MARKING REPORT

**Group number: 20\_\_\_**

|  |  |
| --- | --- |
| Name | Student ID |
| 1. Hwang Jen Fung | 2104489 |
| 2. Kok Yu San | 2104903 |
| 3. Tham Jing Yi | 2103429 |
| 4. Wong Yong Jun | 2103486 |

**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

[Assumptions 3](#_Toc164892293)

[Class Diagram 4](#_Toc164892294)

[Booking.java 5](#_Toc164892295)

[Printer.java 8](#_Toc164892296)

[Room.java 9](#_Toc164892297)

[User.java 11](#_Toc164892298)

[WaitingList.java 13](#_Toc164892299)

[testSuite.java 15](#_Toc164892300)

[TestBookingUnit.java 15](#_Toc164892301)

[TestUser.java 22](#_Toc164892302)

[TestWaitingList.java 25](#_Toc164892303)

[TestBookingIntegration.java 29](#_Toc164892304)

# Assumptions

1. Assume the same user can only make 1 booking at a time.
2. Assume the printer class is not ready for testing.
3. Assume the room class is not ready for testing.

# Class Diagram

A diagram of a computer

Description automatically generated

Application Code

|  |
| --- |
| Booking.java |
| package assignmenttest;  import java.util.ArrayList;  public class Booking {  private WaitingList waitingList;  ArrayList<Booking> bookedList = new ArrayList<Booking>();  private int tempVipRoom, tempDeluxeRoom, tempStandardRoom;  int[] userRoom = {0,0,0};  private User user;  private Room room;  private Printer printer;  public Booking(User user, Room room, WaitingList waitingList, Printer printer) {  this.user = user;  this.room = room;  this.waitingList = waitingList;  this.printer = printer;  }  //Create a booking list for the user  public Booking(User user, int[] room) {  this.user = user;  this.userRoom = room;  }    public void addBookedList(User user, int[] userRoom) {  bookedList.add(new Booking(user, userRoom));  }  public ArrayList<Booking> getBookedList() {  return bookedList;  }    public void setBookedList(ArrayList<Booking> bookedList) {  this.bookedList =bookedList;  }    public User getUser() {  return this.user;  }    public void setBooking() {  tempVipRoom = room.getVip();  tempDeluxeRoom = room.getDeluxe();  tempStandardRoom = room.getStandard();  int vipRoom = tempVipRoom;  int deluxeRoom = tempDeluxeRoom;  int standardRoom = tempStandardRoom;    if (user.getMember\_type().toLowerCase().equals("vip")) {  if (user.getNumRoom() <= 3 && user.getNumRoom() > 0) {  for (int i = 0; i < user.getNumRoom(); i++){  if (tempVipRoom > 0){  tempVipRoom -= 1;  room.bookRoom("vip");  userRoom[0] +=1;  }  else if (tempDeluxeRoom > 0) {  tempDeluxeRoom -= 1;  room.bookRoom("deluxe");  userRoom[1] +=1;  }  else if (tempStandardRoom > 0) {  tempStandardRoom -= 1;  room.bookRoom("standard");  userRoom[2] +=1;  }  else {  waitingList.addWaiting(user);  userRoom[0] = 0;  userRoom[1] = 0;  userRoom[2] = 0;  room.setVip(vipRoom);  room.setDeluxe(deluxeRoom);  room.setStandard(standardRoom);  printer.printInfo("Added to Waiting List");  break;  }  }  }  else {  throw new IllegalArgumentException();  }  }  else if (user.getMember\_type().toLowerCase().equals("member")) {  if (user.getNumRoom() <= 2 && user.getNumRoom() > 0) {  for (int i = 0; i < user.getNumRoom(); i++){  if (tempDeluxeRoom > 0) {  tempDeluxeRoom -= 1;  room.bookRoom("deluxe");  userRoom[1] +=1;  }  else if (tempStandardRoom > 0) {  tempStandardRoom -= 1;  room.bookRoom("standard");  userRoom[2] +=1;  } else if (user.getExcl\_reward() && tempVipRoom > 0 && (user.getNumRoom() == 1 || i== 1)) {  tempVipRoom -= 1;  user.setExcl\_reward(false);  room.bookRoom("vip");  userRoom[0] +=1;  }  else{  waitingList.addWaiting(user);  userRoom[0] = 0;  userRoom[1] = 0;  userRoom[2] = 0;  room.setVip(vipRoom);  room.setDeluxe(deluxeRoom);  room.setStandard(standardRoom);  printer.printInfo("Added to Waiting List");  break;  }  }  }  else {  throw new IllegalArgumentException();  }  }  else if (user.getMember\_type().toLowerCase().equals("normal")) {  if (user.getNumRoom() == 1) {  if (tempStandardRoom > 0) {  tempStandardRoom -= 1;  room.bookRoom("standard");  userRoom[2] +=1;  }  else {  waitingList.addWaiting(user);  printer.printInfo("Added to Waiting List");  }  }  else {  throw new IllegalArgumentException();  }  }  else {  throw new IllegalArgumentException();  }  if (userRoom[0] != 0 || userRoom[1] != 0 || userRoom[2] !=0) {  bookedList.add(new Booking(user,userRoom));  printer.printInfo("Booked Succesfully");  }    }    public int[] getUserRoom() {  return userRoom;  }    public void cancelBooking(User user) {  boolean userFound = false;  if (waitingList.getWaiting(user.getMember\_type()).contains(user)) {  waitingList.removeWaiting(user);  userFound = true;  }  for (int i = 0; i < bookedList.size(); i++) {  if (bookedList.get(i).getUser() == user) {  bookedList.remove(i);  room.setVip(room.getVip()+userRoom[0]);  room.setDeluxe(room.getDeluxe()+userRoom[1]);  room.setStandard(room.getStandard()+userRoom[2]);  userFound = true;  }  }    if (userFound == false)  throw new IllegalArgumentException();    }  } |

