

CSIT 5100: Object Oriented Programming and Testing

2016 Fall – Written Assignment Number 1

October, 17th 2016.

Introduction to HMS project:

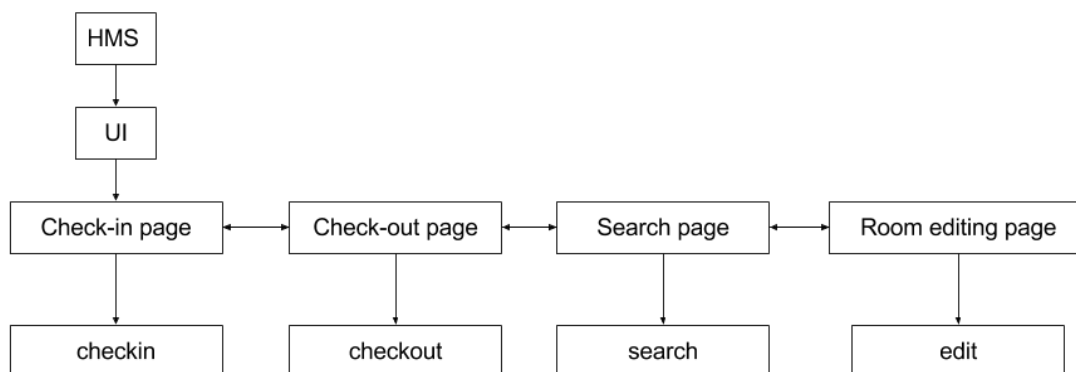
This report demonstrates the testing information of the Hotel Management System (shortened as HMS in the report). The HMS is a management system for hotels that can perform the check-in and check-out of customers, can change the rate of rooms or enable discounts on the room prices. The system does not take care of computing the final pricing for the customer stay. However, the system also contains a graphical interface based on different customizable (user can change the color) panels.

The first part of this report presents an introduction to the structure of the HMS system to be tested. The second part explain how tests have been designed and implemented. The third part discusses in detail the test accuracy obtained with statement coverage and branching coverage as estimation criteria. The fourth part will then list all the possible issues found in the source code, before going further by discussing possible improvements in the last part.

Introduction to HMS project:	1
1- Basic Structure of the HMS system:	3
a. Graphical interface description	3
b. Project structure description	4
2- Implementation of test procedures:	5
a. What should be tested?	5
b. Unit testing and test case structure	5
c. Methodology used for the implementation of test cases	6
3- Discussion of the test accuracy:	8
a. Statement coverage	8
b. Branching coverage	9
c. Discussion of the uncovered statements and branchings	10
i) Main package	10
ii) Model package	11
iii) Command package	12
iv) GUI package	13
4- Testing results:	22
a. List of problems found with unsuccessful testing	22
b. Other problems found while inspecting the source code	24
c. Improvement suggestions	24
5. Remaining work:	26
a. Unsolved problems faced during the testing of the project	26
b. Improving the testing with other criteria	26
Conclusion:	27
Appendix:	28
Test class headers	28
References	30

1- Basic Structure of the HMS system:

a. Graphical interface description



Pic: 1 - Informal description of the user interface

The HMS system consists of a single page that can contain four different panels, but only once is displayed at the same time. Switching from one panel to the other is done by using a check box. Each of these panels is supposed to fulfil one specific function: performing a check-in, performing a check-out, searching among the rooms used by customer or changing the rate and/or discount of the available rooms. By default, at opening the graphical interface is displaying the panel for check-in.

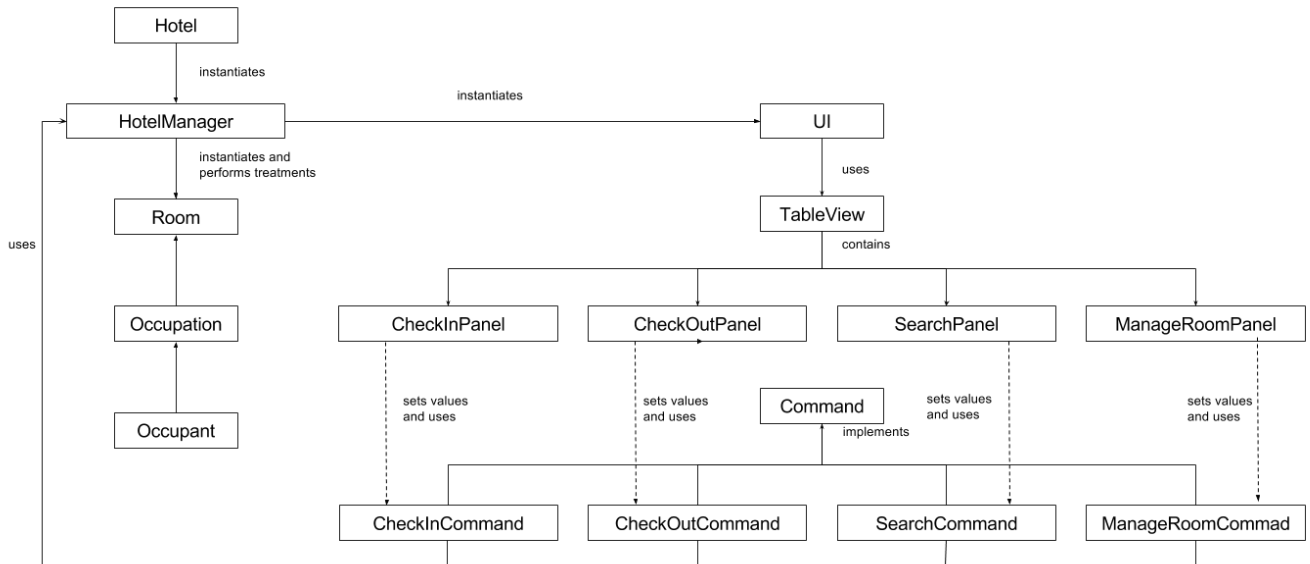
The check-in panel shows a table of the available rooms in the hotel and their related information (room number and floor, room type, capacity, rate and data service activation). It also contains fields in order to perform a check-in. To do so, the user should add a name, a company name and a check-in date. The user should also select a room and a type of stay for the customer among “standard” and “business”. If needed, the data service can be activated for non-standard rooms and an ethernet address should be added. Once everything is correctly completed, the user should use the check-in button.

The check-out panel shows a table of the rooms currently used by customers. In order to perform a check-out, the user should select one of these rooms, input a date of checkout and use the checkout button.

The search panel enables the user to make requests among the currently used rooms. The user has to select a type of field among ID, Company, Name and Type and complete the search field. The request can then be performed by using the search button and the corresponding results are displayed in a table showing rooms and their related occupant information.

