MARKING REPORT

**Group number: \_26\_**

|  |  |
| --- | --- |
| Name | Student ID |
| 1. Andrew Woon Mun Hong | 2102584 |
| 2. Gan Wei Jia | 2102338 |
| 3. Liew Chun Kin | 2103388 |
| 4.Lim Jun Hau | 2102629 |

**Marks breakdown**

Part A: Test Plan (10 marks)

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Max Mark** | **Marks Obtained** | **Remark/Comment** |
| Test objective, scope and test basis | 5 |  |  |
| Test condition, entry and exit criteria | 5 |  |  |

Part B: Test Design (20 marks)

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Max Mark** | **Marks Obtained** | **Remark/Comment** |
| Decision table | 5 |  |  |
| Appropriateness of test cases | 15 |  |  |

Part C: Java Program (application code and test code) (70 marks)

|  |  |  |  |
| --- | --- | --- | --- |
| **Component** | **Max Mark** | **Marks Obtained** | **Remark/Comment** |
| Setup jar file location to C:\ jar\_files | 2 |  |  |
| Source directories | 3 |  |  |
| Appropriate used of assertsXXX methods. | 10 |  |  |
| Using parameterised tests correctly | 10 |  |  |
| Invalid values are checked for in implemented code, and tests for invalid values are performed. | 10 |  |  |
| Use of mocks or stubs for testing. | 10 |  |  |
| Combining test cases into test suites | 5 |  |  |
| Setting up some tests so that test values are read from a text file instead of hardcoding into test code | 10 |  |  |
| Perform integration testing after unit tests have been completed | 10 |  |  |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A: | B: | C: | Total:  /100 | /20 |

Table Of Contents

Part A:

[Test Plan 3](#_Toc164787700)

[Document Control 4](#_Toc164787701)

[Version History 4](#_Toc164787702)

[Introduction 5](#_Toc164787703)

[Objective 5](#_Toc164787704)

[Scope 5](#_Toc164787705)

[Test Plans 5](#_Toc164787706)

[Test Items 5](#_Toc164787707)

[Features to be tested. 5](#_Toc164787708)

[Features not to be tested. 5](#_Toc164787709)

[Test Basis 5](#_Toc164787710)

[Test Conditions 6](#_Toc164787711)

[Entry Criteria 7](#_Toc164787712)

[Exit Criteria 7](#_Toc164787713)

Part B:  
[Test Case Design 8](#_Toc164787714)

Part C:  
[Class Diagram 22](#_Toc164787715)

Part D:  
[HotelBookingSystem/main 23](#_Toc164787716)

[Booking.java 23](#_Toc164787717)

[Printer.java 26](#_Toc164787718)

[Room.java 27](#_Toc164787719)

[User.Java 28](#_Toc164787720)

[WaitingList.Java 29](#_Toc164787721)

[HotelBookingSystem/test 31](#_Toc164787722)

[AddToWaitingListTest.java 31](#_Toc164787723)

[CancelBookingTest.java 32](#_Toc164787724)

[SetBookingTest.java 34](#_Toc164787725)

[TestSuite.java 42](#_Toc164787726)

[TextFileTest.java 42](#_Toc164787727)

[UserTesting.java 45](#_Toc164787728)

[WaitingListTest.java 45](#_Toc164787729)

# Test Plan

Assignment

## Document Control

|  |  |
| --- | --- |
| **Document Name** | Test Plan |
| **Reference Number** | - |
| **Version** | V1.1 |
| **Project Code** | UECS2354 Software Testing Assignment |
| **Status** | [Completed] |
| **Date Released** | [10/4/2024] |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Name** | **Position** | **Signature** |
| Prepared By: | **Andrew Woon Mun Hong** | **Senior Tester** |  |
| Reviewed By: | **Liew Chun Kin** | **Team Leader** |  |
| Approved By: | **Lim Jun Hau** | **QA Manager** |  |
| Approved By: | **Gan Wei Jia** | **Project Manager** |  |

## Version History

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Release Date** | **Section** | **Amendments** |
| 1.0 | 10/4/2024 | - | - |
| 1.1 | 17/4/2024 | 2.1, 2.2 | Updated MUT and features to be tested |

## 

## Introduction

### Objective

The objective of the test is to ensure that all the components in the application are working perfectly under normal circumstances and ensure that all the bugs found are resolved so that users will be able to use the application with satisfaction. Ultimately to ensure that the Hotel Room Booking System functions are working based on the requirements and conforming to the rules of the room allocation based on the member type.

### Scope

The test scope covers all of the major functions of the Hotel Room Booking System. All of the major functions including the allocation of member types, room allocation rules, waiting list, exclusive rewards for each member type, and the cancellation of the reservation and resolve all the bugs found during the testing phase.

## Test Plans

### Test Items

|  |  |
| --- | --- |
| **Classes** | **MUT** |
| User | setExclReward() |
| Room | checkRoom(), bookRoom(), getAvailableRooms() |
| Booking | setBooking(), cancelBooking() |
| Printer | printInfo() |
| WaitingList | addWaiting(), getWaiting(), removeWaiting() |

### Features to be tested.

- The member type

- Exclusive reward

- Number of rooms to book and the types of rooms available (Standard, Deluxe, VIP)

- Request to fulfill

- Booking cancellation

- Waiting list

### Features not to be tested.

- Printing the booking details

- adding users to the waiting list and removing them from the waiting list

### Test Basis

- The requirements of the project that include the functional and the non-functional requirements.

- The UML diagrams show the data flow of the system.

- The use cases of the application.

- Conformance to the business regulations and requirements

### Test Conditions

<list down all the test conditions>

(Default room allocation)

* VIP member: VIP Room
* Normal member: Deluxe room
* Non-member: Standard

(For VIP Member Booking)

1. To verify VIP member can make booking of maximum of 3 rooms at a time. (If VIP room is not sufficient, can combine with Deluxe/Standard rooms)
2. If VIP room is fully booked, VIP member is allocated to Deluxe Room.
3. If VIP and Deluxe Room are fully booked, Standard Room is fully booked.
4. If all Rooms are booked, VIP member is placed into VIP waiting list.
5. If VIP booked more than 3 rooms, they are placed into VIP waiting list.

(For Normal Member Booking)

1. Normal member can make booking of maximum of 2 rooms at a time. (Can be both Deluxe/ Standard rooms, or 1 VIP 1 Deluxe, or 1 VIP 1 Standard, or 1 Deluxe 1 Standard.)
2. If Deluxe Room is fully booked, normal member without exclusive reward is allocated to a Standard Room.
3. If all room types are fully booked, normal member will be placed into member waiting list.
4. Normal member with exclusive rewards can book a VIP room subjected to availability.
   1. If allocated to VIP room, the exclusive reward is marked redeemed.
   2. If VIP room is unavailable, Standard room will be allocated an the exclusive rewards remain.
5. If all the rooms are booked, normal member will be placed into member waiting list.

