sequence | Program handles invalid input and continues | As expected |
| T003 | testEmptyInput | Empty input followed by valid input sequence | Program handles empty input and continues | As expected |
| T004 | testEarlyExit | Input sequence terminated with early exit | Program exits gracefully | As expected |
| T005 | testInvalidCommand | Invalid command followed by valid commands | Program handles invalid command and continues | As expected |
| T006 | testLongRunning | Extended sequence of print commands | Program handles long running operations | As expected |

# TestNode.java

| **Purpose:** To test Node class functionality for managing metro station and attraction information | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructor** | new Node(attraction, "test station") | Non-null Node object | As expected |
| T002 | **testGetName** | node.getName() | "test station" | As expected |
| T003 | **testToString** | node.toString() | "test station" | As expected |
| T004 | **testResult** | node.result() | "test station" | As expected |
| T005 | **testWithNullAttraction** | new Node(null, "test station") | NullPointerException | As expected |

# TestOperatingHours.java

| **Purpose:** To test the functionality of resetting the Selected instance using the TestReflectUtils class. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructorWithStringInput** | new OperatingHours ("09:00-17:00") | Open: 09:00, Close: 17:00 | As expected |
| T002 | **testIsWithinOperatingHoursTrue** | Check 10:00, 9:00, 17:00 | All return true | As expected |
| T003 | **testIsWithinOperatingHoursFalse** | Check 8:59, 17:01 | All return false | As expected |
| T004 | **testIsWithinOperatingHoursEdgeCase** | Check opening and closing times | Both return true | As expected |
| T005 | **testGetAndSetOpeningTime** | Set opening time to 08:00 | Returns 08:00 | As expected |
| T006 | **testGetAndSetClosingTime** | Set closing time to 18:00 | Returns 18:00 | As expected |
| T007 | **testHumanReadable** | hours.humanReadable() | “09:00 - 17:00" | As expected |
| T008 | **testToString** | hours.toString() | "OperatingHours{openingTime=09:00, closingTime=17:00}" | As expected |

# TestPlaza.java

| **Purpose:** To test Node class functionality for managing metro station and attraction information | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testDefaultConstructor** | new Plaza() | Non-null Plaza object | As expected |
| T002 | **testParameterizedConstructor** | New Plaza with all parametersA | All fields match input values | As expected |
| T003 | **testSettersAndGetters** | Set and get all Plaza fields | All getters return set values | As expected |
| T004 | **testGetRegion** | plaza.getRegion() | Non-null region string | As expected |
| T005 | **testToString** | plaza.toString() | String containing all field values | As expected |
| T006 | **testEdgeCases** | Test null, empty, and extreme values | Handle edge cases correctly | As expected |

# PreferenceCollectorTest.java

**Purpose:** To test PreferenceCollector class functionality for collecting and managing user preferences for attractions and restaurants

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | **testSingleton** | PreferenceCollector.getInstance() called twice | Two instances should reference same object | As expected |
| T002 | **testCollectAttractionChoices_ValidPlazaInput** | User inputs multiple valid plaza selections "1 2" | Selected plazas list should contain all valid choices ("Times Square", "Pacific Place") | As expected |
| T003 | **testCollectAttractionChoices_EmptyInput** | User provides empty input string | Empty list returned from collection method | As expected |
| T004 | **testCollectAttractionChoices_InvalidInput** | User inputs mix of valid/invalid choices "invalid 1 3 abc" | List contains only valid plaza selections ("Times Square") | As expected |
| T005 | **testCollectRestaurantChoice_ValidInput** | User selects valid restaurant option "1" | Returns correctly selected restaurant object ("Test Restaurant 1") | As expected |

| | | | | |
|---|---|---|---|---|
| T006 | **testCollectRestaurantChoice_InvalidThenValidInput** | Sequential inputs: "invalid", "0", "4", "1" | After invalid attempts, returns correct restaurant on valid input | As expected |
| T007 | **testCollectRestaurantChoice_BoundaryValues** | Test boundary values: "-1", "0", size+1, "1" | Handles invalid boundary cases and accepts valid input | As expected |
| T008 | **testEmptyLists** | Empty attraction list provided for selection | Returns empty result list | As expected |
| T009 | **testCollectAttractionChoices_NonPlazaType** | Non-plaza attraction selection with input "1" | Successfully collects single non-plaza attraction choice | As expected |
| T010 | **testCollectAttractionChoices_LargeNumbers** | Test large number input "999999999 1" | Handles large numbers and returns valid selection | As expected |