The room management panel shows the list of currently available rooms with detailed information. The user should first select a room and press the edit button. He can then either cancel his selection or insert a new rate or discount value and change it by pressing the update button.

b. Project structure description



Pic: 2 - Informal description of the project structure

This informal description shows the different classes involved in the project and some information on how they are related to each other. It is important to note that the relation between a *Room* and a customer (modeled as an *Occupant* here) is made using the object *Occupation*. The four different panels will then be indirectly dealing with these classes through their related command, at first, and the *HotelManager*. Understanding this basic structure is really important in order to perform accurate tests.

2- Implementation of test procedures:

a. What should be tested?

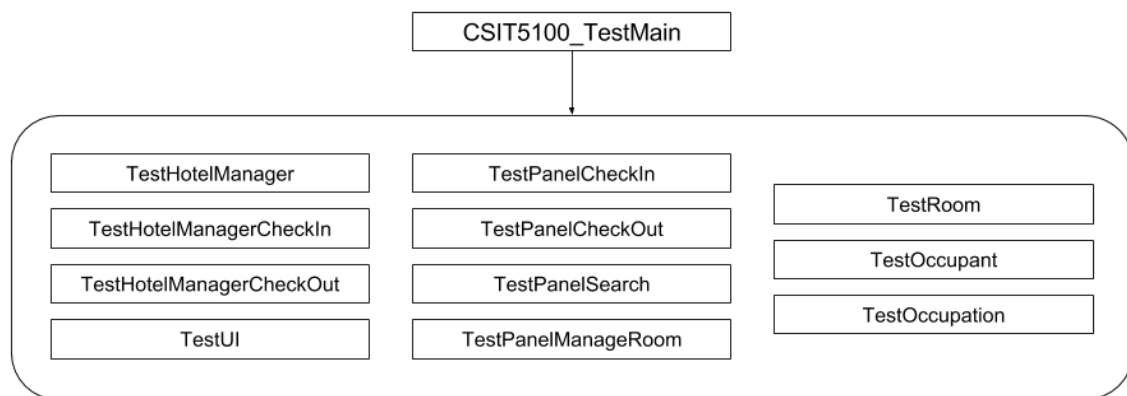
The purpose of our test procedure will be to test the correctness of the HMS when fulfilling his functions previously described.

As a consequence, we need to test the classes referring to the inner logic of the system including the *HotelManager* and its related classes (*hms.model* and *hms.main* packages), but also the graphical interface and all its related panels (*hms.gui* package). Commands don't really need to be tested individually since they have a pretty basic structure and will be covered while performing tests on panels.

b. Unit testing and test case structure

Test cases are constructed in Junit format using the JUnit4 framework. They are stored in the “/test/” folder of the project. All tests have been designed in order to be runnable automatically without user intervention. Eclemma has been used to calculate the source code statement and branching coverage and validate the test accuracy.

The tests cases for the HMS are implemented in different test classes which are all registered in the test suite class *CSIT5100_TestMain* as stated on the structure below.



Pic: 3 - Test case structure

Each of the previously targeted classes has been tested in one particular test class except the *HotelManager* class that has been tested in three different test classes. The classes *TestHotelManagerCheckIn* and *TestHotelManagerCheckOut* are performing tests focusing on the behavior of the *checkIn* and *checkOut* methods of the *HotelManager*. The class *TestHotelManager* is testing all other methods. This split in three different classes has been motivated by the use of parameterized test classes to cover all possible scenarios and by an improved readability of the test code.

c. Methodology used for the implementation of test cases

The target of this test implementation is to obtain a dynamic analysis of the software to detect potential faults that can be run automatically without user intervention.

All the tests have then been designed in order to be runnable automatically. This implies two design consequences. On the one hand, all tests are using assertions in order to test outputs and expected behaviour of the methods and classes defined in the software. This way, the user doesn't need to think on how to interpret the result of the tests. He can easily distinguish the failing tests from the passing ones. On the other hand, no intervention is also needed considering the input. All input values are defined inside the test classes. This is also the case for graphical interfaces since the user interactions are mocked in the code.

When implementing the tests, I followed a bottom-up approach. I started by testing leaf classes such as *Room* or *HotelManager*. I used parameterized classes in order to cover all possible cases I could notice in the source code (white box testing) while avoiding code redundancy when writing the test cases. This has allowed me to improve my understanding of the software structure and of the possible cases. It also helped me to get sure that these classes were well defined before trying to test more complex classes that were calling these classes. In fact, high level functions can sometimes have many small bugs that together fake a good behaviour. Thus, I tested the panel classes in a second time.

```
@Parameters(name = "{index}:{0},{1},{2},{3},{4},{5},{6},{7},{8}")
public static Collection<Object[]> parameters() {
    return Arrays.asList(new Object[][] {
        // Well formatted data sets
        {"X12345678", "Gates", "Standard", "Microsoft", dateIn, "No", "", new Integer(5), false},
        {"X12345678", "Gates", "Business", "Microsoft", dateIn, "Yes", "01:23:45:67:89:ab", new Integer(1), false}, // Presidential room
        {"X12345678", "Gates", "Business", "Microsoft", dateIn, "Yes", "01:23:45:67:89:ab", new Integer(3), false}, // Executive room

        // Data sets with not allowed null inputs
        {null, "Gates", "Business", "Microsoft", dateIn, "No", "", new Integer(1), true},
        {"X12345678", null, "Business", "Microsoft", dateIn, "No", "", new Integer(1), true},
        {"X12345678", "Gates", null, "Microsoft", dateIn, "No", "", new Integer(1), true},
        {"X12345678", "Gates", "Business", null, dateIn, "No", "", new Integer(1), true},
        {"X12345678", "Gates", "Standard", "Microsoft", null, "No", "", new Integer(1), true},
        {"X12345678", "Gates", "Standard", "Microsoft", dateIn, "No", null, new Integer(1), true},

        // Data sets with not allowed empty inputs
        {"", "Gates", "Business", "Microsoft", dateIn, "No", "", 1, true},
        {"X12345678", "", "Business", "Microsoft", dateIn, "No", "", 1, true},
        {"X12345678", "Gates", "Business", "", dateIn, "No", "", 1, true},

        // Data set with inconsistent information
        {"X12345678", "Gates", "Standard", "Microsoft", dateIn, "Yes", "01:23:45:67:89:ab", 1, true},
        {"X12345678", "Gates", "Business", "Microsoft", dateIn, "Yes", "01:23:45:67:89:ab", 5, true}, // Standard room

        // Data set with invalid room type
        {"X12345678", "Gates", "Luxuuous", "Microsoft", dateIn, "No", "", 1, true},

        // Data set with invalid ID
        {"X12345", "Gates", "Business", "Microsoft", dateIn, "No", "", 1, true},

        // Data set with invalid ethernetAddress
        {"X12345678", "Gates", "Business", "Microsoft", dateIn, "Yes", "XXX", 1, true},

        // Data set with invalid date
        {"X12345678", "Gates", "Business", "Microsoft", "INVALID DATE", "Yes", "01:23:45:67:89:ab", 1, true}
    });
}
```

Pic: 4 - Example of parameters for the test cases of *CheckInPanel*

All methods outputs have been tested. Printing have been tested using `ByteArrayOutputStream` to read the messages intended to be printed in the console and perform some assertions on them.