(For Non-member Booking)

1. Non-member can make booking of maximum of 1 room at a time.
2. If standard rooms are fully booked, non-member will be placed into normal waiting list.

(Cancellation)

1. Bookings can be cancelled.
2. If the user is on a waiting list, it will be removed from the waiting list.

(Error Handling)

1. Check for invalid input during booking.
2. System error during the booking or cancellation process.

### Entry Criteria

1. The development team has completed coding all the features and details of the application code.

2. The development team has completed the unit testing for the individual system components.

3. The application code has been reviewed and confirmed that it is working in the real environment.

4. The availability of the test environment that is mocking the production environment

5. The test case has been developed and is available to be used for testing.

6. The test case has been reviewed and confirmed that it is feasible to be used for testing.

7. The completion of sign-off for the system to perform integration testing.

### Exit Criteria

1. All the test cases have been executed and have passed the test.

2. There are no changes to be made to the overall application and to be confirmed as the final product after the tests.

3. All of the bugs found during the testing phase have been resolved.

4. All of the test results have been reviewed by the QA management team.

5. The system is considered to be ready for deployment in the real working environment.

6. The results of the User Acceptance Testing (UAT) are deemed satisfactory by the clients.

# Test Case Design

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Test Case#** | **Test Title** | **Test Summary** | **Test Steps** | **Test Data** | **Expected Result** | **Post-condition** | **Actual Result** | **Status** | **Notes** |
| 1 | To verify rule#1 | if condition #1 is true, the action will be action #1 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available=">=3" | Number of VIP room booked="3" | 3 VIP room booked | Number of VIP room booked="3" | Pass | - |
| 2 | To verify rule#2 | if condition #2 is true, the action will be action #2 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="2" Deluxe room available=">=1" | Number of VIP room booked="2" Number of Deluxe room booked="1" | 2 VIP room booked 1 Deluxe room booked | Number of VIP room booked="2" Number of Deluxe room booked="1" | Pass | - |
| 3 | To verify rule#3 | if condition #3 is true, the action will be action #3 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="2" Deluxe room available="0" Standard room available=">=1" | Number of VIP room booked="2" Number of Deluxe room booked="0" Number of Standard room booked="1" | 2 VIP room booked 1 Standard room booked | Number of VIP room booked="2" Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 4 | To verify rule#4 | if condition #4 is true, the action will be action #4 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="1" Deluxe room available=">=2" | Number of VIP room booked="1" Number of Deluxe room booked="2" | 1 VIP room booked 2 Deluxe room booked | Number of VIP room booked="1" Number of Deluxe room booked="2" | Pass | - |
| 5 | To verify rule#5 | if condition #5 is true, the action will be action #5 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="1" Deluxe room available="0" Standard room available=">=2" | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="2" | 1 VIP room booked 2 Standard room booked | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="2" | Pass | - |
| 6 | To verify rule#6 | if condition #6 is true, the action will be action #6 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="1" Deluxe room available="1" Standard room available=">=1" | Number of VIP room booked="1" Number of Deluxe room booked="1" Number of Standard room booked="1" | 1 VIP room booked 1 Deluxe room booked 1 Standard room booked | Number of VIP room booked="1" Number of Deluxe room booked="1" Number of Standard room booked="1" | Pass | - |
| 7 | To verify rule#7 | if condition #7 is true, the action will be action #7 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="0" Deluxe room available=">=3" | Number of VIP room booked="0" Number of Deluxe room booked="3" | 3 Deluxe room booked | Number of VIP room booked="0" Number of Deluxe room booked="3" | Pass | - |
| 8 | To verify rule#8 | if condition #8 is true, the action will be action #8 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="0" Deluxe room available="0" Standard room available=">=3" | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="3" | 3 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="3" | Pass | - |
| 9 | To verify rule#9 | if condition #9 is true, the action will be action #9 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="0" Deluxe room available="2" Standard room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="2" Number of Standard room booked="1" | 2 Deluxe room booked 1 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="2" Number of Standard room booked="1" | Pass | - |
| 10 | To verify rule#10 | if condition #10 is true, the action will be action #10 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="0" Deluxe room available="1" Standard room available=">=2" | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="2" | 1 Deluxe room booked 2 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="2" | Pass | - |
| 11 | To verify rule#11 | if condition #11 is true, the action will be action #11 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available=">=2" | Number of VIP room booked="2" | 2 VIP room booked | Number of VIP room booked="2" | Pass | - |
| 12 | To verify rule#12 | if condition #12 is true, the action will be action #12 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available="1" Deluxe room available=">=1" | Number of VIP room booked="1" Number of Deluxe room booked="1" | 1 VIP room booked 1 Deluxe room booked | Number of VIP room booked="1" Number of Deluxe room booked="1" | Pass | - |
| 13 | To verify rule#13 | if condition #13 is true, the action will be action #13 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available="1" Deluxe room available="0" Standard room available=">=1" | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="1" | 1 VIP room booked 1 Standard room booked | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 14 | To verify rule#14 | if condition #14 is true, the action will be action #14 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available="0" Deluxe room available=">=2" | Number of VIP room booked="0" Number of Deluxe room booked="2" | 2 Deluxe room booked | Number of VIP room booked="0" Number of Deluxe room booked="2" | Pass | - |
| 15 | To verify rule#15 | if condition #15 is true, the action will be action #15 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available="0" Deluxe room available="1" Standard room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="1" | 1 Deluxe room booked 1 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="1" | Pass | - |
| 16 | To verify rule#16 | if condition #16 is true, the action will be action #16 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="2" VIP room available="0" Deluxe room available="0" Standard room available=">=2" | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="2" | 2 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="2" | Pass | - |
| 17 | To verify rule#17 | if condition #17 is true, the action will be action #17 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="1" VIP room available=">=1" | Number of VIP room booked="1" | 1 VIP room booked | Number of VIP room booked="1" | Pass | - |
| 18 | To verify rule#18 | if condition #18 is true, the action will be action #18 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="1" VIP room available="0" Deluxe room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="1" | 1 Deluxe room booked | Number of VIP room booked="0" Number of Deluxe room booked="1" | Pass | - |
| 19 | To verify rule#19 | if condition #19 is true, the action will be action #19 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="VIP" Exclusive Reward="N" Number of room to book="1" VIP room available="0" Deluxe room available="0" Standard room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="1" | 1 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 20 | To verify rule#20 | if condition #20 is true, the action will be action #20 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="VIP" Exclusive Reward="N" Number of room to book="4" | ERROR | - | ERROR | Pass | - |
| 21 | To verify rule#21 | if condition #21 is true, the action will be action #21 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="VIP" Exclusive Reward="N" Number of room to book="0" | ERROR | - | ERROR | Pass | - |
| 22 | To verify rule#22 | if condition #22 is true, the action will be action #22 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="2" Deluxe room available=">=2" | Number of Deluxe room booked="2" | 2 Deluxe room booked | Number of Deluxe room booked="2" | Pass | - |
| 23 | To verify rule#23 | if condition #23 is true, the action will be action #23 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="N" Number of room to book="2" Deluxe room available="0" Standard room available=">=2" | Number of Deluxe room booked="0" Number of Standard room booked="2" | 2 Standard room booked | Number of Deluxe room booked="0" Number of Standard room booked="2" | Pass | - |
