MARKING REPORT

**Group number: \_12\_\_\_**

|  |  |
| --- | --- |
| Name | Student ID |
| 1. KUIK RUI LUAN | 2103790 |
| 2. LEE WAI LUN | 2103328 |
| 3. LOW SI QING | 2104872 |
| 4. VINCY LIM WEI JUN | 2105978 |

**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 Content

[Part A Test Plan 3](#_Toc164881393)

[Part B Test Cases Design 3](#_Toc164881394)

[Decision Tables 3](#_Toc164881395)

[Test Cases 3](#_Toc164881396)

[Assumptions 3](#_Toc164881397)

[Class Diagram 4](#_Toc164881398)

[Part C 5](#_Toc164881399)

[Application Code 5](#_Toc164881400)

[Test Code 12](#_Toc164881401)

[Project folder 35](#_Toc164881402)

# **Part A Test Plan**

Refer to attached *Test Plan.doc*

# **Part B Test Cases Design**

## Decision Tables

Referto attached *Decision Table.xlsx*

## Test Cases

Refer to attached *Test Case.xlsx*

## Assumptions

The test plan includes following assumptions:

1. One exclusive reward only can be redeemed for one VIP room.
2. Reward used to book VIP room will not be returned after cancellation.

## Class Diagram

A black screen with white text

Description automatically generated

# **Part C**

## Application Code

|  |
| --- |
| User |
| package hotel.booking.system;  public class User {    private String name;  private String member\_type;  private boolean excl\_reward;    //constructor  public User(String name, String member\_type, boolean excl\_reward){  if (name == null || name.isEmpty()) {  throw new IllegalArgumentException("Empty name");  }  if (member\_type == null || member\_type.isEmpty()) {  throw new IllegalArgumentException("Empty name");  }  if (!member\_type.equals("VIP") && !member\_type.equals("member") && !member\_type.equals("normal")) {  throw new IllegalArgumentException("Invalid member type");  }  if (member\_type.equals("normal") && excl\_reward == true){  throw new IllegalArgumentException("Non-member are not entitled to exclusive reward");  }  if(!(excl\_reward==true||excl\_reward==false)) {  throw new IllegalArgumentException("Invalid exclusive reward status");  }    this.name = name;  this.member\_type = member\_type;  this.excl\_reward = excl\_reward;  }    public String getName(){  return name;  }    public String get\_member\_type(){  return member\_type;  }    public boolean get\_excl\_reward(){  return excl\_reward;  }    public void set\_excl\_reward(boolean excl\_reward){  if (member\_type.equals("normal") && excl\_reward == true)  throw new IllegalArgumentException("Non-member are not entitled to exclusive reward");  if(!(excl\_reward==true||excl\_reward==false)) {  throw new IllegalArgumentException("Invalid exclusive reward status");  }  this.excl\_reward = excl\_reward;  }  } |

|  |
| --- |
| WaitingList |
| package hotel.booking.system;  import java.util.ArrayList;  public class WaitingList {      private ArrayList <User> VIP;  private ArrayList <User> member;  private ArrayList <User> normal;    public WaitingList() {  VIP = new ArrayList <User>();  member = new ArrayList <User>();  normal = new ArrayList <User>();  }    public void addWaiting(Booking booking){  User user = booking.getUser();  switch(user.get\_member\_type())  {  //add VIP into VIP array list  case "VIP":  VIP.add(user);  break;  //add member into member array list  case "member":  member.add(user);  break;  //add non-member into normal array list  case "normal":  normal.add(user);  break;  //invalid member type cannot add waiting list  default:  throw new IllegalArgumentException("Invalid user type");  }  }    public ArrayList <User> getWaiting(String member\_type){    switch(member\_type)  {  //return to VIP array list  case "VIP":  return VIP;  //return to member array list  case "member":  return member;  //return to normal array list  case "normal":  return normal;  default:  throw new IllegalArgumentException("Invalid user type");  }    }    public void removeWaiting(Booking booking){  User user = booking.getUser();  switch(user.get\_member\_type())  {  //remove VIP from VIP array list  case "VIP":  VIP.remove(user);  break;  //remove member from member array list  case "member":  member.remove(user);  break;  //remove non-member from normal array list  case "normal":  normal.remove(user);  break;  default:  throw new IllegalArgumentException("Invalid user type");  }    }      } |