In order to approve my coverage, I sometimes have manipulated data directly from the existing command or in the panel in order to get some specific errors. This has not been done exhaustively, but is discussed in the uncovered statements when needed. The following piece of code shows tests made when replacing the current selected room by a null element, which isn't possible from the graphical interface.

```
@Test
public void test_getCommande_nullRoom_inTable(){

    // Selecting a room to edit
    JTable table = manageRoomPanel.editableRoomsTable;
    table.setRowSelectionInterval(tableLine, tableLine);

    // Replacing the selected room by a null element
    manageRoomPanel.editableRooms[tableLine] = null;

    // Should return null whatever the value requested is
    TableModel model = manageRoomPanel.editableRoomsTable.getModel();
    assertNull(model.getValueAt(tableLine, 0));
    assertNull(model.getValueAt(tableLine, 1));
    assertNull(model.getValueAt(tableLine, 2));
    assertNull(model.getValueAt(tableLine, 3));
    assertNull(model.getValueAt(tableLine, 4));
    assertNull(model.getValueAt(tableLine, 5));
}
```

Pic: 5 - Testing the getCommand from ManageRoomPanel with null room























While implementing my test cases, I tried to use a consistent syntax and meaningful names that are describing the test cases objectives. I also tried to add as many comments as necessary to make it readable and understandable by other users. This also includes the creation of commented headers for each class and each function in order to explain its objectives and give the most relevant information on what is performed.

A table containing these header descriptions for all implemented test classes has been added to the appendix.

3- Discussion of the test accuracy:

a. Statement coverage

The test cases can achieve 97,1% statement coverage of the source code of HMS. Specific coverage for each Java class is as follow:
















Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
▲ HMS	 97,5 %	9 596	244	9 840
▸ test	 97,8 %	5 457	120	5 577
▲ src	 97,1 %	4 139	124	4 263
▲ hms.model	 100,0 %	254	0	254
▸ Occupant.java	 100,0 %	46	0	46
▸ Occupation.java	 100,0 %	27	0	27
▸ Room.java	 100,0 %	181	0	181
▲ hms.command	 98,4 %	380	6	386
▸ ManageRoomCommand.java	 100,0 %	13	0	13
▸ SearchCommand.java	 100,0 %	47	0	47
▸ CheckInCommand.java	 98,2 %	162	3	165
▸ CheckOutCommand.java	 98,1 %	158	3	161
▲ hms.main	 98,3 %	460	8	468
▸ HotelManager.java	 98,9 %	445	5	450
▸ Hotel.java	 83,3 %	15	3	18
▲ hms.gui	 96,5 %	3 045	110	3 155
▸ TableView.java	 100,0 %	43	0	43
▸ ManageRoomPanel.java	 99,5 %	834	4	838
▸ UI.java	 98,8 %	398	5	403
▸ SearchPanel.java	 96,6 %	449	16	465
▸ CheckInPanel.java	 94,1 %	855	54	909
▸ CheckOutPanel.java	 93,8 %	466	31	497

Pic: 6 - Statement coverage of the project

We can see that the test cases have been reaching almost all the different statements of the source code. Even if some classes, like the commands, have not been specifically tested, they are still almost completely covered by our test cases. A few infeasible statements explaining some of the not covered statements will be described later.

b. Branching coverage

The test cases can achieve 90,5% branch coverage of the source code of HMS. Specific coverage for each Java class is as follow:

Element	Coverage	Covered Branches	Missed Branches	Total Branches
▲ HMS	 90,8 %	267	27	294
▸ test	 92,3 %	48	4	52
▲ src	 90,5 %	219	23	242
▲ hms.model	 100,0 %	16	0	16
▸ Occupant.java		0	0	0
▸ Occupation.java		0	0	0
▸ Room.java	 100,0 %	16	0	16
▲ hms.main	 98,9 %	87	1	88
▸ Hotel.java		0	0	0
▸ HotelManager.java	 98,9 %	87	1	88
▲ hms.gui	 84,3 %	113	21	134
▸ TableView.java		0	0	0
▸ UI.java		0	0	0
▸ ManageRoomPanel.java	 93,8 %	30	2	32
▸ CheckOutPanel.java	 85,7 %	24	4	28
▸ SearchPanel.java	 83,3 %	20	4	24
▸ CheckInPanel.java	 78,0 %	39	11	50
▲ hms.command	 75,0 %	3	1	4
▸ ManageRoomCommand.java		0	0	0
▸ SearchCommand.java		0	0	0
▸ CheckOutCommand.java	 100,0 %	2	0	2
▸ CheckInCommand.java	 50,0 %	1	1	2

Pic: 7 - Branching coverage of the project

As with the statement coverage, we can see that the test cases have been reaching almost all the different branches of the code. A few infeasible branching explaining some of the missed branches will be described in the following part.

c. Discussion of the uncovered statements and branchings

The following section of the report discusses the statements and branching that were not covered by the tests. It focuses on highlighting the infeasible ones and on explaining why others were not covered. The discussion of some of the infeasible statements also helps us finding potential error states in the software.

This section doesn't show an exhaustive list of the infeasible statements of the program since, as explained previously, some of them have been tested by changing the panel or command attributes which is not possible from the user interface. However, some of them are still discussed here and we can observe some similarities between them. For instance, when a statement is infeasible in one of the panel classes, it is very likely to find some infeasible ones in a similar location in the other panel classes, since they are all based on a similar structure.

i) Main package

HotelManager:

The class hotel manager is covered at 98,9% and only 5 statements and 1 branch are not covered. These statements correspond to an exception that is not caught during the tests. This exception is related to the parsing of the xml file of the hotel that is well defined in the project. The branch and the other statements are unfeasible and are due to a condition that cannot be reached.

```
if (ID == null || name == null || type == null || company == null || checkInDate == null || ethernetAddress == null || room == null)
    return "No input can be null";
}
else if (ID.equals("") || name.equals("") || company.equals("")) {
    return "ID, name, and company cannot be empty";
}
else if (room == null) {
    return "No room is selected";
}
```

Pic: 8 - Uncovered statement in HotelManager

This branch corresponds to the condition of a null room. However, this situation is checked in the first line and then a return is called. Thus, we have **infeasible statements**.