|  |
| --- |
| Printer.java |
| package assignmenttest;  public class Printer {  public void printInfo(String info) {  System.out.print(info);  }  } |

|  |
| --- |
| Room.java |
| package assignmenttest;  public class Room {  private int vip;  public int getVip() {  return vip;  }  public void setVip(int i) {  this.vip = i;  }  private int deluxe;  public int getDeluxe() {  return deluxe;  }  public void setDeluxe(int i) {  this.deluxe = i;  }  private int standard;  public int getStandard() {  return standard;  }  public void setStandard(int i) {  this.standard = i;  }    public boolean checkRoom(String roomType) {  switch(roomType) {  case "vip":  return (vip>0);  case "deluxe":  return (deluxe>0);  case "standard":  return (standard>0);  default:  throw new IllegalArgumentException();  }  }      public void bookRoom(String roomType) {  switch(roomType) {  case "vip":  vip -= 1;  break;  case "deluxe":  deluxe -= 1;  break;  case "standard":  standard -= 1;  break;  }  }  } |

|  |
| --- |
| User.java |
| package assignmenttest;  public class User{  private String name;  public String getName() {  return name;  }    public void setName(String name) {  if (name == null || name.equals(""))  throw new IllegalArgumentException();    this.name = name;  }  private String member\_type;  public String getMember\_type() {  return member\_type;  }    public void setMember\_type(String member\_type) {  if (member\_type == null)  throw new IllegalArgumentException();    if (!member\_type.toLowerCase().equals("vip") && !member\_type.toLowerCase().equals("member") && !member\_type.toLowerCase().equals("normal"))  throw new IllegalArgumentException();    this.member\_type = member\_type.toLowerCase();  }  private boolean excl\_reward;  public boolean getExcl\_reward() {  return excl\_reward;  }  public void setExcl\_reward(boolean excl\_reward) {  this.excl\_reward = excl\_reward;  }  private int numRoom;  public int getNumRoom() {  return numRoom;  }    public void setNumRoom(int numRoom) {  if (numRoom <= 0 || numRoom > 3)  throw new IllegalArgumentException();    this.numRoom = numRoom;  }    public User(String name, String member\_type, boolean excl\_reward, int numRoom) {  this.name = name;  this.member\_type = member\_type.toLowerCase();  this.excl\_reward = excl\_reward;  this.numRoom = numRoom;  }  public User() {  }  } |

|  |
| --- |
| WaitingList.java |
| package assignmenttest;  import java.util.ArrayList;  public class WaitingList{  ArrayList<User> VIP = new ArrayList<User>();  ArrayList<User> member = new ArrayList<User>();  ArrayList<User> normal = new ArrayList<User>();    public WaitingList() {  VIP = new ArrayList<>();  member = new ArrayList<>();  normal = new ArrayList<>();  }    public void addWaiting(User user){  if (user.getMember\_type().toLowerCase().equals("vip")){  VIP.add(user);  }  else if (user.getMember\_type().toLowerCase().equals("member")){  member.add(user);  }  else if (user.getMember\_type().toLowerCase().equals("normal")){  normal.add(user);  }  else{  throw new IllegalArgumentException();  }  }  public ArrayList<User> getWaiting(String member\_type){  if (member\_type.toLowerCase().equals("vip")){  return VIP;  }  else if (member\_type.toLowerCase().equals("member")){  return member;  }  else if (member\_type.toLowerCase().equals("normal")) {  return normal;  }  else  throw new IllegalArgumentException();  }  public void removeWaiting(User user) {  if (user.getMember\_type().toLowerCase().equals("vip") && VIP.contains(user)){  VIP.remove(user);  }  else if (user.getMember\_type().toLowerCase().equals("member") && member.contains(user)){  member.remove(user);  }  else if (user.getMember\_type().toLowerCase().equals("normal") && normal.contains(user)){  normal.remove(user);  }  else{  throw new IllegalArgumentException();  }  }  } |