|  |
| --- |
| Booking |
| package hotel.booking.system;  import java.util.\*;  public class Booking {    private static final int ***VIP\_MEMBER\_MAX\_ROOM*** = 3;  private static final int ***MEMBER\_MAX\_ROOM*** = 2;  private static final int ***NORMAL\_MAX\_ROOM*** = 1;    private Printer printer = new Printer();  private User user;  private int vip\_rooms;  private int deluxe\_rooms;  private int standard\_rooms;    public Booking(User user, int vip\_rooms, int deluxe\_rooms, int standard\_rooms){  this.user = user;  this.vip\_rooms = vip\_rooms;  this.deluxe\_rooms = deluxe\_rooms;  this.standard\_rooms = standard\_rooms;  }    public User getUser() {  return user;  }    public int get\_vip\_rooms() {  return vip\_rooms;  }    public int get\_deluxe\_rooms() {  return deluxe\_rooms;  }    public int get\_standard\_rooms() {  return standard\_rooms;  }    public void setBooking(Room room, WaitingList waiting\_list, int requested\_room\_qty) {    if (requested\_room\_qty <= 0) {  throw new IllegalArgumentException("Room quantity should be greater than 0");  }    String member\_type = user.get\_member\_type();  int vip\_room\_qty = room.get\_VIP();  int deluxe\_room\_qty = room.get\_deluxe();  int standard\_room\_qty = room.get\_standard();  // Store initial room quantities to revert changes if necessary  int initialVipQty = vip\_room\_qty;  int initialDeluxeQty = deluxe\_room\_qty;  int initialStandardQty = standard\_room\_qty;  boolean rewardRedemption = false;    //VIP members  if(member\_type.equals("VIP")) {  //Check if the room quantity requested by VIP is more than maximum room can be booked  if(requested\_room\_qty > ***VIP\_MEMBER\_MAX\_ROOM***){  throw new IllegalArgumentException("Room quantity requested more than maximum quantity allowed");  }    //When the number of rooms requested cannot be fulfilled, place VIP into waiting list  else if (vip\_room\_qty + deluxe\_room\_qty + standard\_room\_qty < requested\_room\_qty) {  waiting\_list.addWaiting(this);  }    else {  int allocatedRooms = 0;  int vipRooms = 0;  int deluxeRooms = 0;  int standardRooms = 0;    for (int i=0; i < requested\_room\_qty; i++) {  //Allocate default room of VIP: VIP room  if (room.checkRoom("VIP")) {  vipRooms++;  vip\_room\_qty--;  room.set\_VIP(vip\_room\_qty);  allocatedRooms++;  vip\_room\_qty = room.get\_VIP();  }  //When VIP unavailable, allocate Deluxe room  else if(room.checkRoom("Deluxe")) {  deluxeRooms++;  deluxe\_room\_qty--;  room.set\_deluxe(deluxe\_room\_qty);  allocatedRooms++;  deluxe\_room\_qty = room.get\_deluxe();  }  //When VIP and Deluxe unavailable, allocate Standard room  else {  standardRooms++;  standard\_room\_qty--;  room.set\_standard(standard\_room\_qty);  allocatedRooms++;  standard\_room\_qty = room.get\_standard();  }  }  // If all requested rooms were allocated, else place the member on the waiting list  if (allocatedRooms == requested\_room\_qty) {  this.vip\_rooms = vipRooms;  this.deluxe\_rooms = deluxeRooms;  this.standard\_rooms = standardRooms;  printer.printInfo(this);  } else {  // Revert changes  room.set\_VIP(initialVipQty);  room.set\_deluxe(initialDeluxeQty);  room.set\_standard(initialStandardQty);  waiting\_list.addWaiting(this);  }  }  }  //Normal member  else if (member\_type.equals("member")) {  //Check if the room quantity requested by normal member is more than maximum room can be booked  if(requested\_room\_qty > ***MEMBER\_MAX\_ROOM***){  throw new IllegalArgumentException("Room quantity requested more than maximum quantity allowed");  }    //When the number of rooms requested cannot be fulfilled, place normal member into waiting list  else if (!user.get\_excl\_reward() && deluxe\_room\_qty + standard\_room\_qty < requested\_room\_qty) {  waiting\_list.addWaiting(this);  }  else if (user.get\_excl\_reward() && vip\_room\_qty + deluxe\_room\_qty + standard\_room\_qty < requested\_room\_qty) {  waiting\_list.addWaiting(this);  }    else {    int allocatedRooms = 0;  int vipRooms = 0;  int deluxeRooms = 0;  int standardRooms = 0;    for (int i=0; i < requested\_room\_qty; i++) {  //Allocate default room of normal member: Deluxe  if (room.checkRoom("Deluxe")) {  deluxeRooms++;  deluxe\_room\_qty--;  room.set\_deluxe(deluxe\_room\_qty);  allocatedRooms++;  }  //Default room unavailable, assign standard room to normal member  else if(room.checkRoom("Standard")){  standardRooms++;  standard\_room\_qty--;  room.set\_standard(standard\_room\_qty);  allocatedRooms++;  }  //When Standard room not available, allocate VIP room to user with exclusive reward  else if (user.get\_excl\_reward() && room.checkRoom("VIP") && !rewardRedemption){  vipRooms++;  vip\_room\_qty--;  room.set\_VIP(vip\_room\_qty);  rewardRedemption = true;  allocatedRooms++;  }  //When Deluxe room not available, allocate user to waiting list  else {  break;  }  }  // If all requested rooms were allocated, else place the member on the waiting list  if (allocatedRooms == requested\_room\_qty) {  this.vip\_rooms = vipRooms;  this.deluxe\_rooms = deluxeRooms;  this.standard\_rooms = standardRooms;  printer.printInfo(this);  if (rewardRedemption) {  user.set\_excl\_reward(false);  }  } else {  // Revert changes  room.set\_VIP(initialVipQty);  room.set\_deluxe(initialDeluxeQty);  room.set\_standard(initialStandardQty);  waiting\_list.addWaiting(this);  }  }  }  //Non-member  else {  //Check if room quantity requested by non-member is more than maximum room can be booked  if(requested\_room\_qty > ***NORMAL\_MAX\_ROOM***){  throw new IllegalArgumentException("Room quantity requested more than maximum quantity allowed");  }  //When the number of rooms requested cannot be fulfilled, place non-member into waiting list  else if (standard\_room\_qty < requested\_room\_qty) {  waiting\_list.addWaiting(this);  }  //Allocate default room of non-member: Standard room  else {  this.standard\_rooms++;  standard\_room\_qty--;  room.set\_standard(standard\_room\_qty);  printer.printInfo(this);  }  }  }    public void cancelBooking(Room room, WaitingList waiting\_list) {    boolean waitingListRemoved = false;    String member\_type = getUser().get\_member\_type();  for (User waitList: waiting\_list.getWaiting(member\_type)) {  if (waitList == this.getUser()) {  waiting\_list.removeWaiting(this);  waitingListRemoved = true;  break;  }  }  if (!waitingListRemoved) {  int vipToRelease = get\_vip\_rooms();  int deluxeToRelease = get\_deluxe\_rooms();  int standardToRelease = get\_standard\_rooms();    room.set\_VIP(room.get\_VIP() + vipToRelease);  room.set\_deluxe(room.get\_deluxe() + deluxeToRelease);  room.set\_standard(room.get\_standard() + standardToRelease);  }  }  } |

|  |
| --- |
| Room |
| package hotel.booking.system;  public class Room {    private int vip;  private int deluxe;  private int standard;  private boolean available;    public Room(int vip\_rooms, int deluxe\_rooms, int standard\_rooms){  vip = vip\_rooms;  deluxe = deluxe\_rooms;  standard = standard\_rooms;  }    //method to check for room availability  public boolean checkRoom(String room\_type){  //not ready for testing  return available;  }    public int get\_VIP(){  return vip;  }    public void set\_VIP(int num){  vip = num;  }    public int get\_deluxe(){  return deluxe;  }    public void set\_deluxe(int num){  deluxe = num;  }    public int get\_standard(){  return standard;  }  public void set\_standard(int num){  standard = num;  }    } |

|  |
| --- |
| Printer |
| package hotel.booking.system;  public class Printer {    public void printInfo(Booking booking) {  //Not ready for testing  }  } |