| 24 | To verify rule#24 | if condition #24 is true, the action will be action #24 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="2" VIP room available=">=1" Deluxe room available="1" | Number of VIP room booked="1" Number of Deluxe room booked="1" Exclusive reward="redeemed" | 1 VIP room booked 1 Deluxe room booked | Number of VIP room booked="1" Number of Deluxe room booked="1" Exclusive reward="redeemed" | Pass | - |
| 25 | To verify rule#25 | if condition #25 is true, the action will be action #25 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="2" VIP room available=">=1" Deluxe room available="0" Standard room available=">=1" | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="1" Exclusive reward="redeemed" | 1 VIP room booked 1 Standard room booked | Number of VIP room booked="1" Number of Deluxe room booked="0" Number of Standard room booked="1" Exclusive reward="redeemed" | Pass | - |
| 26 | To verify rule#26 | if condition #26 is true, the action will be action #26 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="2" VIP room available="0" Deluxe room available="1" Standard room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="1" | 1 Deluxe room booked 1 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="1" Number of Standard room booked="1" | Pass | - |
| 27 | To verify rule#27 | if condition #27 is true, the action will be action #27 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="N" Number of room to book="2" Deluxe room available="1" Standard room available=">=1" | Number of Deluxe room booked="1" Number of Standard room booked="1" | 1 Deluxe room booked 1 Standard room booked | Number of Deluxe room booked="1" Number of Standard room booked="1" | Pass | - |
| 28 | To verify rule#28 | if condition #28 is true, the action will be action #28 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="1" Deluxe room available=">=1" | Number of Deluxe room booked="1" | 1 Deluxe room booked | Number of Deluxe room booked="1" | Pass | - |
| 29 | To verify rule#29 | if condition #29 is true, the action will be action #29 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="1" Deluxe room available="0" Standard room available=">=1" | Number of Deluxe room booked="0" Number of Standard room booked="1" | 1 Standard room booked | Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 30 | To verify rule#30 | if condition #30 is true, the action will be action #30 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="1" VIP room available=">=1" Deluxe room available="0" | Number of VIP room booked="1" Number of Deluxe room booked="0" Exclusive reward="redeemed" | 1 VIP room booked | Number of VIP room booked="1" Number of Deluxe room booked="0" Exclusive reward="redeemed" | Pass | - |
| 31 | To verify rule#31 | if condition #31 is true, the action will be action #31 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="Y" Number of room to book="1" VIP room available="0" Deluxe room available="0" Standard room available=">=1" | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="1" | 1 Standard room booked | Number of VIP room booked="0" Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 32 | To verify rule#32 | if condition #32 is true, the action will be action #32 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Normal" Exclusive Reward="N" Number of room to book="1" Deluxe room available="0" Standard room available=">=1" | Number of Deluxe room booked="0" Number of Standard room booked="1" | 1 Standard room booked | Number of Deluxe room booked="0" Number of Standard room booked="1" | Pass | - |
| 33 | To verify rule#33 | if condition #33 is true, the action will be action #33 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Normal" Exclusive Reward="Y" Number of room to book="3" | ERROR | - | ERROR | Pass | - |
| 34 | To verify rule#34 | if condition #34 is true, the action will be action #34 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Normal" Exclusive Reward="Y" Number of room to book="0" | ERROR | - | ERROR | Pass | - |
| 35 | To verify rule#35 | if condition #35 is true, the action will be action #35 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Normal" Exclusive Reward="N" Number of room to book="3" | ERROR | - | ERROR | Pass | - |
| 36 | To verify rule#36 | if condition #36 is true, the action will be action #36 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Normal" Exclusive Reward="N" Number of room to book="0" | ERROR | - | ERROR | Pass | - |
| 37 | To verify rule#37 | if condition #37 is true, the action will be action #37 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability | Member type="Non-member" Exclusive Reward="N" Number of room to book="1" Standard room available=">=1" | Number of Standard room booked="1" | 1 Standard room booked | Number of Standard room booked="1" | Pass | - |
| 38 | To verify rule#38 | if condition #38 is true, the action will be action #38 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Non-member" Exclusive Reward="N" Number of room to book="3" | ERROR | - | ERROR | Pass | - |
| 39 | To verify rule#39 | if condition #39 is true, the action will be action #39 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book | Member type="Non-member" Exclusive Reward="N" Number of room to book="0" | ERROR | - | ERROR | Pass | - |
| 40 | To verify rule#40 | if condition #40 is true, the action will be action #40 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability 5.Check waiting list | Member type="VIP" Exclusive Reward="N" Number of room to book="3" VIP room available="0" Deluxe room available="0" Standard room available="0" Waiting List="N" | Waiting List="Y" | 1 Waiting List placed | Waiting List="Y" | Pass | - |
| 41 | To verify rule#41 | if condition #41 is true, the action will be action #41 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability 5.Check waiting list | Member type="Normal" Exclusive Reward="Y" Number of room to book="2" VIP room available="0" Deluxe room available="0" Standard room available="0" Waiting List="N" | Waiting List="Y" | 1 Waiting List placed | Waiting List="Y" | Pass | - |
| 42 | To verify rule#42 | if condition #42 is true, the action will be action #42 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability 5.Check waiting list | Member type="Normal" Exclusive Reward="N" Number of room to book="2" VIP room available="0" Deluxe room available="0" Standard room available="0" Waiting List="N" | Waiting List="Y" | 1 Waiting List placed | Waiting List="Y" | Pass | - |
| 43 | To verify rule#43 | if condition #43 is true, the action will be action #43 | 1.Indicate the member type 2.Check exclusive reward of member 3.Ask for number of room to book  4.Check room availability 5.Check waiting list | Member type="Normal" Exclusive Reward="N" Number of room to book="1" Standard room available="0" Waiting List="N" | Waiting List="Y" | 1 Waiting List placed | Waiting List="Y" | Pass | - |
| 44 | To verify rule#44 | if condition #44 is true, the action will be action #44 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="VIP" Exclusive Reward="N" Cancel booking="Y" Waiting List="Y" | Waiting List="N" | 1 Waiting List removed | Waiting List="N" | Pass | - |
| 45 | To verify rule#45 | if condition #45 is true, the action will be action #45 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="VIP" Exclusive Reward="N" Cancel booking="Y" Waiting List="N" | User Cancelled="Y" | 1 User Cancelled | User Cancelled="Y" | Pass | - |
| 46 | To verify rule#46 | if condition #46 is true, the action will be action #46 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Normal" Exclusive Reward="Y" Cancel booking="Y" Waiting List="Y" | Waiting List="N" | 1 Waiting List removed | Waiting List="N" | Pass | - |
| 47 | To verify rule#47 | if condition #47 is true, the action will be action #47 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Normal" Exclusive Reward="Y" Cancel booking="Y" Waiting List="N" | User Cancelled="Y" | 1 User Cancelled | User Cancelled="Y" | Pass | - |
| 48 | To verify rule#48 | if condition #48 is true, the action will be action #48 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Normal" Exclusive Reward="N" Cancel booking="Y" Waiting List="Y" | Waiting List="N" | 1 Waiting List removed | Waiting List="N" | Pass |  |
| 49 | To verify rule#49 | if condition #49 is true, the action will be action #49 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Normal" Exclusive Reward="N" Cancel booking="Y" Waiting List="N" | User Cancelled="Y" | 1 User Cancelled | User Cancelled="Y" | Pass |  |
| 50 | To verify rule#50 | if condition #50 is true, the action will be action #50 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Non-member" Exclusive Reward="N" Cancel booking="Y" Waiting List="Y" | Waiting List="N" | 1 Waiting List removed | Waiting List="N" | Pass | - |
| 51 | To verify rule#51 | if condition #51 is true, the action will be action #51 | 1.Indicate the member type 2.Check exclusive reward of member 3.Cancel booking for member 4.Check waiting list | Member type="Non-member" Exclusive Reward="N" Cancel booking="Y" Waiting List="N" | User Cancelled="Y" | 1 User Cancelled | User Cancelled="Y" | Pass | - |