JUnit Test Code

|  |
| --- |
| testSuite.java |
| package assignmenttest;  import org.junit.runner.RunWith;  import org.junit.runners.Suite;  import org.junit.runners.Suite.SuiteClasses;  @RunWith(value = Suite.class)  @SuiteClasses(value = {  TestBookingIntegration.class,  TestBookingUnit.class,  TestUser.class,  TestWaitingList.class,  })  public class testSuite {    } |

|  |
| --- |
| TestBookingUnit.java |
| package assignmenttest;  import static org.junit.Assert.assertArrayEquals;  import static org.junit.Assert.assertEquals;  import static org.junit.Assert.assertFalse;  import static org.mockito.Mockito.mock;  import static org.mockito.Mockito.times;  import static org.mockito.Mockito.verify;  import static org.mockito.Mockito.when;  import java.util.ArrayList;  import org.junit.Test;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  @RunWith(JUnitParamsRunner.class)  public class TestBookingUnit{  private Object[] getParamsForTestSetBooking() {  //num of room booked by user  int[] nums100 = {1, 0, 0};  int[] nums010 = {0, 1, 0};  int[] nums001 = {0, 0, 1};  int[] nums000 = {0, 0, 0};  int[] nums200 = {2, 0, 0};  int[] nums110 = {1, 1, 0};  int[] nums101 = {1, 0, 1};  int[] nums020 = {0, 2, 0};  int[] nums011 = {0, 1, 1};  int[] nums002 = {0, 0, 2};  int[] nums300 = {3, 0, 0};  int[] nums210 = {2, 1, 0};  int[] nums201 = {2, 0, 1};  int[] nums120 = {1, 2, 0};  int[] nums111 = {1, 1, 1};  int[] nums102 = {1, 0, 2};  int[] nums030 = {0, 3, 0};  int[] nums021 = {0, 2, 1};  int[] nums012 = {0, 1, 2};  int[] nums003 = {0, 0, 3};    return new Object[] {  //VIP  new Object[] {"Vip",false, 1, 1, 0, 0, nums100, 0, 1, 1, 0, 0},  new Object[] {"Vip",false, 1, 0, 1, 0, nums010, 0, 1, 0, 1, 0},  new Object[] {"Vip",false, 1, 0, 0, 1, nums001, 0, 1, 0, 0, 1},  new Object[] {"Vip",false, 1, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Vip",false, 2, 2, 0, 0, nums200, 0, 1, 2, 0, 0},  new Object[] {"Vip",false, 2, 1, 1, 0, nums110, 0, 1, 1, 1, 0},  new Object[] {"Vip",false, 2, 1, 0, 1, nums101, 0, 1, 1, 0, 1},  new Object[] {"Vip",false, 2, 0, 2, 0, nums020, 0, 1, 0, 2, 0},  new Object[] {"Vip",false, 2, 0, 1, 1, nums011, 0, 1, 0, 1, 1},  new Object[] {"Vip",false, 2, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Vip",false, 2, 0, 0, 2, nums002, 0, 1, 0, 0, 2},  new Object[] {"Vip",false, 2, 0, 0, 1, nums000, 1, 0, 0, 0, 1},  new Object[] {"Vip",false, 2, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Vip",false, 3, 3, 0, 0, nums300, 0, 1, 3, 0, 0},  new Object[] {"Vip",false, 3, 2, 1, 0, nums210, 0, 1, 2, 1, 0},  new Object[] {"Vip",false, 3, 2, 0, 1, nums201, 0, 1, 2, 0, 1},  new Object[] {"Vip",false, 3, 2, 0, 0, nums000, 1, 0, 2, 0, 0},  new Object[] {"Vip",false, 3, 1, 2, 0, nums120, 0, 1, 1, 2, 0},  new Object[] {"Vip",false, 3, 1, 1, 1, nums111, 0, 1, 1, 1, 1},  new Object[] {"Vip",false, 3, 1, 1, 0, nums000, 1, 0, 1, 1, 0},  new Object[] {"Vip",false, 3, 1, 0, 2, nums102, 0, 1, 1, 0, 2},  new Object[] {"Vip",false, 3, 1, 0, 1, nums000, 1, 0, 1, 0, 1},  new Object[] {"Vip",false, 3, 0, 3, 0, nums030, 0, 1, 0, 3, 0},  new Object[] {"Vip",false, 3, 0, 2, 1, nums021, 0, 1, 0, 2, 1},  new Object[] {"Vip",false, 3, 0, 2, 0, nums000, 1, 0, 0, 2, 0},  new Object[] {"Vip",false, 3, 0, 1, 2, nums012, 0, 1, 0, 1, 2},  new Object[] {"Vip",false, 3, 0, 1, 1, nums000, 1, 0, 0, 1, 1},  new Object[] {"Vip",false, 3, 0, 0, 3, nums003, 0, 1, 0, 0, 3},  new Object[] {"Vip",false, 3, 0, 0, 2, nums000, 1, 0, 0, 0, 2},  //Member  new Object[] {"Member",false, 1, 0, 1, 0, nums010, 0, 1, 0, 1, 0},  new Object[] {"Member",false, 1, 0, 0, 1, nums001, 0, 1, 0, 0, 1},  new Object[] {"Member",true, 1, 1, 0, 0, nums100, 0, 1, 1, 0, 0},  new Object[] {"Member",false, 1, 1, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Member",false, 1, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Member",false, 2, 0, 2, 0, nums020, 0, 1, 0, 2, 0},  new Object[] {"Member",false, 2, 0, 1, 1, nums011, 0, 1, 0, 1, 1},  new Object[] {"Member",true, 2, 1, 1, 0, nums110, 0, 1, 1, 1, 0},  new Object[] {"Member",false, 2, 1, 1, 0, nums000, 1, 0, 0, 1, 0},  new Object[] {"Member",false, 2, 0, 1, 0, nums000, 1, 0, 0, 1, 0},  new Object[] {"Member",false, 2, 0, 0, 2, nums002, 0, 1, 0, 0, 2},  new Object[] {"Member",true, 2, 1, 0, 1, nums101, 0, 1, 1, 0, 1},  new Object[] {"Member",false, 2, 1, 0, 1, nums000, 1, 0, 0, 0, 1},  new Object[] {"Member",true, 2, 1, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Member",false, 2, 1, 0, 0, nums000, 1, 0, 0, 0, 0},  new Object[] {"Member",false, 2, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  //Normal  new Object[] {"Normal",false, 1, 0, 0, 1, nums001, 0, 1, 0, 0, 1},  new Object[] {"Normal",false, 1, 0, 0, 0, nums000, 1, 0, 0, 0, 0},  };  }  @Test  @Parameters(method="getParamsForTestSetBooking")  public void testSetBooking(String member\_type, boolean excl\_reward, int numOfRoom,int vipRoom, int deluxeRoom, int standRoom, int[] expectedResult, int times, int size, int vipTimes, int deluxeTimes, int standardTimes ) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standRoom);    User user = mock(User.class);  when(user.getMember\_type()).thenReturn(member\_type);  when(user.getNumRoom()).thenReturn(numOfRoom);  