## Test Code

|  |
| --- |
| UserTest |
| package hotel.booking.system;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  import hotel.booking.system.\*;  import static org.junit.Assert.\*;  import static org.mockito.ArgumentMatchers.*anyInt*;  import static org.mockito.Mockito.\*;  import org.junit.Ignore;  import org.junit.Test;  import org.junit.runner.RunWith;  *@RunWith*(JUnitParamsRunner.class)  public class UserTest {    /\*  \* User Class Unit Test  \* Test Case 1.1.1~1.1.5  \* test with VALID user constructor  \*/    *@Test*  *@Parameters*(method = "ValidUser")  public void testValidUser(String name, String member\_type, boolean excl\_reward) {  User user = new User(name, member\_type, excl\_reward);  //Verify User object is created successfully  *assertEquals*(name, user.getName());  *assertEquals*(member\_type, user.get\_member\_type());  *assertEquals*(excl\_reward, user.get\_excl\_reward());  }    private Object[] ValidUser() {  return new Object[] {  new Object[] {"Alicia", "VIP", false},  new Object[] {"Zebra", "VIP", true},  new Object[] {"Barbel", "member", false},  new Object[] {"Joey", "member", true},  new Object[] {"Cariene", "normal", false}  };  }    /\*  \* User Class Unit Test  \* Test Case 1.2.1~1.2.17  \* test with INVALID user constructor  \*/    *@Test*(expected=IllegalArgumentException.class)  *@Parameters*(method = "InvalidUser")  public void testInvalidUser(String name, String member\_type, boolean excl\_reward) {  User user = new User(name, member\_type, excl\_reward);  }    private Object[] InvalidUser() {  return new Object[] {  new Object[]{null, "VIP", false}, //VIP member with null name (no exclusive reward)  new Object[]{null, "VIP", true}, //VIP member with null name (exclusive reward)  new Object[]{null, "member", false}, //Normal member with null name (no exclusive reward)  new Object[]{null, "member", true}, //Normal member with null name (exclusive reward)  new Object[]{null, "normal", false}, //Non-member with null name  new Object[]{"", "VIP", false}, //VIP member with empty name (no exclusive reward)  new Object[]{"", "VIP", true}, //VIP member with empty name (exclusive reward)  new Object[]{"", "member", false}, //Normal member (no exclusive reward) with empty name  new Object[]{"", "member", true}, //Normal member (exclusive reward) with empty name  new Object[]{"", "normal", false}, //Non-member with empty name  new Object[]{"Alice", null, false}, //null member with no exclusive reward  new Object[]{"Brian", null, true}, //null member with exclusive reward  new Object[]{"Cecelia", "", false}, //empty member with no exclusive reward  new Object[]{"Desmond", "", true}, //empty member with exclusive reward  new Object[]{"Esther", "vipMember", false}, //invalid member with no exclusive reward  new Object[]{"Felicia", "try123", true}, //invalid member with exclusive reward  new Object[] {"Alice", "normal", true} //non-member with exclusive reward  };  }    /\*  \* User Class Unit Test  \* Test Case 1.3.1  \* test with VALID set exclusive reward  \*/      *@Test*  public void testValidSetUserExclReward() {  User user = new User("Alice", "member", true);  user.set\_excl\_reward(false);  *assertEquals*(false, user.get\_excl\_reward());  }    /\*  \* User Class Unit Test  \* Test Case 1.4.1  \* test with INVALID set exclusive reward  \*/      *@Test*(expected=IllegalArgumentException.class)  public void testInvalidSetUserExclReward()  {  User user = new User("Alice", "normal", false);  user.set\_excl\_reward(true);  }  } |

|  |
| --- |
| WaitingListTest |
| package hotel.booking.system;  import hotel.booking.system.\*;  import java.io.File;  import java.io.FileNotFoundException;  import java.util.ArrayList;  import java.util.Scanner;  import org.junit.AfterClass;  import org.junit.Before;  import org.junit.BeforeClass;  import org.junit.Ignore;  import org.junit.Test;  import static org.junit.Assert.\*;  import static org.mockito.Mockito.\*;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  *@RunWith*(JUnitParamsRunner.class)  public class WaitingListTest {  WaitingList waitingList = new WaitingList();  static ArrayList<String[]> *linesRead*;  static Scanner *inputStream*;  *@BeforeClass*  public static void setupClass() {  *linesRead* = new ArrayList<String[]>();  String fileName = "user.txt";  *inputStream* = null;  System.***out***.println("Reading test values from file " + fileName + "\n");  try {  *inputStream* = new Scanner(new File(fileName));  } catch (FileNotFoundException e) {  System.***out***.println("Error opening the file " + fileName);  System.*exit*(0);  }  while (*inputStream*.hasNextLine()) {  String singleLine = *inputStream*.nextLine();  String[] tokens = singleLine.split(",");  *linesRead*.add(tokens);  }  }  *@AfterClass*  public static void endClass() {  // Closing the input file once all tests are complete  System.***out***.println("Closing input file");  *inputStream*.close();  }  *@Before*  public void setupMethod() {  System.***out***.println("Initializing the list");  waitingList = new WaitingList();  }  /\*  \* WaitingList Class Unit Test  \* Test Case 2.1.1  \* Test add waiting list with VIP member\_type  \*/    *@Test*  public void testValid\_AddWaitingVIP() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(0)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  *assertTrue*(waitingList.getWaiting(*linesRead*.get(0)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.1.2  \* Test add waiting list with member member\_type  \*/    *@Test*  public void testValid\_AddWaitingMember() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(1)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  *assertTrue*(waitingList.getWaiting(*linesRead*.get(1)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.1.3  \* Test add waiting list with normal member\_type  \*/    *@Test*  public void testValid\_AddWaitingNormal() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(2)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  *assertTrue*(waitingList.getWaiting(*linesRead*.get(2)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.2.1  \* Test add waiting list with INVALID member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid1\_AddWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(3)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.2.2  \* Test add waiting list with INVALID member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid2\_AddWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(4)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.2.3  \* Test add waiting list with empty member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid3\_AddWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(5)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.2.4  \* Test add waiting list with null member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid4\_AddWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(6)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.3.1  \* Test the remove waiting list method with VIP member\_type  \*/    *@Test*  public void testRemoveWaitingVIP() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(0)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  *assertFalse*(waitingList.getWaiting(*linesRead*.get(0)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.3.2  \* Test the remove waiting list method with member member\_type  \*/    *@Test*  public void testRemoveWaitingMember() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(1)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  *assertFalse*(waitingList.getWaiting(*linesRead*.get(1)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.3.3  \* Test the remove waiting list method with normal member\_type  \*/    *@Test*  public void testRemoveWaitingNormal() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(2)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  *assertFalse*(waitingList.getWaiting(*linesRead*.get(2)[0]).contains(mockUser));  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.4.1  \* Test the remove waiting list method with INVALID member\_type  \*/    //Test remove waiting list with INVALID member type  *@Test*(expected=IllegalArgumentException.class)  public void testInvalid1\_RemoveWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(3)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.4.2  \* Test the remove waiting list method with INVALID member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid2\_RemoveWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(4)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.4.3  \* Test the remove waiting list method with empty member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid3\_RemoveWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(5)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.4.4  \* Test the remove waiting list method with null member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid4\_RemoveWaiting() {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(6)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.removeWaiting(mockBooking);  }  /\*  \*WaitingList Class Unit Test  \* Test Case 2.5.1  \* Test get waiting list to get waiting list based on member\_type  \*/    *@Test*  public void testValid\_GetWaiting() {  // Test getWaiting for each member type  for (int i = 7; i < 12; i++) {  User mockUser = *mock*(User.class);  *when*(mockUser.get\_member\_type()).thenReturn(*linesRead*.get(i)[0]);  Booking mockBooking = *mock*(Booking.class);  *when*(mockBooking.getUser()).thenReturn(mockUser);  waitingList.addWaiting(mockBooking);  }    *assertEquals*(Integer.*parseInt*(*linesRead*.get(12)[0]), waitingList.getWaiting("VIP").size());  *assertEquals*(Integer.*parseInt*(*linesRead*.get(13)[0]), waitingList.getWaiting("member").size());  *assertEquals*(Integer.*parseInt*(*linesRead*.get(14)[0]), waitingList.getWaiting("normal").size());  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.6.1  \* Test get waiting list with INVALID member type  \*/      *@Test*(expected=IllegalArgumentException.class)  public void testInvalid1\_GetWaiting() {  waitingList.getWaiting(*linesRead*.get(3)[0]);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.6.2  \* Test get waiting with INVALID member type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid2\_GetWaiting() {  waitingList.getWaiting(*linesRead*.get(4)[0]);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.6.3  \* Test get waiting with empty member type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid3\_GetWaiting() {  waitingList.getWaiting(*linesRead*.get(5)[0]);  }    /\*  \*WaitingList Class Unit Test  \* Test Case 2.6.4  \* Test get waiting with null member type  \*/    *@Test*(expected=IllegalArgumentException.class)  public void testInvalid4\_GetWaiting() {  waitingList.getWaiting(*linesRead*.get(6)[0]);  }  } |