# Class Diagram

A screenshot of a computer

Description automatically generated

Application Code

## HotelBookingSystem/main

### Booking.java

package utar;

import java.util.HashMap;

import java.util.Map;

public class Booking {

private Room room;

private Map<User, Map<String, Integer>> bookings; // Modified to store user, room type, and number of rooms booked

private int vip;

private int deluxe;

private int standard;

private WaitingList waitingList;

private Printer printer;

public Map<User, Map<String, Integer>> getBookings() {

return bookings;

}

public Booking() {

waitingList = new WaitingList();

bookings = new HashMap<>(); // Initialize the bookings HashMap

printer = new Printer();

}

public Booking(Room room, WaitingList waitingList,Printer printer) {

this.room = room;

this.printer = printer;

this.waitingList = waitingList;

bookings = new HashMap<>(); // Initialize the bookings HashMap

}

public void setBooking(User user, int numOfRooms) {

String memberType = user.getMemberType();

boolean exclReward = user.hasExclReward();

if (numOfRooms <= 0)

throw new IllegalArgumentException("Invalid number of Booking");

if (memberType.equals("VIP")) {

if (numOfRooms > 3)

throw new IllegalArgumentException("Invalid number of Booking");

if (room.getAvailableRooms("VIP") >= numOfRooms) {

room.bookRoom("VIP", numOfRooms);

printer.printInfo(user.getName(), user.getMemberType(), "VIP", numOfRooms);

// Store user, room type, and number of rooms booked in the bookings HashMap

Map<String, Integer> bookingDetails = new HashMap<>();

bookingDetails.put("VIP", numOfRooms);

bookings.put(user, bookingDetails);

return;

} else {

int remainingRooms = numOfRooms;

int vipRoomsAvailable = room.getAvailableRooms("VIP");

int vipRoomsBooked = Math.*min*(3, vipRoomsAvailable);

room.bookRoom("VIP", vipRoomsBooked);

Map<String, Integer> bookingDetails = new HashMap<>();

bookingDetails.put("VIP", vipRoomsBooked);

bookings.put(user, bookingDetails);

printer.printInfo(user.getName(), user.getMemberType(), "VIP", vipRoomsBooked);

remainingRooms -= vipRoomsBooked;

if (remainingRooms > 0) {

int deluxeRooms = Math.*min*(remainingRooms, room.getAvailableRooms("Deluxe"));

room.bookRoom("Deluxe", deluxeRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Deluxe", deluxeRooms);

bookingDetails.put("Deluxe", deluxeRooms);

bookings.put(user, bookingDetails);

remainingRooms -= deluxeRooms;

if (remainingRooms > 0 && room.getAvailableRooms("Standard") >=remainingRooms) {

room.bookRoom("Standard", remainingRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Standard", remainingRooms);

bookingDetails.put("Standard", remainingRooms);

bookings.put(user, bookingDetails);

return;

} else if (remainingRooms > 0) {

waitingList.addWaiting(user, "VIP");

room.cancelBooking("VIP", vipRoomsBooked);

room.cancelBooking("Deluxe", deluxeRooms);

System.***out***.println("Requested number of rooms not available. Adding to VIP waiting list.");

return;

}

return;

}

}

}

if (memberType.equals("Normal")) {

if (numOfRooms > 2)

throw new IllegalArgumentException("Invalid number of Booking");

int remainingRooms = numOfRooms;

if (exclReward && room.getAvailableRooms("VIP") >= 1) {

room.bookRoom("VIP", 1);

Map<String, Integer> bookingDetails = new HashMap<>();

bookingDetails.put("VIP", 1);

bookings.put(user, bookingDetails);

printer.printInfo(user.getName(), user.getMemberType(), "VIP", 1);

user.setExclReward(false);

remainingRooms--;

if (remainingRooms > 0) {

int deluxeRooms = Math.*min*(remainingRooms, room.getAvailableRooms("Deluxe"));

room.bookRoom("Deluxe", deluxeRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Deluxe", deluxeRooms);

bookingDetails.put("Deluxe", deluxeRooms);

bookings.put(user, bookingDetails);

remainingRooms -= deluxeRooms;

if (remainingRooms > 0 && room.getAvailableRooms("Standard")>= remainingRooms) {

room.bookRoom("Standard", remainingRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Standard", remainingRooms);

bookingDetails.put("Standard", remainingRooms);

bookings.put(user, bookingDetails);

return;

} else if (remainingRooms > 0) {

waitingList.addWaiting(user, "VIP");

room.cancelBooking("VIP", 1);

room.cancelBooking("Deluxe", deluxeRooms);

System.***out***.println("Requested number of rooms not available. Adding to VIP waiting list.");

return;

}

return;

}

return;

} else {

int deluxeRoomsAvailable = room.getAvailableRooms("Deluxe");

int deluxeRooms = Math.*min*(deluxeRoomsAvailable, 3);

room.bookRoom("Deluxe", deluxeRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Deluxe", deluxeRooms);

Map<String, Integer> bookingDetails = new HashMap<>();

bookingDetails.put("Deluxe", deluxeRooms);

bookings.put(user, bookingDetails);

remainingRooms -= deluxeRooms;

if (remainingRooms > 0 && room.getAvailableRooms("Standard")>= remainingRooms) {

room.bookRoom("Standard", remainingRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Standard", remainingRooms);

bookingDetails.put("Standard", remainingRooms);

bookings.put(user, bookingDetails);

return;