Hotel:

In the class hotel, only three statements are not covered.

```
public static void main(String[] args) {
    new HotelManager();
}
```

Pic: 9 - Uncovered statement in Hotel

These statements correspond to the main function of the program that we don't need to test.

ii) Model package

Occupant:

The class has been fully covered, no feasible statements or branches.

Occupation:

The class has been fully covered, no feasible statements or branches.

Room:

The class has been fully covered, no feasible statements or branches.

iii) Command package

CheckInCommand:

Only 3 statements are not covered and they are related to the following function.

```
public Date getCurrentDate() {  
    try  
    {  
        return new SimpleDateFormat("dd-MM-yyyy").parse(new SimpleDateFormat("dd-MM-yyyy").format(new GregorianCalendar().getTime()));  
    }  
    catch (ParseException e)  
    { // Never happens  
        return null;  
    }  
}
```

Pic: 10 - Uncovered statement in CheckInCommand

As stated in the comments, I never experienced having a ParseException in this case. Even if it might be useless, it is not a big issue keeping this exception catcher.

CheckOutCommand:

Only 3 statements are not covered in the method *getCurrentDate* for the exact same reason as in CheckInCommad.

```
public Date getCurrentDate() {  
    try  
    {  
        return new SimpleDateFormat("dd-MM-yyyy").parse(new SimpleDateFormat("dd-MM-yyyy").format(new GregorianCalendar().getTime()));  
    }  
    catch (ParseException e)  
    { // Never happens  
        return null;  
    }  
}
```

Pic: 11 - Uncovered statement in CheckOutCommand

ManageRoomCommand:

The class has been fully covered, no feasible statements or branches.

SearchCommand:

The class has been fully covered, no feasible statements or branches.

iv) GUI package

UI:

The class UI is covered at 98,8% and only five statements are not covered.

- UI (constructor)

```
// Add window Listener
addWindowListener(new WindowAdapter() {
    public void windowClosing(WindowEvent e) {
        //store things to database to do
        System.exit(0);
    }
});
```

Pic: 12 - Uncovered statement "exit" in UI

I don't know if this uncovered statement is an infeasible one or not. I assumed it was the implementation of the exit button of the main UI and it could be tested easily on the graphical interface. However while testing the coverage of the code as a java application (which means by running the tool), I have not been able to reach this statement. It might then be an infeasible statement. Since I don't really know how to test it, I am not sure of my conclusion.

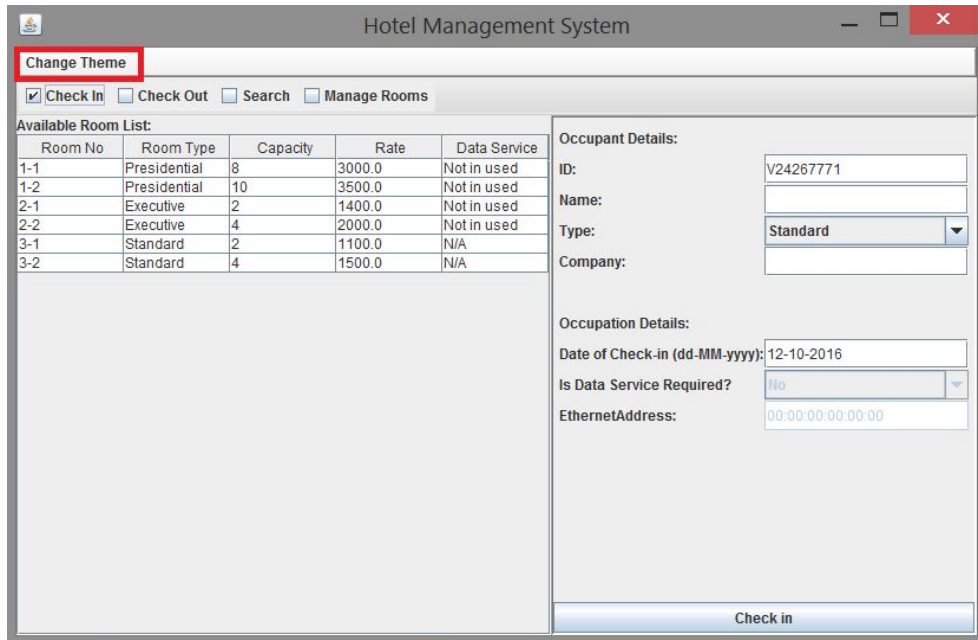
However, a TODO has been mentioned in the comments and has not been implemented which should be either fixed or removed.

- constructMenu

```
lookAndFeelMenu.add(new AbstractAction("Metal", null) {
    public void actionPerformed(ActionEvent e) {
        try {
            UIManager.setLookAndFeel("javax.swing.plaf.metal.MetalLookAndFeel");
            SwingUtilities.updateComponentTreeUI(UI.this);
            setVisible(false); setVisible(true);
        }
        catch (Exception ex) { }
    }
});
lookAndFeelMenu.add(new AbstractAction("Motif", null) {
    public void actionPerformed(ActionEvent e) {
        try {
            UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
            SwingUtilities.updateComponentTreeUI(UI.this);
            setVisible(false); setVisible(true);
        }
        catch (Exception ex) { }
    }
});
```

Pic: 13 - Uncovered statement "change theme" in UI

I tried to test the change "change theme" button which is the behavior implemented in the piece of code available above.



Pic: 14 - Graphical interface showing the button "change theme" in the GUI

I wanted to test it by using the following piece of code. I have been able to test that the default theme is the metal one, as expected. However I have not been able to change it properly.

```
// Testing to change the menu color
assertEquals(2, ui.lookAndFeelMenu.getItemCount());
System.out.println(UIManager.getLookAndFeel());
assertTrue(UIManager.getLookAndFeel().toString().contains("metal"));

/* ui.lookAndFeelMenu.getItem(1).doClick();
System.out.println(UIManager.getLookAndFeel());
assertTrue(UIManager.getLookAndFeel().toString().contains("motif"));

ui.lookAndFeelMenu.getItem(0).doClick();
System.out.println(UIManager.getLookAndFeel());
assertTrue(UIManager.getLookAndFeel().toString().contains("metal"));

// Closing the window
WindowListener[] wListeners = ui.getWindowListeners();
assertEquals("Wrong number of windows listeners", 1, wListeners.length);

*/
ui.dispose();
```

Pic: 15 - Test code rising errors when testing "change theme" in UI

Even if I could see while running the test, that the theme of the UI was changing, the commented piece of code is rising exceptions. This can be seen below. The printing are stating that the theme has been changed, but an exception has been raised right after.

```
[The CDE/Motif Look and Feel - com.sun.java.swing.plaf.motif.MotifLookAndFeel]
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
```

Pic: 16 - Error raised during tests of "change theme" in UI

Fixing my test case would be something needed to improve the project.