when(user.getExcl\_reward()).thenReturn(excl\_reward);    WaitingList waitingList = mock(WaitingList.class);    Printer printer = mock(Printer.class);    Booking booking = new Booking(user, rMock, waitingList, printer);    booking.setBooking();    assertArrayEquals(expectedResult, booking.getUserRoom());    verify(rMock,times(vipTimes)).bookRoom("vip");  verify(rMock,times(deluxeTimes)).bookRoom("deluxe");  verify(rMock,times(standardTimes)).bookRoom("standard");    int[] result = booking.getUserRoom();  if(result[0] != 0 || result[1] != 0 || result[2] != 0 )  verify(printer).printInfo("Booked Succesfully");  else  verify(printer).printInfo("Added to Waiting List");    // verify(waitingList, times(times)).addWaiting(user);  assertEquals(size, booking.getBookedList().size());  }    private Object[] getParamsForTestSetBookingInvalid() {    return new Object[] {  //VIP  new Object[] {"Vip",false, 0, 0, 0, 3},  new Object[] {"Vip",false, 4, 0, 0, 3},  //Member  new Object[] {"Member",false, 0, 0, 0, 3},  new Object[] {"Member",false, 3, 0, 0, 3},  //Normal  new Object[] {"Normal",false, 0, 0, 0, 0},  new Object[] {"Normal",false, 2, 0, 0, 0},  };  }  @Test (expected = IllegalArgumentException.class)  @Parameters(method="getParamsForTestSetBookingInvalid")  public void testSetBookingInvalid(String member\_type, boolean excl\_reward, int numOfRoom, int vipRoom, int deluxeRoom, int standRoom ) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standRoom);  User user = mock(User.class);  when(user.getMember\_type()).thenReturn(member\_type);  when(user.getNumRoom()).thenReturn(numOfRoom);  when(user.getExcl\_reward()).thenReturn(excl\_reward);    WaitingList waitingList = mock(WaitingList.class);    Printer printer = mock(Printer.class);    Booking booking = new Booking(user, rMock, waitingList, printer);    booking.setBooking();  }  private Object[] getParamsForTestCancelBooking() {  User user1 = new User("kok1","vip", true, 1);  User user2 = new User("kok2","vip", true, 2);  int[] num = {0,0,0};  Booking booking1 = new Booking(user1, num);  Booking booking2 = new Booking(user2, num);  return new Object[] {  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking1); }}, new ArrayList<User>(){{ add(user1); add(user2); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking2); }}, new ArrayList<User>(){{add(user1); add(user2); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking1); add(booking2); }}, new ArrayList<User>()},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking1); add(booking2);}}, new ArrayList<User>(){{ add(user1); add(user2); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(), new ArrayList<User>(){{add(user1); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking1); add(booking2); }}, new ArrayList<User>(){{ add(user2); }}}  };  }  @Test  @Parameters(method="getParamsForTestCancelBooking")  public void testCancelBooking(User user, Booking book, ArrayList<Booking> bookList, ArrayList<User> waitList) {  Room rMock = mock(Room.class);  WaitingList waitingList = new WaitingList();  Printer printer = mock(Printer.class);  Booking booking = new Booking(user, rMock, waitingList, printer);    booking.bookedList = bookList;  if (user.getMember\_type().toLowerCase().equals("vip"))  waitingList.VIP = waitList;  else if (user.getMember\_type().toLowerCase().equals("member"))  waitingList.member = waitList;  else if (user.getMember\_type().toLowerCase().equals("normal"))  waitingList.normal = waitList;  // Perform the action to be tested  booking.cancelBooking(user);  // Assert the expected results  assertFalse(booking.getBookedList().contains(book));  assertFalse(waitingList.getWaiting(user.getMember\_type()).contains(user));  }    private Object[] getParamsForTestCancelBookingInvalid() {  User user1 = new User("kok1","vip", true, 1);  User user2 = new User("kok2","vip", true, 2);    int[] num = {0,0,0};  Booking booking1 = new Booking(user1, num);  Booking booking2 = new Booking(user2, num);  return new Object[] {  new Object[]{user1, booking1, new ArrayList<Booking>(), new ArrayList<User>()},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking2); }}, new ArrayList<User>(){{ add(user2); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(), new ArrayList<User>(){{ add(user2); }}},  new Object[]{user1, booking1, new ArrayList<Booking>(){{ add(booking2); }}, new ArrayList<User>()},  };  }  // Both bookedList and waitingList are empty  @Test (expected = IllegalArgumentException.class)  @Parameters(method="getParamsForTestCancelBookingInvalid")  public void testCancelBookingInvalid(User user, Booking book, ArrayList<Booking> bookList, ArrayList<User> waitList) {  WaitingList waitingList = mock(WaitingList.class);  Room room = new Room();  Printer printer = mock(Printer.class);    Booking booking = new Booking(user, room, waitingList, printer);  booking.bookedList = bookList;  if (user.getMember\_type().toLowerCase().equals("vip"))  waitingList.VIP = waitList;  else if (user.getMember\_type().toLowerCase().equals("member"))  waitingList.member = waitList;  else if (user.getMember\_type().toLowerCase().equals("normal"))  waitingList.normal = waitList;  // Perform the action to be tested  booking.cancelBooking(user);  }  } |