} else if (remainingRooms > 0){

waitingList.addWaiting(user, "Normal");

room.cancelBooking("Deluxe", deluxeRooms);

System.***out***.println("No rooms available. Adding to normal member waiting list.");

return;

}

return;

}

}

else { // Non-member

if (numOfRooms > 1)

throw new IllegalArgumentException("Invalid number of Booking");

if (room.getAvailableRooms("Standard")>= numOfRooms) {

room.bookRoom("Standard", numOfRooms);

printer.printInfo(user.getName(), user.getMemberType(), "Standard", numOfRooms);

Map<String, Integer> bookingDetails = new HashMap<>();

bookingDetails.put("Standard", numOfRooms);

bookings.put(user, bookingDetails);

return;

} else if (room.getAvailableRooms("Standard") >= numOfRooms == false){

waitingList.addWaiting(user, "Non-member");

System.***out***.println("Standard Room fully booked. Adding to non-member waiting list.");

return;

}

}

}

public void cancelBooking(User user) {

String memberType = user.getMemberType();

if (bookings.containsKey(user)) {

Map<String, Integer> bookingDetails = bookings.get(user);

for (Map.Entry<String, Integer> entry : bookingDetails.entrySet()) {

String roomType = entry.getKey();

int numOfRooms = entry.getValue();

room.cancelBooking(roomType, numOfRooms);

}

bookings.remove(user);

System.***out***.println("Booking canceled for user: " + user.getName());

System.***out***.print(memberType);

if (memberType.equals("VIP")) {

waitingList.removeWaiting(user, "VIP");

} else if (memberType.equals("Normal")) {

waitingList.removeWaiting(user, "Normal");

}

}

else if(!bookings.containsKey(user) && memberType.equals("Non-member"))

{

waitingList.removeWaiting(user, "Non-member");

}

else {

System.***out***.println("No booking found for user: " + user.getName());

}

}

}

### Printer.java

package utar;

public class Printer {

public void printInfo(String name, String member\_type, String room\_type, int numOfRoom) {

System.***out***.println("Booking Information:");

System.***out***.println("User: " + name + ", Member Type: " + member\_type + ", Room Type: " + room\_type +", Number of Room is: "+ numOfRoom);

}

}

### Room.java

package utar;

public class Room {

private int vip;

private int deluxe;

private int standard;

public Room(int vip, int deluxe, int standard) {

this.vip = vip;

this.deluxe = deluxe;

this.standard = standard;

}

public boolean checkRoom(String roomType, int numOfRooms) {

switch (roomType) {

case "VIP":

return vip >= numOfRooms;

case "Deluxe":

return deluxe >= numOfRooms;

case "Standard":

return standard >= numOfRooms;

default:

return false;

}

}

public void bookRoom(String roomType, int numOfRooms) {

switch (roomType) {

case "VIP":

vip -= numOfRooms;

break;

case "Deluxe":

deluxe -= numOfRooms;

break;

case "Standard":

standard -= numOfRooms;

break;

}

}

public void cancelBooking(String roomType, int numOfRooms) {

switch (roomType) {

case "VIP":

vip += numOfRooms;

break;

case "Deluxe":

deluxe += numOfRooms;

break;

case "Standard":

standard += numOfRooms;

break;

}

}

public int getAvailableRooms(String roomType) {

switch (roomType) {

case "VIP":

return vip;

case "Deluxe":

return deluxe;

case "Standard":

return standard;

default:

return 0;

}

}

}

### User.Java

package utar;

import static org.mockito.Mockito.*verify*;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

import java.util.Objects;

import java.util.ArrayList;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.Map;

class User {

private String name;

private String memberType;

private boolean exclReward;

public User(String name, String memberType, boolean exclReward) {

if (name == null || memberType == null) {

throw new IllegalArgumentException("Name and member type cannot be null");

}

// Check for empty or blank name

if (name.trim().isEmpty()) {

throw new IllegalArgumentException("Name cannot be empty or blank");

}

// Check for valid member type

if (!memberType.equals("VIP") && !memberType.equals("Normal") && !memberType.equals("Non-member")) {

throw new IllegalArgumentException("Invalid member type: " + memberType);

}

// Check for VIP reward

if(memberType.equals("VIP") && exclReward == true)

throw new IllegalArgumentException("No exlusive reward for " + memberType);

// Check for Non-member reward

if(memberType.equals("Non-member") && exclReward == true)

throw new IllegalArgumentException("No exlusive reward for " + memberType);

// Check for valid boolean value for exclReward

if (exclReward != true && exclReward != false) {

throw new IllegalArgumentException("Invalid boolean value for exclReward: " + exclReward);

}

this.name = name;

this.memberType = memberType;

this.exclReward = exclReward;

}

public String getName() {

return name;

}

public String getMemberType() {

return memberType;

}

public boolean hasExclReward() {

return exclReward;

}

public void setExclReward(boolean b) {

exclReward = b;

}

// Inside User class

*@Override*

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj == null || getClass() != obj.getClass()) {

return false;

}

User other = (User) obj;

return Objects.*equals*(name, other.name) && Objects.*equals*(memberType, other.memberType) && Objects.*equals*(exclReward, other.exclReward) ;

}

}

### WaitingList.Java

package utar;

import java.util.ArrayList;

public class WaitingList {

private ArrayList<User> vipWaitingList;

private ArrayList<User> memberWaitingList;

private ArrayList<User> normalWaitingList;

public WaitingList() {

vipWaitingList = new ArrayList<>();

memberWaitingList = new ArrayList<>();

normalWaitingList = new ArrayList<>();

}

public void addWaiting(User user, String memberType) {

if (user == null || memberType == null) {

throw new IllegalArgumentException("User and member type cannot be null");

}

if(!memberType.equals("VIP") && !memberType.equals("Non-member") && !memberType.equals("Normal"))

throw new IllegalArgumentException("Invalid Member Type");

if(user.getMemberType() != memberType)

throw new IllegalArgumentException("Mismatch Member Type");

switch (memberType) {

case "VIP":

vipWaitingList.add(user);

break;

case "Normal":

memberWaitingList.add(user);

break;

default:

normalWaitingList.add(user);

break;

}

}

public ArrayList<User> getWaiting(String memberType) {

switch (memberType) {

case "VIP":

return vipWaitingList;

case "Normal":

return memberWaitingList;

default:

return normalWaitingList;

}

}

public void removeWaiting(User user, String memberType) {

if (user == null || memberType == null) {

throw new IllegalArgumentException("User and member type cannot be null");

}

if(!memberType.equals("VIP") && !memberType.equals("Non-member") && !memberType.equals("Normal"))

throw new IllegalArgumentException("Invalid Member Type");

if(user.getMemberType() != memberType)

throw new IllegalArgumentException("Mismatch Member Type");

switch (memberType) {

case "VIP":

if (!vipWaitingList.contains(user)) {

throw new IllegalArgumentException("User is not in the VIP waiting list");