CheckInPanel:

In *ChekInPanel*, 54 instructions and 11 branches are not covered.

- *AddOccupantInfo*:

```
String result = command.checkIn();
if (result.equals("Success")) {
    getCommandValues();
} else {
    JOptionPane.showMessageDialog(container, result,
        "Input Error", JOptionPane.INFORMATION_MESSAGE);
    System.err.print("Input Error: " + result);
}
```

Pic: 17 - Infeasible statement in CheckInPanel

The error printing message here is **infeasible**. All inputs are checked, returning an error is they are not well formatted. This way, having an unsuccessful check-in is actually not possible here.

- *GetCommandValue*

```
Room selectedRoom = command.getSelectedRoom();
if (selectedRoom != null) {
    int index = list.indexOf(selectedRoom);
    availableRoomTable.setRowSelectionInterval(index, index);
}
```

Pic: 18 - Infeasible branching in CheckInPanel

These statement cannot be reached. These lines are called inside the *getCommandValue* method. This method is called from the *actionListner* of *addOccupantInfo* (as you can see it on the previous code snippet discussed). It means that this test is performed after having checked all the data input from the interface (including a null selected room), after having updated the command value and after having successfully performed the check-in. This statement could be only achieved by changing the value of the selected room directly in the command. However these statement are anyway **infeasible** from the graphical interface.

- *updateDataServiceRequired*

```
dataServiceRequiredBox.setEnabled(true);
if (command.isDataServiceRequired()) {
    if (dataServiceRequiredBox.getSelectedItem().equals("No")) {
        dataServiceRequiredBox.setSelectedItem("Yes");
    }
    ethernetAddressField.setEnabled(true);
} else {
    if (dataServiceRequiredBox.getSelectedItem().equals("Yes")) {
        dataServiceRequiredBox.setSelectedItem("No");
    }
    ethernetAddressField.setEnabled(false);
}
```


Pic: 19 - Uncovered branchings in CheckInPanel

These statements are used when the front-end user is performing many booking in a row with different *dataRequired* parameters. However, between each of my unit tests, I am using a new *CheckInPanel* object. That's why I haven't reached these statements. However, since the logic is pretty simple and looks correctly implemented, I am not sure it needed to be tested by adding a new test case performing two or three check-in a row. Especially, since *CheckInPanel* is tested using parameterised class, it would generate a lot of new tests and increase the test time. This is why I haven't done it.

- *getOccupantInfoHelper*:

```
typeField.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
        command.setType((String) typeField.getSelectedItem());
        updateDataServiceRequired();
    }
});
typeField.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (UIManager.getLookAndFeel().getName().equals("Windows")) {
            JFrame f = (JFrame) typeField.getTopLevelAncestor();
            if (f != null) {
                f.setVisible(false); f.setVisible(true);
            }
        }
    }
});
```

Pic: 20 - Uncovered "change theme" statements in CheckInPanel

These uncovered statements might be related to the unsatisfying testing of the "change theme" features. See comments in UI for more details.

```
dataServiceRequiredBox.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (UIManager.getLookAndFeel().getName().equals("Windows")) {
            JFrame f = (JFrame) dataServiceRequiredBox.getTopLevelAncestor();
            if (f != null) {
                f.setVisible(false); f.setVisible(true);
            }
        }
    }
});
```

Pic: 21 - Uncovered "change theme" statements in CheckInPanel

As before, these uncovered statements might be related to the unsatisfying testing of the "change theme" features. See comments in UI for more details.

- addRoomInfo

```
public void valueChanged(ListSelectionEvent e) {  
    int index = model.getMinSelectionIndex();  
    if (index != -1) {  
        Room room = (Room) availableRooms[index];  
        if (room != null) {  
            command.setSelectedRoom(room);  
            updateDataServiceRequired();  
        }  
    }  
}
```

Pic: 22 -Infeasible branching "index" in CheckInPanel

When the method *valueChanged* is called in the *ListSelectionListener* of the selection model of available rooms. It means that when the method is called, the index has a value given from the model which corresponds to one of the available rooms. Thus, the room cannot be null. This is an **infeasible statement** from the graphical interface point of view. This could have been covered by creating a test case manipulating directly the command or the panel attributes. However, this is not really needed because this branching cannot be reached by the end-user and this condition could have then been deleted.

CheckOutPanel:

In *CheckOutPanel*, 31 instructions and 4 branches have not been covered. These ones are all similar to the ones described in *CheckInPanel*.

- addOccupantInfo

```
String date = checkOutDateField.getText();
try {
    command.setCheckOutDate(new SimpleDateFormat("dd-MM-yyyy").parse(date));
}
catch (Exception ex) {
    checkOutDateField.setText(new SimpleDateFormat("dd-MM-yyyy").format(command.getCheckInDate()));
    JOptionPane.showMessageDialog(container, "The format of the inputted check-in date is invalid",
        "Input Error", JOptionPane.INFORMATION_MESSAGE);
    System.err.print("Input Error: The format of the inputted check-in date is invalid");
    return;
}
```

Pic: 23 - Uncovered branching of date error in *CheckOutPanel*

See comments in *CheckInPanel* for “Pic: 17 - Uncovered branching of date error in *CheckInPanel*.”

- getCommandValues

```
public Object getValueAt(int r, int c) {
    Room room = (Room) occupiedRooms[r];
    Occupation occupation = room.getOccupation();
    Occupant occupant = occupation.getOccupant();
    if (room == null) { return null; }
    switch(c) {
        case 0: return room.getFloorNo() + "-" + room.getRoomNo();
        case 1: return room.getTypeString();
        case 2: return new Integer(room.getCapacity());
        case 3: return new Double(room.getRate());
        case 4: if (room.getType() == 1) { return "N/A"; }
                else if (occupation.isDataServiceRequired()) { return "In used"; }
                else { return "Not in used"; }
        case 5: return occupation.getEthernetAddress();
        case 6: return occupant.getName();
        case 7: return occupant.getType();
        case 8: return occupant.getID();
        case 9: return occupant.getCompany();
        case 10: return new SimpleDateFormat("dd-MM-yyyy").format(occupation.getCheckInDate());
    }
    return "";
}
```

Pic: 24 - Infeasible branching “null room” in *CheckOutPanel*

These **infeasible branching and statement** show a possible **error**.