# PriceRangeTest.java

| **Purpose:** To test PriceRange class functionality for handling different price range formats and validations | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructorWithNullAndEmpty** | Pass null and empty string to constructor | Both min and max price should be 0 | As expected |
| T002 | **testConstructorWithMichelinLabel** | Pass "米芝蓮2024" to constructor | Special label flag set true, min=400, max=Integer.MAX_VALUE | As expected |
| T003 | **testConstructorWithPriceRange** | Pass "100-200" to constructor | min=100, max=200, specialLabel=false | As expected |
| T004 | **testConstructorWithPriceBelow** | Pass "200以下" to constructor | min=0, max=200, specialLabel=false | As expected |
| T005 | **testConstructorWithPriceAbove** | Pass "200以上" to constructor | min=200, max=Integer.MAX_VALUE, specialLabel=false | As expected |
| T006 | **testConstructorWithCurrencySymbol** | Pass "$100-200" to constructor | min=100, max=200 ignoring currency symbol | As expected |
| T007 | **testIsWithinRange** | Test various range comparisons | Correctly identify overlapping ranges | As expected |

| | | | | |
|---|---|---|---|---|
| T008 | **testToString** | Test string representation of different price ranges | Correct format for each range type | As expected |
| T009 | **testExceptionalCases** | Test invalid inputs: "abc-def", "100", "abc以上", "abc以下" | Appropriate exceptions thrown | As expected |

# RcommendationGeneratorTest.java

**Purpose:** To test RecommendationGenerator class functionality for generating travel recommendations based on user preferences

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testSingleton | Multiple calls to getInstance() | Same instance returned for all calls | As expected |
| T002 | testThreadSafeSingleton | 10 concurrent threads calling getInstance() | Single instance created and returned across all threads | As expected |
| T003 | testGenerateRecommendations | Valid scenicSpots, plazas and preferences data | Non-null recommendation object with valid spots, restaurants and plazas lists | As expected |
| T004 | testEmptyInputs | Empty lists for scenicSpots and plazas | Valid recommendation object returned despite empty inputs | As expected |
| T005 | testNullPreferences | Null preferences parameter | NullPointerException thrown | As expected |

# Recommender_PTest.java

| **Purpose:** To test RecommendationGenerator class functionality for generating travel recommendations based on user preferences | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testBasicRecommendation** | Basic preferences with shopping tag and Hong Kong region | Non-null results with "Harbour City" as top recommendation | As expected |
| T002 | **testRatingFilter** | Filter plazas by rating threshold | All recommended plazas have rating >= 4.0 | As expected |
| T003 | **testPopularSorting** | Sort plazas by popularity (review count) | Recommendations sorted in descending order by review count | As expected |
| T004 | **testTagFiltering** | Filter plazas by "Luxury" tag | Non-empty list with plazas containing "Luxury" tag | As expected |
| T005 | **testFallbackStrategy** | Invalid tag preference "NonExistentTag" | Non-empty fallback recommendations from other regions | As expected |

# Recommender_RTest.java

| **Purpose:** To test RecommendationGenerator class functionality for generating travel recommendations based on user preferences | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testRecommendByPreferences_SingleDay** | Single day preferences with Hong Kong region and price range 100-200 | Non-null results with single day recommendations | As expected |