}

vipWaitingList.remove(user);

break;

case "Normal":

if (!memberWaitingList.contains(user)) {

throw new IllegalArgumentException("User is not in the normal member waiting list");

}

memberWaitingList.remove(user);

break;

default:

if (!normalWaitingList.contains(user)) {

throw new IllegalArgumentException("User is not in the waiting list");

}

normalWaitingList.remove(user);

break;

}

}

}

## HotelBookingSystem/test

### AddToWaitingListTest.java

package utar;

import static org.junit.Assert.*assertArrayEquals*;

import static org.mockito.Mockito.*mock*;

import static org.mockito.Mockito.*when*;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import junitparams.JUnitParamsRunner;

import junitparams.Parameters;

*@RunWith*(JUnitParamsRunner.class)

public class AddToWaitingListTest {

WaitingList wl;

*@Before*

public void setUp() {

wl = new WaitingList();

}

private Object[] ParamForCheckingWaitingListNonMember() {

User[] normalWaitingList = { new User("Akao", "Non-member", false) };

return new Object[] { new Object[] { new User("Akao", "Non-member", false), 1, 0, normalWaitingList }, // test 43

};

}

*@Test*()

*@Parameters*(method = "ParamForCheckingWaitingListNonMember")

public void testWaitingListNonMember(User user, int bookRoomNumber, int availableOfAllRoom, User[] er) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("Non-member").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

private Object[] ParamForCheckingWaitingListVIP() {

User[] vipWaitingList = { new User("Ali", "VIP", false) };

return new Object[] {

new Object[] { new User("Ali", "VIP", false), 3, 0, vipWaitingList },//test40

};

}

*@Test*()

*@Parameters*(method = " ParamForCheckingWaitingListVIP")

public void testWaitingListVIP(User user, int bookRoomNumber, int availableOfAllRoom, User[] er) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("VIP").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

private Object[] ParamForCheckingWaitingListNormal() {

User[] memberWaitingList = { new User("Abu", "Normal", false) };

User[] memberWaitingList2 = { new User("Abu", "Normal", true) };

return new Object[] {

new Object[] { new User("Abu", "Normal", false), 2, memberWaitingList, 0 }, // test42

new Object[] { new User("Abu", "Normal", true), 2, memberWaitingList2, 0 },// test41

};

}

*@Test*()

*@Parameters*(method = "ParamForCheckingWaitingListNormal")

public void testWaitingListNormal(User user, int bookRoomNumber, User[] er, int availableOfAllRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("Normal").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

}

### CancelBookingTest.java

package utar;

import static org.junit.Assert.*assertArrayEquals*;

import static org.mockito.Mockito.*mock*;

import static org.mockito.Mockito.*when*;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import junitparams.JUnitParamsRunner;

import junitparams.Parameters;

*@RunWith*(JUnitParamsRunner.class)

public class AddToWaitingListTest {

WaitingList wl;

*@Before*

public void setUp() {

wl = new WaitingList();

}

private Object[] ParamForCheckingWaitingListNonMember() {

User[] normalWaitingList = { new User("Akao", "Non-member", false) };

return new Object[] { new Object[] { new User("Akao", "Non-member", false), 1, 0, normalWaitingList }, // test 43

};

}

*@Test*()

*@Parameters*(method = "ParamForCheckingWaitingListNonMember")

public void testWaitingListNonMember(User user, int bookRoomNumber, int availableOfAllRoom, User[] er) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("Non-member").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

private Object[] ParamForCheckingWaitingListVIP() {

User[] vipWaitingList = { new User("Ali", "VIP", false) };

return new Object[] {

new Object[] { new User("Ali", "VIP", false), 3, 0, vipWaitingList },//test40

};

}

*@Test*()

*@Parameters*(method = " ParamForCheckingWaitingListVIP")

public void testWaitingListVIP(User user, int bookRoomNumber, int availableOfAllRoom, User[] er) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("VIP").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

private Object[] ParamForCheckingWaitingListNormal() {

User[] memberWaitingList = { new User("Abu", "Normal", false) };

User[] memberWaitingList2 = { new User("Abu", "Normal", true) };

return new Object[] {

new Object[] { new User("Abu", "Normal", false), 2, memberWaitingList, 0 }, // test42

new Object[] { new User("Abu", "Normal", true), 2, memberWaitingList2, 0 },// test41

};

}

*@Test*()

*@Parameters*(method = "ParamForCheckingWaitingListNormal")

public void testWaitingListNormal(User user, int bookRoomNumber, User[] er, int availableOfAllRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableOfAllRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableOfAllRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

User[] actualResult = wl.getWaiting("Normal").toArray(new User[0]);

*assertArrayEquals*(er, actualResult);

}

}

### SetBookingTest.java

package utar;

import static org.junit.Assert.*assertEquals*;

import static org.junit.Assert.*assertTrue*;

import static org.mockito.Mockito.*mock*;

import static org.mockito.Mockito.*verify*;

import static org.mockito.Mockito.*when*;

import java.util.Map;

import org.junit.Before;

import org.junit.Test;

import org.junit.runner.RunWith;

import junitparams.JUnitParamsRunner;

import junitparams.Parameters;

*@RunWith*(JUnitParamsRunner.class)

public class SetBookingTest {

WaitingList wl;

*@Before*

public void setUp() {

wl = new WaitingList();

}

private Object[] ParamForSetBookingVIP1() {

return new Object[] { new Object[] { new User("Ali", "VIP", false), 3, 3 }, // test1

new Object[] { new User("Ali", "VIP", false), 2, 2 }, // test11

new Object[] { new User("Ali", "VIP", false), 1, 1 }, // test17

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingVIP1")

public void testSetBookingVIP1(User user, int availableVIPRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

}

private Object[] ParamForSetBookingNonMember1() {

return new Object[] { new Object[] { new User("Akao", "Non-member", false), 1, 1 } // test37

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNonMember1")

public void testSetBookingNonMember1(User user, int availableStandardRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal1() {

return new Object[] {

new Object[] { new User("Abu", "Normal", true), 0, 1, 1 }, // test28

new Object[] { new User("Abu", "Normal", true), 0, 2, 2 }, // test22

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal1")

public void testSetBookingNormal1(User user, int availableVIPRoom, int availableDeluxeRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

Printer printer = *mock*(Printer.class);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*assertEquals*(user.hasExclReward(), true);

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

}

private Object[] ParamForSetBookingNormal2() {

return new Object[] {

new Object[] { new User("Abu", "Normal", false), 0, 1, 1, 2 }, // test27

new Object[] { new User("Abu", "Normal", false), 0, 0, 1, 1 }, // test32

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal2")

public void testSetBookingNormal2(User user, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal3() {

return new Object[] { new Object[] { new User("Abu", "Normal", true), 0, 1, 1, 2 }, // test26

new Object[] { new User("Abu", "Normal", true), 0, 0, 1, 1 }, // test31

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal3")

public void testSetBookingNormal3(User user, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