Some precautions are taken in order to deal with a null room by returning a null result. However, it should be done before trying to get the occupation. In case of a null room, the second line that is trying to access the *Occupation* attribute of the room, will then raise a *NullPointerException* before even reaching the if condition. The if condition is then unreachable and it shows an implementation mistake. Note, that this has been achieved by

changing the value of the selected room to null directly in the command it shouldn't be reachable from the user interface.

```
Room selectedRoom = command.getSelectedRoom();
if (selectedRoom != null) {
    int index = list.indexOf(selectedRoom);
    occupiedRoomTable.setRowSelectionInterval(index, index);
}
checkOutDateField.setText(new SimpleDateFormat("dd-MM-yyyy").format(command.getCheckOutDate()));
```

Pic: 25 - Infeasible branching in CheckOutPanel

These statements are **infeasible**. For more details see comments in *CheckInPanel* for “Pic: 18 - Infeasible branching in CheckInPanel”.

- addRoomInfo

```
public void valueChanged(ListSelectionEvent e) {
    int index = model.getMinSelectionIndex();
    if (index != -1) {
        Room room = (Room) occupiedRooms[index];
        if (room != null) {
            command.setSelectedRoom(room);
            Occupation occupation = room.getOccupation();
            Occupant occupant = occupation.getOccupant();
            command.setID(occupant.getID());
            command.setName(occupant.getName());
            command.setType(occupant.getType());
            command.setCompany(occupant.getCompany());
            command.setCheckInDate(occupation.getCheckInDate());
            command.setDataServiceRequired(occupation.isDataServiceRequired());
            command.setEthernetAddress(occupation.getEthernetAddress());
        }
    }
}
```

Pic: 26 - Infeasible branching “index” in CheckInPanel

These statements are **infeasible**. For more details see comments in *CheckInPanel* for “Pic: 22 - Infeasible branching “index” in CheckInPanel”.

SearchPanel:

In *SearchPanel*, 16 instructions and 4 branching are not covered.

- addSearchInfo

```
typeField.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        if (UIManager.getLookAndFeel().getName().equals("Windows")) {  
            JFrame f = (JFrame) typeField.getTopLevelAncestor();  
            if (f != null) {  
                f.setVisible(false); f.setVisible(true);  
            }  
        }  
    }  
});
```

Pic: 27 - Infeasible statements “change theme” in *SearchPanel*

These uncovered statements might be related to the unsatisfying testing of the “change theme” features. See comments in UI for more details.

- getCommandValues

```
public Object getValueAt(int r, int c) {  
    Room room = (Room) searchResults[r];  
    Occupation occupation = room.getOccupation();  
    Occupant occupant = occupation.getOccupant();  
    if (room == null) { return null; }  
    switch(c) {  
        case 0: return room.getFloorNo() + "-" + room.getRoomNo();  
        case 1: return room.getTypeString();  
        case 2: return new Integer(room.getCapacity());  
        case 3: return new Double(room.getRate());  
        case 4: if (room.getType() == 1) { return "N/A"; }  
                else if (occupation.isDataServiceRequired()) { return "In used"; }  
                else { return "Not in used"; }  
        case 5: return occupation.getEthernetAddress();  
        case 6: return occupant.getName();  
        case 7: return occupant.getType();  
        case 8: return occupant.getID();  
        case 9: return occupant.getCompany();  
        case 10: return new SimpleDateFormat("dd-MM-yyyy").format(occupation.getCheckInDate());  
    }  
    return "";  
}
```

Pic: 28 - Infeasible branching “null room” in *SearchPanel*

These **infeasible branching and statement** show a possible **error**. For more details, see comments in *CheckOutPanel* for “Pic: 24 - Infeasible branching “null room” in *CheckOutPanel*”

ManageRoomPanel:

In *ManageRoomPanel*, 4 instructions and 2 branches are not covered.

- addEditableRoom

```
public void valueChanged(ListSelectionEvent e) {  
    int index = model.getMinSelectionIndex();  
    if (index != -1) {  
        Room room = (Room) editableRooms[index];  
        if (room != null) {  
            command.setSelectedRoom(room);  
        }  
    }  
}
```

Pic: 29 - Infeasible branching "index" in ManageRoomPanel

This represents an **infeasible statement**. For more details, see comments in *CheckInPanel* for *Pic: 22 - Infeasible branching "index" in CheckInPanel*.

- addButtonAction

```
if(newRate > 0 && newDiscount >= 0 && newDiscount < 100){  
    Double finalRate = newRate * (1 - newDiscount/100);  
    boolean result = hotelManager.updateRoomRate(command.getSelectedRoom(), finalRate.intValue());  
    if(result){  
        if(roomInfoPanel != null){  
            controlPanel.remove(roomInfoPanel);  
            controlPanel.validate();  
            controlPanel.repaint();  
        }  
        getCommandValues();  
        editableRoomsTable.setEnabled(true);  
    } else{  
        JOptionPane.showMessageDialog(null, "room update failed!", "Error", JOptionPane.WARNING_MESSAGE);  
        System.err.print("Error: Room update failed!");  
    }  
} else{  
    JOptionPane.showMessageDialog(null, "invalid room rate or discount!", "Error", JOptionPane.ERROR_MESSAGE);  
    System.err.print("Error: Invalid room rate or discount!");  
}
```

Pic: 30 - Infeasible branchings in ManageRoomPanel

I have some issues explaining this coverage results. I cannot understand how the last statement "else" can remain uncovered whereas an error is printed right after. This might be an error from the coverage tool that has switched the colouring of the two "else" statements.

The other branching which is not covered corresponds also to an infeasible statements. The boolean result, getting his value from the *HotelManager* class, can only be true since *updateRoomRate* always return true.

TableView:

The class has been fully covered, no feasible statements or branches.

4- Testing results:

To sum up, 229 unit tests are performed including 4 that are failing and 4 raising errors. These tests have helped me to highlight problems in the production code.