# Recommender_STest.java

| **Purpose:** To test Recommender_S class functionality for generating scenic spot recommendations based on region and tag preferences | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testBasicRecommendation** | Region="Hong Kong Island", Tag="Viewpoint", Popular=true | Non-null results with "Victoria Peak" as top recommendation | Empty recommendation list returned |
| T002 | **testEmptyList** | Region="Unknown Region" | Empty recommendation list returned | As expected |
| T003 | **testTagFiltering** | Region="Hong Kong Island", Tag="Theme Park" | Non-empty list with spots containing "Theme Park" tag | As expected |

# RemoveCommandTest.java

**Purpose:** To test RemoveCommand class functionality for removing selected attractions from daily itinerary

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testRemoveSuccess | Date="2024-01-01", attraction=first attraction | Remove successful, selected list size reduced to 1 | As expected |
| T002 | testRemoveFail1 | Date="2024-01-03", attraction=second attraction | Remove failed due to non-existent date | As expected |
| T003 | testRemoveFail2 | Date="2024-01-01", attraction=fifth attraction | Remove failed due to non-existent attraction | As expected |
| T004 | testRemoveFail3 | Date="2024-01-01", attraction=third attraction | Remove failed due to attraction not in daily selection | As expected |

# RestaurantDisplayTest.java

| **Purpose:** To test RemoveCommand class functionality for removing selected attractions from daily itinerary | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testInstanceCreation** | Create RestaurantDisplay instance | Valid instance created | As expected |
| T002 | **testDisplayRecommendationsWithNormalData** | Single day, single area restaurant data | Output contains day, area, and restaurant details | As expected |
| T003 | **testDisplayRecommendationsWithEmptyData** | Empty recommendations map | Shows day header without area section | As expected |
| T004 | **testDisplayRecommendationsWithNoOperatingHours** | Restaurant without operating hours | Displays "Operating Hours: N/A" | As expected |
| T005 | **testDisplayRecommendationsWithMultipleDays** | Three days of recommendations | Contains headers for all three days | As expected |
| T006 | **testDisplayRecommendationsWithMultipleAreasPerDay** | Single day with multiple areas | Shows all areas with their restaurants | As expected |
| T007 | **testDisplayRecommendationsFormatting** | Standard restaurant data | Contains all required formatting elements and fields | As expected |

# RestaurantTest.java

**Purpose:** Test that the Restaurant object is correctly initialized with the given constructor parameters.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testParameterizedConstructor | A Restaurant object is created with predefined values for name, location, price range, review score, and operating hours. | The Restaurant object's properties are correctly initialized and can be accessed with their respective getter methods. | As expected |
| T002 | testSettersAndGetters | A Restaurant object is created with initial values. | Each getter method should return the updated value after the setter method has been called. | As expected |
| T003 | testIsOpenAt | A Restaurant object is created with a list of OperatingHours containing a single entry: from 9:00 AM to 10:00 PM. | The restaurant.getOperatingHours() method returns a list of OperatingHours objects. | As expected |
| T004 | testIsOpenAtWithMultipleTimeSlots | A Restaurant object is created with a price range value "100-200". | The restaurant.getPriceRange() method returns an instance of the PriceRange class. | As expected |
| T005 | testToString | A Restaurant object is created with review count 100 and review score 4.5. | restaurant.getReviewCount() returns 100. restaurant.getReviewScore() returns 4.5. | As expected |
| T006 | testGetAvgExpense | A Restaurant object is created with initial operating hours. | restaurant.getOperatingHours() returns the updated list with the new operating hours. | As expected |