*assertEquals*(user.hasExclReward(), true);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingVIP2() {

return new Object[] { new Object[] { new User("Ali", "VIP", false), 2, 1, 3 }, // test2

new Object[] { new User("Ali", "VIP", false), 1, 2, 3 }, // test4

new Object[] { new User("Ali", "VIP", false), 0, 3, 3 }, // test7

new Object[] { new User("Ali", "VIP", false), 1, 1, 2 }, // test12

new Object[] { new User("Ali", "VIP", false), 0, 1, 1 }, // test18

new Object[] { new User("Ali", "VIP", false), 0, 2, 2 }, // test14

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingVIP2")

public void testSetBookingVIP2(User user, int availableVIPRoom, int availableDeluxeRoom, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

}

private Object[] ParamForSetBookingVIP3() {

return new Object[] { new Object[] { new User("Ali", "VIP", false), 3, 2, 0, 1 }, // test3

new Object[] { new User("Ali", "VIP", false), 3, 1, 0, 2 }, // test5

new Object[] { new User("Ali", "VIP", false), 2, 1, 0, 1 }, // test13

new Object[] { new User("Ali", "VIP", false), 3, 0, 0, 3 }, // test8

new Object[] { new User("Ali", "VIP", false), 3, 1, 1, 1 }, // test6

new Object[] { new User("Ali", "VIP", false), 3, 0, 2, 1 }, // test9

new Object[] { new User("Ali", "VIP", false), 3, 0, 1, 2 }, // test10

new Object[] { new User("Ali", "VIP", false), 2, 0, 1, 1 }, // test15

new Object[] { new User("Ali", "VIP", false), 2, 0, 0, 2 }, // test16

new Object[] { new User("Ali", "VIP", false), 1, 0, 0, 1 }, // test19

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingVIP3")

public void testSetBookingVIP3(User user, int bookRoomNumber, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

// Verify the booking behavior based on the availability of rooms

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal4() {

return new Object[] { new Object[] { new User("Abu", "Normal", true), 2, 1, 1 }, // test24

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal4")

public void testSetBookingNormal4(User user, int bookRoomNumber, int availableVIPRoom, int availableDeluxeRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

*assertEquals*(user.hasExclReward(), false);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

}

private Object[] ParamForSetBookingNormal5() {

return new Object[] { new Object[] { new User("Abu", "Normal", true), 2, 1, 0, 1 }, // test25

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal5")

public void testSetBookingNormal5(User user, int bookRoomNumber, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

*assertEquals*(user.hasExclReward(), false);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal6() {

return new Object[] { new Object[] { new User("Abu", "Normal", false), 2, 0, 0, 2 }, // test23

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal6")

public void testSetBookingNormal6(User user, int bookRoomNumber, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal7() {

return new Object[] {

new Object[] { new User("Abu", "Normal", true), 1, 0, 0, 1 }, // test29

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal7")

public void testSetBookingNormal7(User user, int bookRoomNumber, int availableVIPRoom, int availableDeluxeRoom,

int availableStandardRoom) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

*when*(rm.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRoom);

*when*(rm.getAvailableRooms("Standard")).thenReturn(availableStandardRoom);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

*assertEquals*(user.hasExclReward(), true);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("Deluxe", availableDeluxeRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Deluxe", availableDeluxeRoom);

*verify*(rm).bookRoom("Standard", availableStandardRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "Standard", availableStandardRoom);

}

private Object[] ParamForSetBookingNormal8() {

return new Object[] { new Object[] { new User("Abu", "Normal", true), 1, 1 }, // test30

};

}

*@Test*

*@Parameters*(method = "ParamForSetBookingNormal8")

public void testSetBookingNormal8(User user, int bookRoomNumber, int availableVIPRoom) {

Room rm = *mock*(Room.class);

*when*(rm.getAvailableRooms("VIP")).thenReturn(availableVIPRoom);

Printer printer = *mock*(Printer.class);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

*assertEquals*(user.hasExclReward(), false);

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(user));

*verify*(rm).bookRoom("VIP", availableVIPRoom);

*verify*(printer).printInfo(user.getName(), user.getMemberType(), "VIP", availableVIPRoom);

}

private Object[] ParamForSetBookingExceptionVIP() {

return new Object[] { new Object[] { new User("Ali", "VIP", false), 4 }, // test20

new Object[] { new User("Ali", "VIP", false), 0 }, // test21

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamForSetBookingExceptionVIP")

public void testSetBookingExceptionVIP(User user, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

// Verify the booking behavior based on the availability of rooms

}

private Object[] ParamForSetBookingExceptionNormal() {

return new Object[] { new Object[] { new User("Abu", "Normal", true), 3 }, // test 33

new Object[] { new User("Abu", "Normal", true), 0 }, // test 34

new Object[] { new User("Abu", "Normal", false), 3 }, // test 35

new Object[] { new User("Abu", "Normal", false), 0 },// test 36

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamForSetBookingExceptionNormal")

public void testSetBookingExceptionNormal(User user, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

// Verify the booking behavior based on the availability of rooms

}

private Object[] ParamForSetBookingExceptionNonMember() {

return new Object[] { new Object[] { new User("Akao", "Non-member", false), 2 }, // test38

new Object[] { new User("Akao", "Non-member", false), 0 }, // test39

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamForSetBookingExceptionNonMember")

public void testSetBookingExceptionNonMember(User user, int bookRoomNumber) {

Room rm = *mock*(Room.class);

Printer printer = *mock*(Printer.class);

Booking booking = new Booking(rm, wl, printer);

booking.setBooking(user, bookRoomNumber);

// Verify the booking behavior based on the availability of rooms

}

}

### TestSuite.java

package utar;

import org.junit.runner.RunWith;

import org.junit.runners.Suite;

import org.junit.runners.Suite.SuiteClasses;

*@RunWith*(value = Suite.class)

*@SuiteClasses*(value = {AddToWaitingListTest.class, CancelBookingTest.class, SetBookingTest.class,

TextFileTest.class, UserTesting.class,WaitingListTest.class })

public class TestSuite {

}

### TextFileTest.java

package utar;

import org.junit.Before;

import org.junit.BeforeClass;

import org.junit.Test;

import static org.junit.Assert.\*;

import static org.mockito.Mockito.\*;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import java.util.Map;

public class TextFileTest {

private static User *user*;

private static Room *rm*;

private static Printer *printer*;

private static int *bookRoomNumber*;

private Booking booking;

private static WaitingList *waitingList*;

private List<User> users = new ArrayList<>();

private List<Integer> numOfRoomBook = new ArrayList<>();

private List<Integer> availableVIPRooms = new ArrayList<>();

private List<Integer> availableDeluxeRooms = new ArrayList<>();

private List<Integer> availableStandardRooms = new ArrayList<>();