|  |
| --- |
| BookingTest |
| package hotel.booking.system;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  import hotel.booking.system.\*;  import static org.junit.Assert.\*;  import static org.mockito.ArgumentMatchers.*anyInt*;  import static org.mockito.Mockito.\*;  import java.sql.Array;  import java.util.ArrayList;  import java.util.Arrays;  import org.junit.Ignore;  import org.junit.Test;  import org.junit.runner.RunWith;  *@RunWith*(JUnitParamsRunner.class)  public class BookingTest {    int vipRoom;  int deluxeRoom;  int standardRoom;    /\*  \*Booking Class Unit Test  \* Test Case 3.1.1  \* Test get user method  \*/    *@Test*  public void testGetUser() {  User user = new User("Alice", "VIP", true);  Booking booking = new Booking(user, 0, 0, 0);  User ARuser = booking.getUser();  *assertEquals*(user, ARuser);  }    /\*  \*Booking Class Unit Test  \* Test Case 3.2.1  \* Test get VIP room method  \*/    *@Test*  public void testGetVIPRoom() {  User user = new User("Alice", "VIP", true);  Booking booking = new Booking(user, 2, 1, 0);  int ARvip = booking.get\_vip\_rooms();  *assertEquals*(2, ARvip);  }    /\*  \*Booking Class Unit Test  \* Test Case 3.3.1  \* Test get Deluxe room method  \*/    *@Test*  public void testGetDeluxeRoom() {  User user = new User("Alice", "VIP", true);  Booking booking = new Booking(user, 2, 1, 0);  int ARdeluxe = booking.get\_deluxe\_rooms();  *assertEquals*(1, ARdeluxe);  }    /\*  \*Booking Class Unit Test  \* Test Case 3.4.1  \* Test get Standard room method  \*/    *@Test*  public void testGetStandardRoom() {  User user = new User("Alice", "VIP", true);  Booking booking = new Booking(user, 2, 1, 0);  int ARstandard = booking.get\_standard\_rooms();  *assertEquals*(0, ARstandard);  }    /\*  \*Booking Class Unit Test  \* Test Case 3.5.1 ~ 3.5.25  \* Test VALID set booking method for VIP member\_type  \*/    *@Test*  *@Parameters*({  "VIP, 3, 3, 3, 1, 1, 0, 0, false",  "VIP, 3, 3, 3, 3, 3, 0, 0, false",  "VIP, 2, 3, 3, 3, 2, 1, 0, false",  "VIP, 1, 1, 3, 3, 1, 1, 1, false",  "VIP, 3, 3, 0, 1, 1, 0, 0, false",  "VIP, 3, 3, 0, 3, 3, 0, 0, false",  "VIP, 2, 3, 0, 3, 2, 1, 0, false",  "VIP, 1, 1, 0, 3, 0, 0, 0, true",  "VIP, 3, 0, 3, 1, 1, 0, 0, false",  "VIP, 2, 0, 3, 3, 2, 0, 1, false",  "VIP, 1, 0, 1, 3, 0, 0, 0, true",  "VIP, 3, 0, 0, 1, 1, 0, 0, false",  "VIP, 3, 0, 0, 3, 3, 0, 0, false",  "VIP, 2, 0, 0, 3, 0, 0, 0, true",  "VIP, 0, 3, 3, 1, 0, 1, 0, false",  "VIP, 0, 3, 3, 3, 0, 3, 0, false",  "VIP, 0, 2, 3, 3, 0, 2, 1, false",  "VIP, 0, 1, 1, 3, 0, 0, 0, true",  "VIP, 0, 3, 0, 1, 0, 1, 0, false",  "VIP, 0, 3, 0, 3, 0, 3, 0, false",  "VIP, 0, 1, 0, 3, 0, 0, 0, true",  "VIP, 0, 0, 3, 1, 0, 0, 1, false",  "VIP, 0, 0, 3, 3, 0, 0, 3, false",  "VIP, 0, 0, 1, 3, 0, 0, 0, true",  "VIP, 0, 0, 0, 1, 0, 0, 0, true"  })    public void testValidSetBooking\_VIPMember(String member\_type, int initialVipRoom, int initialDeluxeRoom, int initialStandardRoom,int reqRoom, int bookVIP, int bookDeluxe, int bookStandard, boolean waitinglist) {  vipRoom = initialVipRoom;  deluxeRoom = initialDeluxeRoom;  standardRoom = initialStandardRoom;    // Mock the User, Room, and WaitingList  User user = *mock*(User.class);  Room room = *mock*(Room.class);  WaitingList waitingList = *mock*(WaitingList.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);    // Mocking room availability  *when*(room.checkRoom("VIP")).thenAnswer(invocation -> vipRoom > 0);  *when*(room.checkRoom("Deluxe")).thenAnswer(invocation -> deluxeRoom > 0);  *when*(room.checkRoom("Standard")).thenAnswer(invocation -> standardRoom > 0);    *doAnswer*(invocation ->{  vipRoom--;  return null;  }).when(room).set\_VIP(*anyInt*());    *doAnswer*(invocation ->{  deluxeRoom--;  return null;  }).when(room).set\_deluxe(*anyInt*());    *doAnswer*(invocation ->{  standardRoom--;  return null;  }).when(room).set\_standard(*anyInt*());  // Mocking room count getters  *when*(room.get\_VIP()).thenReturn(vipRoom);  *when*(room.get\_deluxe()).thenReturn(deluxeRoom);  *when*(room.get\_standard()).thenReturn(standardRoom);  // Booking instantiation and action  Booking booking = new Booking(user, 0, 0, 0);  booking.setBooking(room, waitingList, reqRoom);  // Assertions on room allocation  *assertEquals*(bookVIP, booking.get\_vip\_rooms());  *assertEquals*(bookDeluxe, booking.get\_deluxe\_rooms());  *assertEquals*(bookStandard, booking.get\_standard\_rooms());  // Verify interactions with the waiting list  if (waitinglist) {  *verify*(waitingList).addWaiting(booking);  } else {  *verify*(waitingList, *never*()).addWaiting(booking);  }  }    /\*  \*Booking Class Unit Test  \* Test Case 3.6.1 ~ 3.6.24  \* Test INVALID set booking method for VIP member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  *@Parameters*({  "VIP, 3, 3, 3, -1",  "VIP, 3, 3, 3, 0",  "VIP, 3, 3, 3, 4",  "VIP, 3, 3, 0, -1",  "VIP, 3, 3, 0, 0",  "VIP, 3, 3, 0, 4",  "VIP, 3, 0, 