# ResultDisplayTest.java

**Purpose:** Test the ResultDisplay class to ensure it correctly displays the details of objects like Restaurant and Plaza when passed to the displayPlans() method.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testDisplayRestaurantDetails() | A Restaurant object with properties like name, location, and operating hours is added to a map for display.. | The output should contain the restaurant's name, location, operating hours, and review score. | As expected |
| T002 | testDisplayPlazaDetails() | A Plaza object is created with properties like name, metro station, and capacity and added to a map with a specific date. | The output should display the plaza's name, metro station, and capacity. | As expected |
| T003 | testDisplayRestaurantWithEmptyDetails() | A Restaurant object with empty or default properties (e.g., empty name, no operating hours) is created and added to a map with a specific date. | The output should display the default or empty values for the restaurant's properties, such as an empty name or missing operating hours. | As expected |
| T004 | testDisplayPlazaWithEmptyDetails() | A Plaza object with empty or default properties (e.g., empty name, no metro station) is created and added to a map with a specific date. | The output should display the plaza's default or empty values, such as an empty name or missing metro station. | As expected |
| T005 | testDisplayMultipleRestaurants() | The output should display details for each restaurant, including name, location, operating hours, and review score, for all the restaurants in the | The output should display details for each restaurant, including name, location, operating hours, and review score, for all the restaurants in the map. | As expected |

| | | | | |
|---|---|---|---|---|
| | | map. | | |
| T006 | **testDisplayMultiplePlazas()** | Multiple Plaza objects with different properties (e.g., names, metro stations, and capacities) are created and added to the map under different dates. | The output should display details for each plaza, including name, metro station, and capacity, for all plazas in the map. | As expected |

# RouteGeneratorTest.java

**Purpose:** Test the functionality of the RouteGenerator class, including its singleton pattern and route generation capabilities. The tests ensure that the RouteGenerator instance behaves as expected when interacting with attractions and generating routes.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testSingleton() () | Two instances of the RouteGenerator class are obtained using the getInstance() method. | The two instances should be the same, confirming that the RouteGenerator class follows the singleton pattern. | As expected |
| T002 | testGenerateRouteWithMultipleAttractions() | A list of multiple Attraction objects (e.g., TestAttraction instances) is provided to the RouteGenerator to generate a route. | The output should contain a generated route based on the provided attractions, with the correct order and details for each attraction. | As expected |
| T003 | testGenerateRouteWithSingleAttraction() | A list containing a single Attraction object is provided to the RouteGenerator to generate a route. | The output should generate a route containing only the single attraction, with correct details displayed. | As expected |
| T004 | testGenerateRouteWithEmptyList() | An empty list of attractions is provided to the RouteGenerator. | The output should handle the empty input gracefully, potentially returning an empty route or an appropriate message indicating no attractions were provided. | As expected |
| T005 | testAttractionDetails()() | A TestAttraction object is created with specific details (name, location, metro station), and | The output should correctly display the attraction's details, including its name, location, and metro station, in the generated route. | As expected |

| | | | | |
|---|---|---|---|---|
| | | used within the route generation process. | | |

## RoutePrinterTest.java

**Purpose:** Test the functionality of the RoutePrinter class, ensuring it correctly prints route information when provided with different types of input, including an empty map, valid data, and specific attraction details.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | **testPrintEmptyMap()()** | An empty map of Node objects (representing routes) is provided to the RoutePrinter to print. | The output should indicate that the map is empty, or no route details should be printed, ensuring that the method handles empty inputs correctly. | As expected |
| T002 | **testPrintRouteWithMultipleNodes()** | A map containing multiple Node objects, each with a list of attractions (e.g., TestAttraction instances), is provided to the RoutePrinter. | The output should correctly print the details for each attraction in the route, including the name and location of each attraction in the correct order. | As expected |
| T003 | **testPrintRouteWithSingleNode** | A map containing a single Node object, with one or more attractions, is provided to the RoutePrinter. | The output should print the details of the single node and its attractions in the correct format. | As expected |
| T004 | **testPrintRouteWithEmptyAttractions()** | A map containing one or more Node objects, but with empty lists of attractions, is provided to the RoutePrinter. | The output should indicate that no attractions are available to print, or show an appropriate message indicating the absence of attractions. | As expected |
| T005 | **testPrintAttractionDetails()()** | A map containing Node objects with TestAttraction instances, each containing specific details such as name and location, is provided to the RoutePrinter. | The output should correctly display the details for each TestAttraction, including name and location, in the proper format. | As expected |