|  |
| --- |
| TestUser.java |
| package assignmenttest;  import static org.junit.Assert.assertEquals;  import org.junit.Test;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  @RunWith(JUnitParamsRunner.class)  public class TestUser {  User user = new User();  @Test  public void testGetSetName() {  String expectedResult = "John";  user.setName(expectedResult);  assertEquals(expectedResult, user.getName());  }    @Test (expected=IllegalArgumentException.class)  public void testSetNameInvalid1() {  user.setName(null);  }    @Test (expected=IllegalArgumentException.class)  public void testSetNameInvalid2() {  user.setName("");  }    @Test  @Parameters({"vip, vip","member,member","normal,normal","VIP,vip","Vip,vip","MEMBER,member","MemBer,member","NORMAL,normal","NormaL,normal"})  public void testGetSetMember\_type(String member\_type, String expected) {  user.setMember\_type(member\_type);  assertEquals(expected, user.getMember\_type());  }    @Test (expected=IllegalArgumentException.class)  public void testSetMember\_typeInvalid1() {  user.setMember\_type(null);  }    @Test (expected=IllegalArgumentException.class)  public void testSetMember\_typeInvalid2() {  user.setMember\_type("");  }    @Test (expected=IllegalArgumentException.class)  @Parameters({"John","Beta","alpha","dEta","Ana","KyS"})  public void testSetMember\_typeInvalid3(String member\_type) {  user.setMember\_type(member\_type);  }    @Test (expected=IllegalArgumentException.class)  @Parameters({"0","5","14","50","583","6589"})  public void testSetMember\_typeInvalid4(String member\_type) {  user.setMember\_type(member\_type);  }    @Test (expected=IllegalArgumentException.class)  @Parameters({"@","#","`","&","+","$","?",","})  public void testSetMember\_typeInvalid5(String member\_type) {  user.setMember\_type(member\_type);  }  @Test  @Parameters({"true","false"})  public void testGetSetExcl\_reward(boolean result) {  boolean expectedResult = result;  user.setExcl\_reward(result);  assertEquals(expectedResult, user.getExcl\_reward());  }    @Test  @Parameters({"1","2","3"})  public void testGetSetNumRoom(int numRoom) {  int expectedResult = numRoom;  user.setNumRoom(numRoom);  assertEquals(expectedResult, user.getNumRoom());  }    @Test(expected=IllegalArgumentException.class)  @Parameters({"-12","50","100","1000"})  public void testSetNumRoomInvalid1(int numRoom) {  user.setNumRoom(numRoom);  }    @Test(expected=IllegalArgumentException.class)  @Parameters({"0","4"})  public void testSetNumRoomInvalid2(int numRoom) {  user.setNumRoom(numRoom);  }    } |