3, -1",  "VIP, 3, 0, 3, 0",  "VIP, 3, 0, 3, 4",  "VIP, 3, 0, 0, -1",  "VIP, 3, 0, 0, 0",  "VIP, 3, 0, 0, 4",  "VIP, 0, 3, 3, -1",  "VIP, 0, 3, 3, 0",  "VIP, 0, 3, 3, 4",  "VIP, 0, 3, 0, -1",  "VIP, 0, 3, 0, 0",  "VIP, 0, 3, 0, 4",  "VIP, 0, 0, 3, -1",  "VIP, 0, 0, 3, 0",  "VIP, 0, 0, 3, 4",  "VIP, 0, 0, 0, -1",  "VIP, 0, 0, 0, 0",  "VIP, 0, 0, 0, 4"  })    public void testInvalidSetBooking\_VIPMember(String member\_type, int initialVipRoom, int initialDeluxeRoom, int initialStandardRoom, int reqRoom) {  vipRoom = initialVipRoom;  deluxeRoom = initialDeluxeRoom;  standardRoom = initialStandardRoom;    // Mock the User, Room, and WaitingList  User user = *mock*(User.class);  Room room = *mock*(Room.class);  WaitingList waitingList = *mock*(WaitingList.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);    // Mocking room availability  *when*(room.checkRoom("VIP")).thenAnswer(invocation -> vipRoom > 0);  *when*(room.checkRoom("Deluxe")).thenAnswer(invocation -> deluxeRoom > 0);  *when*(room.checkRoom("Standard")).thenAnswer(invocation -> standardRoom > 0);    *doAnswer*(invocation ->{  vipRoom--;  return null;  }).when(room).set\_VIP(*anyInt*());    *doAnswer*(invocation ->{  deluxeRoom--;  return null;  }).when(room).set\_deluxe(*anyInt*());    *doAnswer*(invocation ->{  standardRoom--;  return null;  }).when(room).set\_standard(*anyInt*());  // Mocking room count getters  *when*(room.get\_VIP()).thenReturn(vipRoom);  *when*(room.get\_deluxe()).thenReturn(deluxeRoom);  *when*(room.get\_standard()).thenReturn(standardRoom);  // Booking instantiation and action  Booking booking = new Booking(user, 0, 0, 0);  booking.setBooking(room, waitingList, reqRoom);  }  /\*  \*Booking Class Unit Test  \* Test Case 3.7.1 ~ 3.7.43  \* Test VALID set booking method for member member\_type  \*/    *@Test*  *@Parameters*({  "member, true, 3, 3, 3, 1, 0, 1, 0, false, false",  "member, true, 3, 3, 3, 2, 0, 2, 0, false, false",  "member, true, 3, 1, 3, 2, 0, 1, 1, false, false",  "member, false, 3, 3, 3, 1, 0, 1, 0, false, false",  "member, false, 3, 3, 3, 2, 0, 2, 0, false, false",  "member, false, 3, 1, 3, 2, 0, 1, 1, false, false",  "member, true, 3, 3, 0, 1, 0, 1, 0, false, false",  "member, true, 3, 3, 0, 2, 0, 2, 0, false, false",  "member, true, 3, 1, 0, 2, 1, 1, 0, false, true",  "member, false, 3, 3, 0, 1, 0, 1, 0, false, false",  "member, false, 3, 3, 0, 2, 0, 2, 0, false, false",  "member, false, 3, 1, 0, 2, 0, 0, 0, true, false",  "member, true, 3, 0, 3, 1, 0, 0, 1, false, false",  "member, true, 3, 0, 3, 2, 0, 0, 2, false, false",  "member, true, 1, 0, 1, 2, 1, 0, 1, false, true",  "member, false, 3, 0, 3, 1, 0, 0, 1, false, false",  "member, false, 3, 0, 3, 2, 0, 0, 2, false, false",  "member, false, 3, 0, 1, 2, 0, 0, 0, true, false",  "member, true, 3, 0, 0, 1, 1, 0, 0, false, true",  "member, true, 3, 0, 0, 2, 0, 0, 0, true, false",  "member, false, 3, 0, 0, 1, 0, 0, 0, true, false",  "member, true, 0, 3, 3, 1, 0, 1, 0, false, false",  "member, true, 0, 3, 3, 2, 0, 2, 0, false, false",  "member, true, 0, 1, 3, 2, 0, 1, 1, false, false",  "member, false, 0, 3, 3, 1, 0, 1, 0, false, false",  "member, false, 0, 3, 3, 2, 0, 2, 0, false, false",  "member, false, 0, 1, 3, 2, 0, 1, 1, false, false",  "member, true, 0, 3, 0, 1, 0, 1, 0, false, false",  "member, true, 0, 3, 0, 2, 0, 2, 0, false, false",  "member, true, 0, 1, 0, 2, 0, 0, 0, true, false",  "member, false, 0, 3, 0, 1, 0, 1, 0, false, false",  "member, false, 0, 3, 0, 2, 0, 2, 0, false, false",  "member, false, 0, 1, 0, 2, 0, 0, 0, true, false",  "member, true, 0, 0, 3, 1, 0, 0, 1, false, false",  "member, true, 0, 0, 3, 2, 0, 0, 2, false, false",  "member, true, 0, 0, 1, 2, 0, 0, 0, true, false",  "member, false, 0, 0, 3, 1, 0, 0, 1, false, false",  "member, false, 0, 0, 3, 2, 0, 0, 2, false, false",  "member, false, 0, 0, 1, 2, 0, 0, 0, true, false",  "member, true, 0, 0, 0, 1, 0, 0, 0, true, false",  "member, true, 0, 0, 0, 2, 0, 0, 0, true, false",  "member, false, 0, 0, 0, 1, 0, 0, 0, true, false",  "member, false, 0, 0, 0, 2, 0, 0, 0, true, false"  })      public void testValidSetBooking\_NormalMember(String member\_type, boolean excReward, int initialVipRoom, int initialDeluxeRoom, int initialStandardRoom,int reqRoom, int bookVIP, int bookDeluxe, int bookStandard, boolean waitinglist, boolean rewardRedemption) {  vipRoom = initialVipRoom;  deluxeRoom = initialDeluxeRoom;  standardRoom = initialStandardRoom;    // Mock the User, Room, and WaitingList  User user = *mock*(User.class);  Room room = *mock*(Room.class);  WaitingList waitingList = *mock*(WaitingList.