# ScenicSpotTest.java

**Purpose:Test the functionality of the ScenicSpot class, including its constructors, setter and getter methods, and data handling (such as location, features, review count, etc.).**

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testDefaultConstructor()() | A ScenicSpot object is created using the default constructor with no initial values. | The output should confirm that the object is not null, indicating that the default constructor creates a valid ScenicSpot object. | As expected |
| T002 | testSettersAndGetters() | A ScenicSpot object is created with pre-defined values for properties such as name, location, metro station, review count, region, and features. | The output should verify that the getter methods return the correct values set through the corresponding setter methods. | As expected |
| T003 | testSetterGetterName | A ScenicSpot object is created, and the name property is set using the setter method. | The output should ensure that the getName() method returns the correct value set by setName(). | As expected |
| T004 | testSetterGetterNameZh() | A ScenicSpot object is created, and the nameZh property is set using the setter method. | The output should ensure that the getNameZh() method returns the correct value set by setNameZh(). | As expected |
| T005 | testSetterGetterLocation()() | A ScenicSpot object is created, and the location property is set using the setter method. | The output should ensure that the getLocation() method returns the correct value set by setLocation(). | As expected |
| T006 | testSetterGetterMetroStation() | A ScenicSpot object is created, and the metroStation property is set using the setter method. | The output should ensure that the getMetroStation() method returns the correct value set by setMetroStation(). | As expected |

| | | | | |
|---|---|---|---|---|
| T007 | **testSetterGetterReviewCount()** | A ScenicSpot object is created, and the reviewCount property is set using the setter method. | The output should ensure that the getReviewCount() method returns the correct value set by setReviewCount(). | As expected |
| T008 | **testSetterGetterRegion** | A ScenicSpot object is created, and the region property is set using the setter method. | The output should ensure that the getRegion() method returns the correct value set by setRegion(). | As expected |
| T009 | **testSetterGetterFeatures** | A ScenicSpot object is created, and the features list is set using the setter method. | The output should ensure that the getFeature() method returns the correct list of features set by setFeature(). | As expected |

# SelectedTest Class.java

**Purpose:** Test the functionality of the Selected class, particularly focusing on how it handles and processes a collection of Plaza objects (e.g., sorting, adding to selection, etc.). The tests ensure that the expected behaviors, such as correctly adding, sorting, or processing the selection, are implemented properly.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | **testAddPlazaToSelection()** | A Plaza object (p1, p2, etc.) is added to the Selected selection list for a specific date (e.g., "2024-01-01").. | The output should confirm that the plaza has been successfully added to the selection list for the given date. | As expected |
| T002 | **testSortPlazasByReviewScore** | A collection of Plaza objects is created with varying review scores and added to the selection list. | The output should confirm that the plazas are correctly sorted by their review scores in descending order. | As expected |
| T003 | **testSelectPlazasByCriteria** | A selection of Plaza objects is made based on specific criteria (e.g., review score greater than 4.0, or specific opening times). | The output should display only those Plaza objects that meet the selection criteria, such as those with a review score greater than 4.0.. | As expected |
| T004 | **testReflectUtilsFunctionality()** | The TestReflectUtils class is used to test reflection methods on Selected or related objects. | The output should confirm that the reflection functionality correctly identifies and interacts with the fields of the Selected class, providing accurate data. | As expected |
| T005 | **testSelectedMapStructure** | A Map is created to hold selected Plaza objects for various dates, and the map is populated with data. | Confirm that the map is correctly structured and populated with the right data, ensuring the Map<String, List<Attraction>> is correctly mapped by date to a list of | As expected |