The first and second parts discuss problems that could potentially lead to errors. The third one discusses improvement suggestions for the code consistency and readability. These problems can be added the ones related to the infeasible statements and branching mentioned previously and that are not discussed again here.

a. List of problems found with unsuccessful testing

The test results have the following results:

Runs: 229/229	Errors: 4	Failures: 4
---------------	-----------	-------------

Failures discussions:

While testing the *ManageRoomPanel*, I tested all the console output to check if they were the right ones. However, I obtained one assertion failure for error printings in the *test_ManageRoom_PreviousRoomInfoPanel* and *test_ManageRoom* test cases. Since *ManageRoomPanel* is a parameterized class with many input sets, this results in 4 test failures. The error output obtained is the following one:

```
Input: [8:0,500,Zero,true,Error: Discount not correctly set!]
Test: test_ManageRoom
→ org.junit.ComparisonFailure
Expected: <Error: Discount not correctly set!>
but was: <Error: Discount not correctly set![Error: Invalid room rate or discount!]>
```

```
Input: [8:0,500,Zero,true,Error: Discount not correctly set!]
Test: test_ManageRoom_PreviousRoomInfoPanel
→ org.junit.ComparisonFailure
Expected: <Error: Discount not correctly set!>
but was: <Error: Discount not correctly set![Error: Invalid room rate or discount!]>
```

```
Input: [10:1,500,null,true,Error: Discount not correctly set!]
Test: test_Manage_Room
→ org.junit.ComparisonFailure
Expected:<Error: Discount not correctly set![]>
But was:<Error: Discount not correctly set![Error: Invalid room rate or discount!]>
```

```
Input: [10:1,500,null,true,Error: Discount not correctly set!]
Test: test_ManageRoom_PreviousRoomInfoPanel → org.junit.ComparisonFailure
Expected:<Error: Discount not correctly set![]>
But was:<Error: Discount not correctly set![Error: Invalid room rate or discount!]>
```

These errors are all due to the following lines:

```
double newRate = -1;

try{
    newRate = Double.parseDouble(rate.getText());
} catch(Exception e){
    JOptionPane.showMessageDialog(null, "Rate not correctly set!", "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print("Error: Rate not correctly set!");
    return;
}

double newDiscount = -1;
try{
    newDiscount = Double.parseDouble(discount.getText());
} catch(Exception e){
    JOptionPane.showMessageDialog(null, "Discount not correctly set!", "Error", JOptionPane.ERROR_MESSAGE);
    System.err.print("Error: Discount not correctly set!");
}
```

In the second catch option, a statement “return” is missing. When raising an exception during the parsing of *newDiscount*, a first error is printed and the code keeps running until another error for an invalid discount is printed a few lines later. This should be corrected by adding a statement “return” like in the first catch block.

Errors discussion:

4 from one test of the parameterized class *TestPanelSearch* showing a null pointer exception (discussed at 4) that is run 4 times.

[test_searchPanel_null_room\[0:0,X12345678\] → java.lang.NullPointerException](#)

[test_searchPanel_null_room\[1:1,Gates\] → java.lang.NullPointerException](#)

[test_searchPanel_null_room\[2:2,Business\] → java.lang.NullPointerException](#)

[test_searchPanel_null_room\[3:3,Microsoft\] → java.lang.NullPointerException](#)

As described for the infeasible statement (*Pic: 29 - Infeasible branching “null room” in SearchPanel*), we have problems with the checking of nullity of the room in *getValue* of each Panel class. This isn’t supposed to be happening, but in case of null room, the function would return a null pointer exception because the condition is room different of room is checked too late.

b. Other problems found while inspecting the source code

The following problems were also found:

1. In the *HotelManager* class, all inputs are tested for the check-in except the check-in date which could then be null.
2. In *Occupant* and *Room*, none of the inputs is checked in order to allow or not nullable inputs. As a consequence, we need to be careful with the usage of the *toString* method of these classes since it can then raise exceptions for null parameters.
3. In *Occupant*, *Occupation* and *Room*, parameters are defined as public even if getters and setters are implemented. These parameters should be turned to private.

c. Improvement suggestions

Even if it doesn't lead to any problems right now, some improvement can then be done in the code and its organization. This could lead to an improved code consistency and readability. A non exhaustive list of remarks is available below.

1. The code rarely add the type of object when declaring lists. Adding the type of object would help the readability and make the types more explicit. It would also avoid having warnings from the Eclipse IDE.
2. In *HotelManager*, the naming of the function *findOccupant* is a bit confusing. Based on its name, we could expect to get as return a list of *Occupant*. However, we are getting a list of *Room*. The documentation is also not helping since the return description is "search result" which doesn't add any valuable information.
3. Some indexing in the layout of the code are not always respected. For instance in *HotelManager*, the method *check-in* should be moved to the left in order to fit with the class structure.
4. A lot of code related to errors handling is commented in the panel classes. It looks like old code related to an information panel that was used previously. This code should then be deleted since it isn't used anymore.
5. In *ManageRoomPanel* and *UI*, some TODO written as a comment without proper tag remain in the code and have not been implement. See "//add listener here or use button to retrieve the value", line 216 or line 234 in *ManageRoomPanel* for instance.
6. The boxing type of *int*, *float*, *double* and their related types *Integer*, *Float* and *Double* should be checked in order to avoid boxing warnings. For instance in *ManageRoomPanel*, line 279, a double is boxed to Double. This should be done by

using the constructor of *Double* or by using brackets to box it: (Double). This is usually not a big issue, but it makes the code cleaner.

7. In *ManageRoomCommand* no constructor is defined whereas the other command have one. This is not mandatory to give him one since the class is really simple but it would make it more consistent.
8. The handling of errors is not always consistent. Some of it is made by using exceptions, some if it is done by using console printing using `system.err`.
9. While printings in the console, it could be more readable to print on new lines and not only on the current one.

5. Remaining work:

As stated before, our project composed of 229 unit tests has been able to highlight some errors and improvement possibilities. However, some improvements could still be made on our test procedure. The following section discusses these potential improvements.

a. Unsolved problems faced during the testing of the project

While implemented the unit tests for the project, I have faced one problem that remains unsolved in the code. This problem is related to the testing of the “change theme” feature. Solving these problems would be a first step on improving the testing quality.

b. Improving the testing with other criteria

We have been focusing here on statement and branch coverage for the software. These two indicators are good in order to get easily a quantitative measurement of the code tested. However, it does not really help to get a qualitative estimation. For instance, just calling the creators and some methods of each classes without performing any tests would achieve a decent coverage without being helpful in order to detect problems. During this project, I tried to be methodical to obtain a good quantitative and qualitative testing. The methodology applied has been discussed and expressed mostly informally. I have been able to find some infeasible statements and errors. However, even if I tried to cover all possible cases, some of the uncovered statements show that it is difficult to achieve it and that it is not always possible.

A more reliable way to obtain qualitative testing, would have been to define test requirements and test criteria for, at least, some of the most important classes of the software. By processing first with a static analysis of the program, for instance by determining all definitions and usages, we can determine all the critical paths that should be tested in each class. This way we could obtain a testing that is matching our quality requirements.