|  |
| --- |
| TestWaitingList.java |
| package assignmenttest;  import static org.junit.Assert.assertFalse;  import static org.junit.Assert.assertTrue;  import static org.mockito.Mockito.mock;  import static org.mockito.Mockito.when;  import java.io.BufferedReader;  import java.io.File;  import java.io.FileReader;  import java.util.ArrayList;  import org.junit.Test;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  @RunWith(JUnitParamsRunner.class)  public class TestWaitingList {  WaitingList wl = new WaitingList();  private ArrayList<User> userList = new ArrayList<User>();  private File userFile = new File("User.txt");  public ArrayList<User> getUserList() {  return userList;  }  public void checkFileExist() throws Exception {  if (!userFile.exists()) {  userFile.createNewFile();  }  }  public void loadFile() throws Exception {  try (BufferedReader userReader = new BufferedReader(new FileReader(userFile))) {  String line;  userList.clear();  while ((line = userReader.readLine()) != null) {  String[] fields = line.split(",");  if (fields.length == 4) {  String name = fields[0].trim();  String memberType = fields[1].trim();  boolean exclRewards = Boolean.parseBoolean(fields[2].trim());  int numRoom = Integer.parseInt(fields[3].trim());  User user = new User(name, memberType, exclRewards, numRoom);  userList.add(user);  } else {  // Handle invalid lines or missing fields  System.out.println("Invalid line: " + line);  }  }  }  }  private Object[] getParamsForTestAddWaiting() {  try {  checkFileExist();  loadFile();  } catch (Exception e) {  e.printStackTrace();  }    ArrayList<User> users = getUserList();    return new Object[] {  new Object[]{users.get(0),0,0,0},  new Object[]{users.get(1),0,0,0},  new Object[]{users.get(2),0,0,0},  new Object[]{users.get(3),0,0,0},  new Object[]{users.get(4),0,0,0},  new Object[]{users.get(5),0,0,0},  new Object[]{users.get(6),0,0,0},  new Object[]{users.get(7),0,0,0},  };  }  @Test  @Parameters(method="getParamsForTestAddWaiting")  public void testAddWaiting(User user, int vipRoom, int deluxeRoom, int standardRoom) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standardRoom);    User userMock = mock(User.class);  when(userMock.getMember\_type()).thenReturn(user.getMember\_type());  when(userMock.getNumRoom()).thenReturn(user.getNumRoom());  when(userMock.getExcl\_reward()).thenReturn(user.getExcl\_reward());    Printer printer = mock(Printer.class);    Booking booking = new Booking(userMock, rMock, wl, printer);    booking.setBooking();  assertTrue(wl.getWaiting(userMock.getMember\_type()).contains(userMock));  }  @Test  @Parameters(method="getParamsForTestAddWaiting")  public void testWaitingListIntegration(User user, int vipRoom, int deluxeRoom, int standardRoom) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standardRoom);  Printer printer = mock(Printer.class);    Booking booking = new Booking(user, rMock, wl, printer);  booking.setBooking();  assertTrue(wl.getWaiting(user.getMember\_type()).contains(user));  booking.cancelBooking(user);  assertFalse(wl.getWaiting(user.getMember\_type()).contains(user));  }  private Object[] getParamsForTestAddWaitingInvalid() {  return new Object[] {  new Object[] { new User("kok1","hi", true, 1) },  };  }  @Test (expected=IllegalArgumentException.class)  @Parameters(method="getParamsForTestAddWaitingInvalid")  public void testAddWaitingInvalid(User user) {  wl.addWaiting(user);  }    private Object[] getParamsForTestRemoveWating() {  try {  checkFileExist();  loadFile();  } catch (Exception e) {  e.printStackTrace();  }    ArrayList<User> users = getUserList();    return new Object[] {  new Object[]{users.get(0), new ArrayList<User>(){{ add(users.get(0)); }}},  new Object[]{users.get(1), new ArrayList<User>(){{ add(users.get(1)); }}},  new Object[]{users.get(2), new ArrayList<User>(){{ add(users.get(2)); }}},  new Object[]{users.get(3), new ArrayList<User>(){{ add(users.get(3)); }}},  new Object[]{users.get(4), new ArrayList<User>(){{ add(users.get(4)); }}},  new Object[]{users.get(5), new ArrayList<User>(){{ add(users.get(5)); }}},  new Object[]{users.get(6), new ArrayList<User>(){{ add(users.get(6)); }}},  new Object[]{users.get(7), new ArrayList<User>(){{ add(users.get(7)); }}},  };  }  @Test  @Parameters(method="getParamsForTestRemoveWating")  public void testRemoveWaiting(User user, ArrayList<User> arrList) {  if (user.getMember\_type().toLowerCase().equals("vip"))  