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);  *when*(user.get\_excl\_reward()).thenAnswer(invocation -> excReward);    *doAnswer*(invocation -> {  boolean arg = invocation.getArgument(0); // Get the boolean argument passed to the method  if (arg) {  // Update the excl\_reward to false after the first loop  *when*(user.get\_excl\_reward()).thenReturn(false);  }  return null; // Return null or any other value if necessary  }).when(user).set\_excl\_reward(*anyBoolean*());  // Mocking room availability  *when*(room.checkRoom("VIP")).thenAnswer(invocation -> vipRoom > 0);  *when*(room.checkRoom("Deluxe")).thenAnswer(invocation -> deluxeRoom > 0);  *when*(room.checkRoom("Standard")).thenAnswer(invocation -> standardRoom > 0);    *doAnswer*(invocation ->{  vipRoom--;  return null;  }).when(room).set\_VIP(*anyInt*());    *doAnswer*(invocation ->{  deluxeRoom--;  return null;  }).when(room).set\_deluxe(*anyInt*());    *doAnswer*(invocation ->{  standardRoom--;  return null;  }).when(room).set\_standard(*anyInt*());  // Mocking room count getters  *when*(room.get\_VIP()).thenReturn(vipRoom);  *when*(room.get\_deluxe()).thenReturn(deluxeRoom);  *when*(room.get\_standard()).thenReturn(standardRoom);  // Booking instantiation and action  Booking booking = new Booking(user, 0, 0, 0);  booking.setBooking(room, waitingList, reqRoom);  // Assertions on room allocation  *assertEquals*(bookVIP, booking.get\_vip\_rooms());  *assertEquals*(bookDeluxe, booking.get\_deluxe\_rooms());  *assertEquals*(bookStandard, booking.get\_standard\_rooms());  // Verify interactions with the waiting list  if (waitinglist) {  *verify*(waitingList).addWaiting(booking);  } else {  *verify*(waitingList, *never*()).addWaiting(booking);  }  // Verify reward redemption  if (rewardRedemption) {  *verify*(user).set\_excl\_reward(false);  } else {  *verify*(user, *never*()).set\_excl\_reward(*anyBoolean*());  }  }    /\*  \*Booking Class Unit Test  \* Test Case 3.8.1 ~ 3.8.48  \* Test INVALID set booking method for member member\_type  \*/    *@Test*(expected=IllegalArgumentException.class)  *@Parameters*({  "member, true, 3, 3, 3, -1",  "member, true, 3, 3, 3, 0",  "member, true, 3, 3, 3, 3",  "member, false, 3, 3, 3, -1",  "member, false, 3, 3, 3, 0",  "member, false, 3, 3, 3, 3",  "member, true, 3, 3, 0, -1",  "member, true, 3, 3, 0, 0",  "member, true, 3, 3, 0, 3",  "member, false, 3, 3, 0, -1",  "member, false, 3, 3, 0, 0",  "member, false, 3, 3, 0, 3",  "member, true, 3, 0, 3, -1",  "member, true, 3, 0, 3, 0",  "member, true, 3, 0, 3, 3",  "member, false, 3, 0, 3,-1",  "member, false, 3, 0, 3, 0",  "member, false, 3, 0, 3, 3",  "member, true, 3, 0, 0, -1",  "member, true, 3, 0, 0, 0",  "member, true, 3, 0, 0, 3",  "member, false, 3, 0, 0, -1",  "member, false, 3, 0, 0, 0",  "member, false, 3, 0, 0, 3",  "member, true, 0, 3, 3, -1",  "member, true, 0, 3, 3, 0",  "member, true, 0, 3, 3, 3",  "member, false, 0, 3, 3, -1",  "member, false, 0, 3, 3, 0",  "member, false, 0, 3, 3, 3",  "member, true, 0, 3, 0, -1",  "member, true, 0, 3, 0, 0",  "member, true, 0, 3, 0, 3",  "member, false, 0, 3, 0, -1",  "member, false, 0, 3, 0, 0",  "member, false, 0, 3, 0, 3",  "member, true, 0, 0, 3, -1",  "member, true, 0, 0, 3, 0",  "member, true, 0, 0, 3, 3",  "member, false, 0, 0, 3, -1",  "member, false, 0, 0, 3, 0",  "member, false, 0, 0, 3, 3",  "member, true, 0, 0, 0, -1",  "member, true, 0, 0, 0, 0",  "member, true, 0, 0, 0, 3",  "member, false, 0, 0, 0, -1",  "member, false, 0, 0, 0, 0",  "member, false, 0, 0, 0, 3"  })      public void testInvalidSetBooking\_NormalMember(String member\_type, boolean excReward, int initialVipRoom, int initialDeluxeRoom, int initialStandardRoom,int reqRoom) {  vipRoom = initialVipRoom;  deluxeRoom = initialDeluxeRoom;  standardRoom = initialStandardRoom;    // Mock the User, Room, and WaitingList  User user = *mock*(User.class);  Room room = *mock*(Room.class);  WaitingList waitingList = *mock*(WaitingList.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);  *when*(user.get\_excl\_reward()).thenAnswer(invocation -> excReward);    *doAnswer*(invocation -> {  boolean arg = invocation.getArgument(0); // Get the boolean argument passed to the method  if (arg) {  // Update the excl\_reward to false after the first loop  *when*(user.get\_excl\_reward()).thenReturn(false);  }  return null; // Return null or any other value if necessary  }).when(user).set\_excl\_reward(*anyBoolean*());  // Mocking room availability  *when*(room.checkRoom("VIP")).thenAnswer(invocation -> vipRoom > 0);  *when*(room.checkRoom("Deluxe")).thenAnswer(invocation -> deluxeRoom > 0);  *when*(room.checkRoom("Standard")).thenAnswer(invocation -> standardRoom > 0);    *doAnswer*(invocation ->{  vipRoom--;  return null;  }).when(room).set\_VIP(*anyInt*());    *doAnswer*(invocation ->{  deluxeRoom--;  return null;  }).when(room).set\_deluxe(*anyInt*());    *doAnswer*(invocation ->{  standardRoom--;  return null;  }).when(room).set\_standard(*anyInt*());  // Mocking room count getters  *when*(room.get\_VIP()).thenReturn(vipRoom);  *when*(room.get\_deluxe()).thenReturn(deluxeRoom);  *when*(room.get\_standard()).thenReturn(standardRoom);  // Booking instantiation and action  Booking booking = new Booking(user, 0, 0, 0);  booking.setBooking(room, waitingList, reqRoom);  }  /\*  \*Booking Class Unit Test  \* Test Case 3.9.1 ~ 3.9.2  \* Test VALID set booking method for normal member\_type  \*/    *@Test*  *@Parameters*({  "normal, 1, 1, 1, false",  "normal, 0, 1, 0, true",  })  public void testValidSetBooking\_NonMember(String member\_type, int standardRoom, int reqRoom, int bookStandard, boolean waitinglist) {  // Create a mock User for non-member  User user = *mock*(User.