Another way to check the quality of our tests would have been to use mutation testing. The principle of mutation testing is to create mutant programs by introducing small changes in the source code. The purpose is then to verify that our tests are able to detect mutant programs. If our program is able to detect mutants, it means that our test cases are precise enough. If they are not, it means that our test cases are too generic and could be qualitatively improved. In Eclipse IDE for Java, tools as Jester or Pitclips could be used. In order to avoid having huge computation costs, these tests can be performed for instance on the most important classes and their related test cases.

Conclusion:

Unit testing based on coverage is a good way to get a first approach on quantitative testing. It gives a first idea of which part of the source code have been covered or not. Getting a decent coverage is not the most challenging part. Calling high-level functions without performing any tests can achieve a pretty decent coverage. However, getting a really high coverage and especially branching coverage implies a deep understanding of the project in order to instantiate the right inputs and define the right test procedures. In this project, I started by trying to get a good understanding of the program. Then, I tested its classes by carefully selecting all possible input in order to design the most accurate tests based on the source code. I have finally been able to get high coverages percentages (i.e. 97,1% for statements and 90,5% for branchings) and to find some problems in the source code. However, coverage criteria are not always enough in order to validate the testing of a software. In the context of a real life project, it could be interesting to mix it with testing quality metrics such as prime path coverage or mutation testing for example.

I already had some experience with Java, unit testing using the framework JUnit and Eclemma as a coverage tool. However, through this project I have been able to learn more about unit test testing. Especially, I have been able to apply concepts seen during the lectures and to implement my first parameterized classes. This project also helped me to have a deeper understanding of the Swing library and on how to test it.

Appendix:

Test class headers

Class name	Class Description Header
TestHotelManager	<p>Class used in order to test most of the functions from {@link HotelManager}.</p> <p>All the cases of CheckIn and CheckOut are tested in detail in the parameterized classes {@link TestHotelManagerCheckIn} and {@link TestHotelManagerCheckOut}.</p>
TestHotelManager CheckIn	<p>Class used in order to all the cases that could happen when using the check in function from the class {@link HotelManager}.</p> <p>This class is a parameterized class performing three tests (one with a presidential room, one with a standard room and one with null input) on a set of 15 inputs.</p>
TestHotelManager - CheckOut	<p>Class used in order to test all the cases that could happen when using the check out function from the class {@link HotelManager}.</p> <p>This class is a parameterized class performing three tests (one with an occupied room, one with an empty room and one with null input) on a set of 3 inputs.</p>
TestOccupant	<p>Parameterized class testing the behavior of the class {@link Occupant} with various sets of input containing in particular some null values.</p>
TestOccupation	<p>Class testing the behavior of the class {@link Occupation}.</p>
TestPanelCheckIn	<p>Class used in order to test all the cases that could happen when performing a check in with the user interface.</p> <p>The tests performed are similar to the ones in {@link TestHotelManagerCheckIn}. This class is a parameterized class performing three tests (one with an occupied room, one with an empty room and one with null input) on a set of 15 inputs.</p> <p>Two other tests have been added to check the consistency between the room information and the table displayed on the screen and to check the initialization of all components of the Panel.</p>
TestPanel - CheckOut	<p>Class used in order to test all the cases that could happen when performing a check out with the user interface.</p> <p>The tests performed are similar to the ones in {@link TestHotelManagerCheckOut}. This class is a parameterized class performing four tests (one with an occupied room with data, one occupied without data, one with an empty room and one with null input) on a set of 3 inputs.</p> <p>Two other tests have been added to check the consistency between the room information and the table displayed on the screen and to check the initialization of all</p>

	<i>components of the Panel.</i>
TestManage - RoomPanel	<p><i>Parametrized class testing {@link ManageRoomPanel}.</i></p> <p><i>This class performs 4 tests (update of one room's rate and discount, cancel of the intended changes of one room's rate and discount, test of the case where no room as been selected, test on a specific error) on a set of 11 inputs.</i></p> <p><i>Two other tests have been added to check the consistency between the room information and the table displayed on the screen and to check the initialization of all components of the Panel.</i></p> <p><i>These tests also focuses on testing the printed outputs since the ManageRoomPanel deals with a lot of printed exceptions.</i></p> <p><i>Note that four of the tests are expected to fail here, since they highlight an implementation inconsistency.</i></p>
TestPanelSearch	<p><i>Tests of the {@link SearchPanel} class. This class is a parameterized class in order to check perform tests on different input values, here depending on the search type (by ID, name, type or company).</i></p> <p><i>The tests performed are the following ones:</i></p> <ul style="list-style-type: none"> <i>- search for a booked standard room</i> <i>- search for a booked presidential room without data</i> <i>- search for a booked presidential room with data</i> <i>- search when no room is booked</i> <i>- search when the selected room is null</i> <p><i>One other tests has been added to check the consistency between the room information and the table displayed on the screen.</i></p> <p><i>Note that this class should create 4 errors highlighting an implementation error in getValues().</i></p>
TestRoom	<i>Parameterized class testing the behavior of the class {@link Room} with different sets of input covering the standard cases, but also containing some null values.</i>
TestUI	<p><i>Tests for the {@link UI} class defining the main model of the UI.</i></p> <p><i>This class targets to test the following properties of the UI:</i></p> <ul style="list-style-type: none"> <i>- Initialization of the UI and its class members</i> <i>- Testing that the default command is CheckIn</i> <i>- Changing the current command from UI</i> <i>- FIXME Changing the design of the UI (raising an error)</i> <i>- FIXME Closing the window (does not work)</i>

References

1 - Documentation of the HMS project.

[https://home.cse.ust.hk/~scc/Password Only/csit5100/assign/doc.zip](https://home.cse.ust.hk/~scc/Password%20Only/csit5100/assign/doc.zip)

2 - Swing Documentation:

<https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html>

3 - Stackoverflow - JUnit Tests for GUI in Java:

<http://stackoverflow.com/questions/16411823/junit-tests-for-gui-in-java>

4 - Stackoverflow - JUnit test for System.out.println():

<http://stackoverflow.com/questions/1119385/junit-test-for-system-out-println>

5 - Stackoverflow - Swing how to close JPanel programmatically

<http://stackoverflow.com/questions/26762324/swing-how-to-close-jpanel-programmatically>

6 - Stackoverflow - How to set selected index JComboBox by value

<http://stackoverflow.com/questions/8327352/how-to-set-selected-index-jcombobox-by-value>

7 - Stackoverflow - Performing an action when an JMenuItem is clicked?

<http://stackoverflow.com/questions/1726115/performing-an-action-when-an-jmenuitem-is-clicked>

8 - Stackoverflow - Programmatically select a row in JTable

<http://stackoverflow.com/questions/8661251/programmatically-select-a-row-in-jtable>