wl.VIP = arrList;  else if (user.getMember\_type().toLowerCase().equals("member"))  wl.member = arrList;  else if (user.getMember\_type().toLowerCase().equals("normal"))  wl.normal = arrList;  wl.removeWaiting(user);  assertFalse(wl.getWaiting(user.getMember\_type()).contains(user));  }  private Object[] getParamsForTestRemoveWatingInvalid() {  User user1 = new User("kok1","vip", true, 1);  User user2 = new User("kok2","hi", true, 2);  return new Object[] {  new Object[]{user1, new ArrayList<User>()},  new Object[]{user2, new ArrayList<User>()},  };  }  @Test (expected=IllegalArgumentException.class)  @Parameters(method="getParamsForTestRemoveWatingInvalid")  public void testRemoveWaitingInvalid(User user, ArrayList<User> arrList) {  if (user.getMember\_type().toLowerCase().equals("vip"))  wl.VIP = arrList;  else if (user.getMember\_type().toLowerCase().equals("member"))  wl.member = arrList;  else if (user.getMember\_type().toLowerCase().equals("normal"))  wl.normal = arrList;  wl.removeWaiting(user);  }  } |
| TestBookingIntegration.java |
| package assignmenttest;  import static org.junit.Assert.assertArrayEquals;  import static org.junit.Assert.assertEquals;  import static org.junit.Assert.assertFalse;  import static org.mockito.Mockito.mock;  import static org.mockito.Mockito.verify;  import static org.mockito.Mockito.times;  import static org.mockito.Mockito.when;  import java.util.ArrayList;  import org.junit.Test;  import org.junit.runner.RunWith;  import junitparams.JUnitParamsRunner;  import junitparams.Parameters;  @RunWith(JUnitParamsRunner.class)  public class TestBookingIntegration {  public Object[] getParamsForTestBooking() {  ArrayList<Booking> expectedArrayListAfterCancel = new ArrayList<>();  ArrayList<User> expectedWaitingListAfterCancel = new ArrayList<>();  User vipUser1 = new User("kok4","vip", false, 1);  User vipUser2 = new User("kok5","vip", false, 2);  User vipUser3 = new User("kok6","vip", false, 3);    User memberUser1 = new User("ys1","member", true, 1);  User memberUser2 = new User("ys2","member", true, 2);  User memberUser2\_2 = new User("ys2","member", true, 2);  User memberUser2\_3 = new User("ys2","member", true, 2);  User memberUser3 = new User("ys3","member", false, 1);  User memberUser4 = new User("ys4","member", false, 2);    User normalUser = new User("bernard","normal", false, 1);  //num of room booked by user  int[] nums100 = {1, 0, 0};  int[] nums010 = {0, 1, 0};  int[] nums001 = {0, 0, 1};  int[] nums000 = {0, 0, 0};  int[] nums200 = {2, 0, 0};  int[] nums110 = {1, 1, 0};  int[] nums101 = {1, 0, 1};  int[] nums020 = {0, 2, 0};  int[] nums011 = {0, 1, 1};  int[] nums002 = {0, 0, 2};  int[] nums300 = {3, 0, 0};  int[] nums210 = {2, 1, 0};  int[] nums201 = {2, 0, 1};  int[] nums120 = {1, 2, 0};  int[] nums111 = {1, 1, 1};  int[] nums102 = {1, 0, 2};  int[] nums030 = {0, 3, 0};  int[] nums021 = {0, 2, 1};  int[] nums012 = {0, 1, 2};  int[] nums003 = {0, 0, 3};    return new Object[] {  //VIP  new Object[] {vipUser1, 1, 0, 0, nums100, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser1, 0, 1, 0, nums010, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser1, 0, 0, 1, nums001, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser1, 0, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 2, 0, 0, nums200, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 1, 1, 0, nums110, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 1, 0, 1, nums101, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 1, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 0, 2, 0, nums020, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 0, 1, 1, nums011, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 0, 1, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 0, 0, 2, nums002, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser2, 0, 0, 1, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 3, 0, 0, nums300, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 2, 1, 0, nums210, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 2, 0, 1, nums201, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 2, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 1, 2, 0, nums120, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 1, 1, 1, nums111, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 1, 1, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 1, 0, 2, nums102, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 