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);    // Create a mock Room with available Standard rooms  Room room = *mock*(Room.class);  *when*(room.checkRoom("Standard")).thenReturn(standardRoom > 0);  *when*(room.get\_standard()).thenReturn(standardRoom);    // Create a mock WaitingList  WaitingList waitingList = *mock*(WaitingList.class);    // Create a Booking instance  Booking booking = new Booking(user, 0, 0, 0);  // Call the setBooking method with requested room quantity  booking.setBooking(room, waitingList, reqRoom);    // Verify that the booking has been made successfully  *assertEquals*(bookStandard, booking.get\_standard\_rooms());    if (waitinglist) {  *verify*(waitingList).addWaiting(booking);  } else {  *verify*(waitingList, *never*()).addWaiting(booking);  }  }    /\*  \*Booking Class Unit Test  \* Test Case 3.10.1 ~ 3.10.6  \* Test INVALID set booking method for normal member\_type  \*/    *@Test*(expected = IllegalArgumentException.class)  *@Parameters*({  "normal, 1, -1",  "normal, 1, 0",  "normal, 1, 2",  "normal, 0, -1",  "normal, 0, 0",  "normal, 0, 2"  })  public void testInvalidSetBooking\_NonMember(String member\_type, int standardRoom, int reqRoom) {  // Create a mock User for non-member  User user = *mock*(User.class);  *when*(user.get\_member\_type()).thenReturn(member\_type);    // Create a mock Room with available Standard rooms  Room room = *mock*(Room.class);  *when*(room.checkRoom("Standard")).thenReturn(standardRoom > 0);  *when*(room.get\_standard()).thenReturn(standardRoom);    // Create a mock WaitingList  WaitingList waitingList = *mock*(WaitingList.class);    // Create a Booking instance  Booking booking = new Booking(user, 0, 0, 0);  // Call the setBooking method with requested room quantity  booking.setBooking(room, waitingList, reqRoom);    }    /\*  \*Booking Class Unit Test  \* Test Case 3.11.1 ~ 3.11.3  \* Test cancel booking method and verify released room quantity  \*/    private Object[] getBookingParam() {  Booking booking1 = new Booking(new User("Ali", "VIP", true), 2, 1, 0);  Booking booking2 = new Booking(new User("Bunny", "member", true), 0, 2, 0);  Booking booking3 = new Booking(new User("Cally", "normal", false), 0, 0, 1);  return new Object[] {  new Object[] {booking1, 3, 3, 3, 5, 4, 3},  new Object[] {booking2, 2, 2, 2, 2, 4, 2},  new Object[] {booking3, 2, 2, 2, 2, 2, 3}  };  }    *@Test*  *@Parameters*(method="getBookingParam")  public void testCancelBooking(Booking booking, int initialVIP, int initialDeluxe, int initialStandard, int expectedVIP, int expectedDeluxe, int expectedStandard) {  // Mock the Room and WaitingList  Room room = *mock*(Room.class);  WaitingList waitingList = *mock*(WaitingList.class);  *when*(room.get\_VIP()).thenReturn(initialVIP);  *when*(room.get\_deluxe()).thenReturn(initialDeluxe);  *when*(room.get\_standard()).thenReturn(initialStandard);    // Call the cancelBooking method  booking.cancelBooking(room, waitingList);  // Verify that the room quantities are updated  *verify*(room).set\_VIP(expectedVIP);  *verify*(room).set\_deluxe(expectedDeluxe);  *verify*(room).set\_standard(expectedStandard);  // Verify that the user is not removed from the waiting list  *verify*(waitingList, *never*()).removeWaiting(booking);  }    /\*  \*Booking Class Unit Test  \* Test Case 3.12.1 ~ 3.12.3  \* Test cancel booking and remove user from waiting list  \*/    private Object[] getWaitingListParam() {  Booking booking1 = new Booking(new User("Ali", "VIP", true), 2, 1, 0);  Booking booking2 = new Booking(new User("Bunny", "member", true), 0, 2, 0);  Booking booking3 = new Booking(new User("Cally", "normal", false), 0, 0, 1);  ArrayList<User> VIP = new ArrayList<User>();  ArrayList<User> member = new ArrayList<User>();  ArrayList<User> normal = new ArrayList<User>();  return new Object[] {  new Object[] {booking1, VIP},  new Object[] {booking2, member},  new Object[] {booking3, normal}  };  }    *@Test*  *@Parameters*(method = "getWaitingListParam")  public void cancelBookingInWaitingList(Booking booking, ArrayList<User> member\_type\_list) {  Room room = *mock*(Room.class);  WaitingList waitinglist = *mock*(WaitingList.class);  member\_type\_list.add(booking.getUser());  *when*(waitinglist.getWaiting(*anyString*())).thenReturn(member\_type\_list);  booking.cancelBooking(room, waitinglist);    *verify*(waitinglist).removeWaiting(booking);  }  } |