| | | | | |
|---|---|---|---|---|
| | | | attractions. | |

# SelectionManagerTest Class.java

**Purpose:** Test the functionality of the SelectionManager class, focusing on its singleton pattern, user input handling, and interaction with various objects such as ScenicSpot, Restaurant, and Plaza. The tests verify that the SelectionManager correctly manages selections based on different inputs and scenarios.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | **testSingletonPattern()** | Test if two SelectionManager instances created via getInstance() are the same (i.e., confirm the singleton pattern). | The output should confirm that the two instances are the same, indicating that the SelectionManager class follows the singleton pattern. | As expected |
| T002 | **testMockUserInput** | Simulate user input by feeding a specific string into the system using System.setIn(). | The output should confirm that the SelectionManager correctly processes the mocked user input and updates its state or performs the correct action based on the input.. | As expected |
| T003 | **testHandleEmptyScenicSpot** | Test the SelectionManager with an empty list of ScenicSpot objects for a specific date. | The output should confirm that the manager handles the empty list correctly, either by showing no results or displaying an appropriate message. | As expected |
| T004 | **testHandleEmptyRestaurantList()** | Test the SelectionManager with an empty list of Restaurant objects for a specific date. | The output should confirm that the manager handles the empty restaurant list correctly, showing no available restaurants or displaying an appropriate | As expected |

| | | | | |
|---|---|---|---|---|
| | | | message. | |
| T005 | **testAddScenicSpotToSelection** | A ScenicSpot object is added to the SelectionManager for a specific date (e.g., "2024-01-01").. | The output should confirm that the ScenicSpot has been correctly added to the selection list for that date. | As expected |
| T006 | **testAddRestaurantToSelection** | A Restaurant object is added to the SelectionManager for a specific date. | The output should confirm that the Restaurant object has been added to the selection for that date, and its details are correctly stored and retrievable. | As expected |
| T007 | **testUserSelectionOfPlaza** | A user selects a Plaza object from a list of available plazas through the input system. | The output should confirm that the correct plaza has been selected based on the user's input | As expected |
| T008 | **testSelectionManagerWithMultipleAttractions** | The SelectionManager is provided with multiple ScenicSpot, Restaurant, and Plaza objects for selection. | The output should confirm that the manager handles multiple attractions correctly, potentially sorting or displaying them based on user preferences or predefined criteria. | As expected |

# SelectedTest Class.java

**Purpose:** Test the functionality of the SelectionProcessor class, ensuring it correctly processes selections based on simulated user inputs and data. The tests verify how the SelectionProcessor interacts with other components like PreferenceCollector, ScenicSpot, Restaurant, and Plaza.

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | **testProcessSelections_SingleDayComplete()** | Simulated user input is provided for selecting preferences for a single day (1 2\n1\n1\n1\n1\n), where the user selects different options related to attractions, restaurants, and possibly other criteria. | The output should correctly reflect the user's selections, processing the inputs and selecting the appropriate ScenicSpot, Plaza, and Restaurant for the specified date (2024-01-01). The selection should match the simulated input, and the data should be processed as expected. | As expected |
| T002 | **testProcessSelections_MultipleDays** | Simulated user input is provided for multiple days with various preferences, representing multiple selections over different days. | The output should correctly handle and process selections for multiple days, reflecting the correct selections of ScenicSpot, Plaza, and Restaurant for each day based on the user's input. | As expected |
| T003 | **testProcessSelections_WithEmptyInput** | Simulated user input is empty or incomplete (e.g., no selection is made). | The output should handle the case where no selections are made, possibly by showing a message indicating that no selections were processed or prompting the user to make selections. | As expected |

| | | | | |
|---|---|---|---|---|
| T004 | **testProcessSelections_WithInvalidInput()** | Simulated user input includes invalid or out-of-range values (e.g., selecting non-existing options or providing incorrect input types). | The output should handle invalid input gracefully, either by showing an error message or prompting the user for valid input. | As expected |
| T005 | **testProcessSelections_WithMultipleRestaurants** | Simulated user input specifies a date with multiple Restaurant selections.. | The output should correctly display or process multiple restaurant selections for the given date, ensuring that the system handles multiple entries as expected. | As expected |