*@BeforeClass*

public static void setUp()

{

*waitingList* = new WaitingList();

}

*@Before*

public void setup() throws IOException {

parseInputFile("validInput.txt");

}

private void parseInputFile(String filename) throws IOException {

BufferedReader br = new BufferedReader(new FileReader(filename));

String line;

while ((line = br.readLine()) != null) {

String[] params = line.split(",");

String name = params[0];

String memberType = params[1];

boolean exclReward = Boolean.*parseBoolean*(params[2]);

*bookRoomNumber* = Integer.*parseInt*(params[3].trim());

int availableVIPRoom = Integer.*parseInt*(params[4].trim());

int availableDeluxeRoom = Integer.*parseInt*(params[5].trim());

int availableStandardRoom = Integer.*parseInt*(params[6].trim());

*user* = new User(name, memberType, exclReward);

users.add(*user*);

numOfRoomBook.add(*bookRoomNumber*);

availableVIPRooms.add(availableVIPRoom);

availableDeluxeRooms.add(availableDeluxeRoom);

availableStandardRooms.add(availableStandardRoom);

}

br.close();

}

*@Test* //The room is sufficient to book, The room for VIP is consider as 0 for the Normal Member Booking.

public void testBookingWithValidInput() throws IOException{

*rm* = *mock*(Room.class);

*printer* = *mock*(Printer.class);

booking = new Booking(*rm*, *waitingList*,*printer*);

for(int i =0; i<users.size();i ++)

{

if (*user*.getMemberType() == "VIP")

{

*when*(*rm*.getAvailableRooms("VIP")).thenReturn(availableVIPRooms.get(i));

booking.setBooking(users.get(i), numOfRoomBook.get(i));

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(*user*));

*verify*(*rm*).bookRoom("VIP", availableVIPRooms.get(i));

*verify*(*printer*).printInfo(*user*.getName(), *user*.getMemberType(), "VIP", availableVIPRooms.get(i));

}

else if (*user*.getMemberType() =="Normal")

{

*when*(*rm*.getAvailableRooms("VIP")).thenReturn(availableVIPRooms.get(i));

*when*(*rm*.getAvailableRooms("Deluxe")).thenReturn(availableDeluxeRooms.get(i));

booking.setBooking(users.get(i), numOfRoomBook.get(i));

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(*user*));

*assertEquals*(*user*.hasExclReward(), true);

*verify*(*rm*).bookRoom("Deluxe", numOfRoomBook.get(i));

*verify*(*printer*).printInfo(*user*.getName(), *user*.getMemberType(), "Deluxe", availableDeluxeRooms.get(i));

}

else if (*user*.getMemberType() == "Non-member")

{

*when*(*rm*.getAvailableRooms("Standard")).thenReturn(availableStandardRooms.get(i));

booking.setBooking(users.get(i), numOfRoomBook.get(i));

Map<User, Map<String, Integer>> bookings = booking.getBookings();

*assertTrue*(bookings.containsKey(*user*));

*verify*(*rm*).bookRoom("Standard", numOfRoomBook.get(i));

*verify*(*printer*).printInfo(*user*.getName(), *user*.getMemberType(), "Standard", availableStandardRooms.get(i));

}

}

}

*@Test*(expected = IllegalArgumentException.class)

public void testBookingWithInvalidInput() throws IOException {

try (BufferedReader br = new BufferedReader(new FileReader("invalidInput.txt"))) {

String line;

while ((line = br.readLine()) != null) {

// Parse input parameters from each line

String[] params = line.split(",");

String name = params[0];

String memberType = params[1];

boolean exclReward = Boolean.*parseBoolean*(params[2]);

int numOfRooms = Integer.*parseInt*(params[3]);

// Create a User object and invoke the method under test with the input parameters

*rm* = *mock*(Room.class);

*printer* = *mock*(Printer.class);

booking = new Booking(*rm*, *waitingList*,*printer*);

booking.setBooking(new User(name, memberType, exclReward), numOfRooms);

}

}

}

}

### UserTesting.java

package utar;

import static org.junit.Assert.*assertEquals*;

import static org.junit.Assert.*assertFalse*;

import static org.junit.Assert.*assertTrue*;

import org.junit.Test;

import org.junit.runner.RunWith;

import junitparams.JUnitParamsRunner;

import junitparams.Parameters;

*@RunWith*(JUnitParamsRunner.class)

public class UserTesting {

*@Test*

public void testValidUserCreation() {

User user = new User("John", "VIP", false);

*assertEquals*("John", user.getName());

*assertEquals*("VIP", user.getMemberType());

*assertFalse*(user.hasExclReward());

}

private Object[] ParamForInvalidUser() {

return new Object[] {

new Object[] {null, "VIP", true},// test 40

new Object[] {"John", null, true},

new Object[] {"", "VIP", true},

new Object[] {"John", "Guest", true}

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamForInvalidUser")

public void testExceptionForUser(String name, String userType, boolean exclusiveReward) {

new User(name,userType, exclusiveReward);

}

}

### WaitingListTest.java

package utar;

import static org.junit.Assert.*assertFalse*;

import static org.junit.Assert.*assertTrue*;

import org.junit.Test;

import org.junit.runner.RunWith;

import junitparams.JUnitParamsRunner;

import junitparams.Parameters;

*@RunWith*(JUnitParamsRunner.class)

public class WaitingListTest {

*@Test*

public void testAddWaiting() {

WaitingList waitingList = new WaitingList();

User user = new User("John", "VIP", false);

waitingList.addWaiting(user, "VIP");

*assertTrue*(waitingList.getWaiting("VIP").contains(user));

}

*@Test*

public void testRemoveWaiting() {

WaitingList waitingList = new WaitingList();

User user = new User("John", "VIP", false);

waitingList.addWaiting(user, "VIP");

waitingList.removeWaiting(user, "VIP");

*assertFalse*(waitingList.getWaiting("VIP").contains(user));

}

private Object[] ParamsForAddWaitingException() {

return new Object[] {

new Object[] {new User("John", "VIP", false), "zz"},

new Object[] {new User("John", "VIP", false), "Normal"},

new Object[] {null, "VIP"}

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamsForAddWaitingException")

public void testAddWaitingException(User user, String userType) {

WaitingList waitingList = new WaitingList();

waitingList.addWaiting(user, userType);

}

private Object[] ParamsForRemoveWaitingException() {

return new Object[] {

new Object[] {new User("John", "VIP", false), null},

new Object[] {new User("John", "VIP", false), "Normal"},

new Object[] {null, "VIP"}

};

}

*@Test*(expected = IllegalArgumentException.class)

*@Parameters*(method = "ParamsForRemoveWaitingException")

public void testRemoveWaitingWithNullMemberType(User user, String userType) {

WaitingList waitingList = new WaitingList();

waitingList.removeWaiting(user, null);

}

}