|  |
| --- |
| IntegrationTest |
| package hotel.booking.system;  import hotel.booking.system.\*;  import java.util.ArrayList;  import java.util.Scanner;  import org.junit.AfterClass;  import org.junit.Before;  import org.junit.BeforeClass;  import org.junit.Ignore;  import org.junit.Test;  import static org.junit.Assert.\*;  import static org.mockito.Mockito.\*;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  *@RunWith*(JUnitParamsRunner.class)  public class IntegrationTest {    /\*  \* Integration Test  \* Test Case 4.1.1  \* Test integration between booking, room, user, and waiting list  \*/  *@Test*  public void testBookingIntegration() {    User user = new User("Alice", "normal", false);  // Create a mock room  Room room = *mock*(Room.class);  *when*(room.checkRoom("VIP")).thenReturn(true);  *when*(room.checkRoom("Deluxe")).thenReturn(true);  *when*(room.checkRoom("Standard")).thenReturn(false);  *when*(room.get\_VIP()).thenReturn(2);  *when*(room.get\_deluxe()).thenReturn(1);  *when*(room.get\_standard()).thenReturn(0);    // Create a waiting list  WaitingList waitingList = new WaitingList();  // Create a booking  Booking booking = new Booking(user, 0, 0, 0);  // Perform the booking  booking.setBooking(room, waitingList, 1);    // Check if the booking was added to the waiting list  *assertTrue*(waitingList.getWaiting("normal").contains(user));  // Cancel the booking  booking.cancelBooking(room, waitingList);  // Check if the booking was removed from the waiting list  *assertFalse*(waitingList.getWaiting("normal").contains(user));  }    /\*  \* Integration Test  \* Test Case 4.1.2  \* Invalid test for integration between booking, room, user, and waiting list  \* with invalid user exclusive reward  \*/  *@Test*(expected = IllegalArgumentException.class)  public void testInvalidUserBookingIntegration() {    User user = new User("Alice", "normal", true); //user with invalid exclusive reward corresponding to member type  // Create a mock room  Room room = *mock*(Room.class);  *when*(room.checkRoom("VIP")).thenReturn(true);  *when*(room.checkRoom("Deluxe")).thenReturn(true);  *when*(room.checkRoom("Standard")).thenReturn(false);  *when*(room.get\_VIP()).thenReturn(2);  *when*(room.get\_deluxe()).thenReturn(1);  *when*(room.get\_standard()).thenReturn(0);    // Create a waiting list  WaitingList waitingList = new WaitingList();  // Create a booking  Booking booking = new Booking(user, 0, 0, 0);  // Perform the booking  booking.setBooking(room, waitingList, 1);    // Cancel the booking  booking.cancelBooking(room, waitingList);  }  /\*  \* Integration Test  \* Test Case 4.1.3  \* Invalid test for integration between booking, room, user, and waiting list  \* with invalid requested room quantity  \*/  *@Test*(expected = IllegalArgumentException.class)  public void testInvalidRoomBookingIntegration() {    User user = new User("Alice", "normal", false);  // Create a mock room  Room room = *mock*(Room.class);  *when*(room.checkRoom("VIP")).thenReturn(true);  *when*(room.checkRoom("Deluxe")).thenReturn(true);  *when*(room.checkRoom("Standard")).thenReturn(false);  *when*(room.get\_VIP()).thenReturn(2);  *when*(room.get\_deluxe()).thenReturn(1);  *when*(room.get\_standard()).thenReturn(0);    // Create a waiting list  WaitingList waitingList = new WaitingList();  // Create a booking  Booking booking = new Booking(user, 0, 0, 0);  // Perform the booking  booking.setBooking(room, waitingList, 3); //invalid requested room quantity for non member    // Cancel the booking  booking.cancelBooking(room, waitingList);  }  } |

|  |
| --- |
| TestSuite |
| package hotel.booking.system;  import org.junit.runner.RunWith;  import org.junit.runners.Suite;  import org.junit.runners.Suite.SuiteClasses;  *@RunWith*(Suite.class)  *@SuiteClasses*({ BookingTest.class, UserTest.class, WaitingListTest.class })  public class TestSuite {  } |

## Project folder

A screenshot of a computer program

Description automatically generated