1, 0, 1, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 3, 0, nums030, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 2, 1, nums021, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 2, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 1, 2, nums012, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 1, 1, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 0, 3, nums003, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {vipUser3, 0, 0, 2, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  //Member  new Object[] {memberUser3, 0, 1, 0, nums010, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser3, 0, 0, 1, nums001, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser1, 1, 0, 0, nums100, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser3, 1, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser3, 0, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 0, 2, 0, nums020, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 0, 1, 1, nums011, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser2, 1, 1, 0, nums110, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 1, 1, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 0, 1, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 0, 0, 2, nums002, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser2\_2, 1, 0, 1, nums101, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 1, 0, 1, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {memberUser4, 0, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  //Normal  new Object[] {normalUser, 0, 0, 1, nums001, 1, expectedArrayListAfterCancel, expectedWaitingListAfterCancel},  new Object[] {normalUser, 0, 0, 0, nums000, 0, expectedArrayListAfterCancel, expectedWaitingListAfterCancel}  };  }  @Test  @Parameters(method="getParamsForTestBooking")  public void testBookingIntegration(User user, int vipRoom, int deluxeRoom, int standardRoom, int[] numOfRoomBooked, int size, ArrayList<Booking> expectedArrayListAfterCancel,ArrayList<User> expectedWaitingListAfterCancel) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standardRoom);    WaitingList waitingList = new WaitingList();    Printer printer = mock(Printer.class);    Booking booking = new Booking(user, rMock, waitingList, printer);    boolean initiateExcl\_reward = user.getExcl\_reward();  booking.setBooking();    if (user.getMember\_type().toLowerCase().equals("member") && initiateExcl\_reward == true && booking.getUserRoom()[0] == 1)  assertFalse(user.getExcl\_reward());  assertArrayEquals(numOfRoomBooked, booking.getUserRoom());  assertEquals(size, booking.getBookedList().size());  // Perform the action to be tested  booking.cancelBooking(user);  // Assert the expected results  assertEquals(expectedArrayListAfterCancel, booking.getBookedList());  assertEquals(expectedWaitingListAfterCancel, waitingList.getWaiting(user.getMember\_type()));  }  public Object[] getParamsForTestBookingInvalid() {  User vipUser1 = new User("kok4","vip", false, 0);  User vipUser2 = new User("kok6","vip", false, 4);    User memberUser1 = new User("ys3","member", false, 0);  User memberUser2 = new User("ys4","member", false, 3);    User normalUser1 = new User("bernard","normal", false, 0);  User normalUser2 = new User("bernard","normal", false, 2);  User invalidUser = new User("bernard", "hi", false, 3);  return new Object[] {  //VIP  new Object[] {vipUser1, vipUser2, 0, 0, 3},  new Object[] {vipUser2, vipUser1, 0, 0, 3},  //Member  new Object[] {memberUser1, memberUser2, 0, 0, 3},  new Object[] {memberUser2, memberUser1, 0, 0, 3},    //Normal  new Object[] {normalUser1, normalUser2, 0, 0, 0},  new Object[] {normalUser2, normalUser1, 0, 0, 0},  new Object[] {invalidUser, invalidUser, 0, 0, 0,}    };  }    @Test (expected = IllegalArgumentException.class)  @Parameters(method="getParamsForTestBookingInvalid")  public void testBookingIntegrationInvalid(User user1, User user2 ,int vipRoom, int deluxeRoom, int standardRoom) {  Room rMock = mock(Room.class);  when(rMock.getVip()).thenReturn(vipRoom);  when(rMock.getDeluxe()).thenReturn(deluxeRoom);  when(rMock.getStandard()).thenReturn(standardRoom);    WaitingList waitingList = new WaitingList();    Printer printer = mock(Printer.class);    Booking booking = new Booking(user1, rMock, waitingList, printer);    booking.setBooking();  // Perform the action to be tested  booking.cancelBooking(user2);  }  } |