# TraverseNewTesting.java

| **Purpose:** To test the functionality of the Traverse_new_testing class and its pathfinding capabilities within a graph of attractions. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructor** | Initialize Traverse_new_testing object | traverse object should not be null | As expected |
| T002 | **testValidPath** | Find shortest path from breakfastNode to lunchNode and dinnerNode | Path should not be null and should include all nodes; order constraints should be satisfied | As expected |
| T003 | **testPathStartsWithBreakFast** | Check path from breakfastNode to lunchNode and dinnerNode | Path should start with breakFastNode | As expected |
| T004 | **testGetShortestPathString** | Get shortest path string from breakfastNode to lunchNode and dinnerNode | Path string should not be null and include arrow separator and relevant nodes | As expected |
| T005 | **testGetShortestPath** | Call findShortestPath and get the shortest path | Retrieved path should not be null and should not be empty | As expected |
| T006 | **testPathCalculation** | Validate connections between nodes in the found shortest path | Each pair of adjacent nodes in the path should be connected by | As expected |

| | | | | |
|---|---|---|---|---|
| | | | an edge | |
| T007 | **testNoAttractionBetweenRestaurants** | Create a graph with insufficient attractions between restaurants | Should return an empty path when looking for a path. | As expected |

# TraverseTest.java

| **Purpose:** To test the functionality of the Traverse class and its pathfinding capabilities within a graph of attractions. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructor** | Initialize Traverse object | traverse object should not be null | As expected |
| T002 | **testFindShortestPath** | Call findShortestPath() on the traverse object | Path should not be null and should include all nodes | As expected |
| T003 | **testGetShortestPathString** | Get shortest path string | Path string should not be null and include arrow separator and relevant nodes | As expected |
| T004 | **testGetShortestPath** | Call findShortestPath and get the shortest path | Retrieved path should not be null and should not be empty | As expected |
| T005 | **testPathCalculation** | Validate connections between nodes in the found shortest path | Each pair of adjacent nodes in the path should be connected by an edge | As expected |

# TripPlannerTest.java

| **Purpose:** To test the functionality of the TripPlanner class and its trip planning capabilities based on user preferences. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testTripPlanning** | Initialize TripPlanner with preferences and call planTrip() | Recommendation should not be null; start date should be "2024-03-20" | As expected |

# TripRecommendationTest.java

| **Purpose:** To test the functionality of the TripRecommendation class, including its construction, data handling, and output methods. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **test_01** | Initialize TripRecommendation with sample data | Object should not be null; getters should return correct data | As expected |
| T002 | **test_02** | Call getCombinedDailyPlan() on tripRecommendation | Combined plan should not be null and contain "Day1"; includes all types of recommendations | As expected |
| T003 | **test_03** | Call toString() on tripRecommendation | Output should contain relevant information about trip recommendation | As expected |
| T004 | **test_04** | Initialize TripRecommendation with empty data | All lists should be empty; toString() should handle empty data gracefully | As expected |
| T005 | **test_05** | Initialize TripRecommendation with null values | Getters should return null; toString() should handle null data gracefully | As expected |

# UserInputHandlerTest.java

| **Purpose:** To test the functionality of the UserInputHandler class, focusing on user input handling and preference collection. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testGetDateRange_ValidInput** | Valid date range input | Start date should be "2024-03-01", end date "2024-03-05" | As expected |
| T002 | **testGetDateRange_EndDateBeforeStartDate** | End date before start date input | Start date should be "2024-03-05", end date "2024-03-06" | As expected |
| T003 | **testGetPopularPlazaPreference** | Valid input for popular plaza preference | Should return true for "yes", false for invalid input | As expected |
| T004 | **testGetHighRatedPlazaPreference** | Valid input for high-rated plaza preference | Should return true for "yes", false for invalid input | As expected |
| T005 | **testGetDateRange_InvalidDateFormat** | Invalid date format input | Should return valid date range with start "2024-03-01", end "2024-03-05" | As expected |
| T006 | **testGetDateRange_InvalidThenValid** | Invalid date followed by valid date input | Should return valid date range with start "2024-03-01", end "2024-03-05" | As expected |
| T007 | **testGetRegionPreference** | Valid region preference input | Should return correct region based on input | As expected |

# UserPreferencesContainerTest.java

**Purpose:** To test the functionality of the UserPreferenceContainer class and its Builder pattern

| Test case ID | Test Name | Input Description | Expected Result | Actual Output |
|---|---|---|---|---|
| T001 | testBasicBuild | Build a UserPreferenceContainer with multiple preferences | Container should not be null; should match date range and budget | As expected |
| T002 | testEmptyDateRange | Build a UserPreferenceContainer with a single-day date range | Container should contain one date in the range | As expected |
| T003 | testBuilderChaining | Test the chaining of Builder methods | Should not return null after chaining | As expected |

# UserPreferences_PTest.java

| **Purpose:** To test the functionality of the UserPreferences_P class, focusing on preference management and data integrity. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testEmptyConstructor** | Initialize with empty constructor | Should not be null; should have correct start and end dates, and be empty in daily preferences | As expected |
| T002 | **testFullConstructor** | Initialize with full constructor data | Should match provided regions, tags, popularity, and rating filters | As expected |
| T003 | **testAddPreference** | Add a preference to the container | Should correctly reflect the added preference | As expected |
| T004 | **testGetDuration** | Check duration calculation with valid dates | Should return the correct duration | As expected |
| T005 | **testDateSettersAndGetters** | Set and get new start and end dates | Should return the updated dates | As expected |
| T006 | **testDailyPreferencesSettersAndGetters** | Set daily preferences and retrieve them | Should match the set values correctly | As expected |
| T007 | **testIndexOutOfBoundsException** | Access index out of bounds after adding preferences | Should throw IndexOutOfBoundsException | As expected |
| T008 | **testToString** | Verify the string representation of the preferences | Should contain all relevant preference information | As expected |

# UserPreferences_RTest.java

| **Purpose:** To test whether the search function inside RequestStorer.java can work well | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testConstructorAndGetters** | Initialize with constructor and check getters | Should return correct start date, end date, regions, and budget | As expected |
| T002 | **testSetters** | Update preferences using setters and check results | Should reflect updated values correctly | As expected |
| T003 | **testGetDays** | Calculate duration in days based on start and end dates | Should return correct number of days | As expected |
| T004 | **testToString** | Verify string representation of preferences | Should match expected string format | As expected |
| T005 | **testGetFilePathsForDay** | Generate file paths based on region and day | Should return correct path count and format | As expected |
| T006 | **testRandomnessWithDifferentSeeds** | Generate paths with different seeds | Should return different paths | As expected |
| T007 | **testConsistencyWithSameSeed** | Generate paths with the same seed | Should return the same paths | As expected |

# UserPreferences_STest.java

| **Purpose:** To test the functionality of the UserPreferences_S class, focusing on preference management for dates, regions, and tags. | | | | |
|---|---|---|---|---|
| **Test case ID** | **Test Name** | **Input Description** | **Expected Result** | **Actual Output** |
| T001 | **testAddAndGetSinglePreference** | Add and retrieve a single preference | Should return correct region, tag, and popularity | As expected |
| T002 | **testGetNonExistentDatePreference** | Retrieve preference for a non-existent date | Should return null for region and tag; false for popularity | As expected |
| T003 | **testAddMultiplePreferences** | Add multiple preferences and retrieve all | Should return correct sizes for regions, tags, and popularity maps | As expected |
| T004 | **testUpdateExistingPreference** | Update an existing preference for a date | Should reflect updated values while maintaining correct size | As expected |
| T005 | **testEmptyStrings** | Add preference with empty strings as parameters | Should handle empty strings and return them correctly | As expected |
| T006 | **testNullValues** | Add preference with null values as parameters | Should handle null inputs, returning nulls for region and tag | As expected |