Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 1 REPORT

AHMET YASIR NACAK
161044042

Course Assistant: Fatma Nur Esirci

1 INTRODUCTION

1.1 Problem Definition

In this homework, the problem is to build a working hotel management system. This hotel management system has two types of users. One of them is guest type and the other is receptionist type user. Each of these user types has certain capabilities. Naturally, a hotel has rooms and these rooms need to be booked, checked in and out. Booking can be done by both guests and receptionists and the opposite of that, cancellation of a booking can also be done both types of users. After a booking operation, if a guest comes to the hotel and wants to start their vacation, they need to do a check in operation. This check in operation guarantees that the user who has booked a room in this hotel is currently staying in it. Since the rights of people currently staying in the hotel is not decided by the guests, only the receptionists can perform a check in/check out operation.

1.2 System Requirements

From the problem definition, it is obvious to see that the solution to this problem requires a hotel, the rooms of that hotel and the users of the hotel management system. All these components make a hotel management system when combined.

1.2.1 Users of the System

The two types of users, guests and receptionists, are both people that is going to log-in to the hotel management system and perform actions. Because of this log-in process, they need an identification system that can separate users from each other and does not allow other people to log-in as other users. These two requirements create a need for an e-mail and a password. Other than these two, users also need a low level and easily controllable value to do quick checks when they are performing actions. Because of this they also need an identification number, an integer starting from 0 for the first user registered to the system and can go up to the integer limit. And lastly, for the informal features of the system, the system needs the first and last names of the users. An example of an informal feature can be the welcome message when a user logs into the system. After these properties of the users, they also need operations they can perform. From the problem definition, one can see that the actions performed by the users can be booking, cancellation of a booking, checking in and checking out but also from the problem definition, one can see that only the receptionist type users can utilize checking in and checking out operations. The system may allow the guests to do a check in operation, but it needs to warn them that they are not allowed to do it.

1.2.2 Rooms of the Hotel

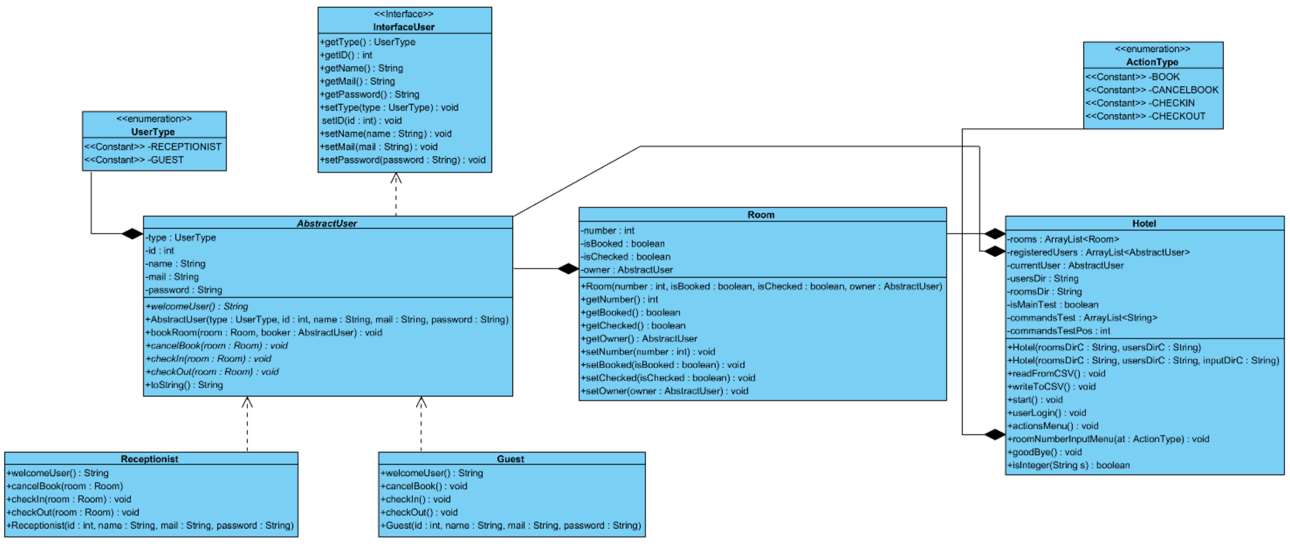
Since the whole system is based on hotels, and booking hotel rooms, the system needs a data structure that can represent a hotel room to the system in a useful form. From that, one can see that the system does not need to know the properties of the room; where its doors and windows are placed etc. The system only needs to know the room number, if the room booked or checked in and who the current owner of the room is. When these four properties are gathered together as a data type, the system can utilize this data type to keep the necessary information about these rooms.

1.2.3 The Hotel Itself

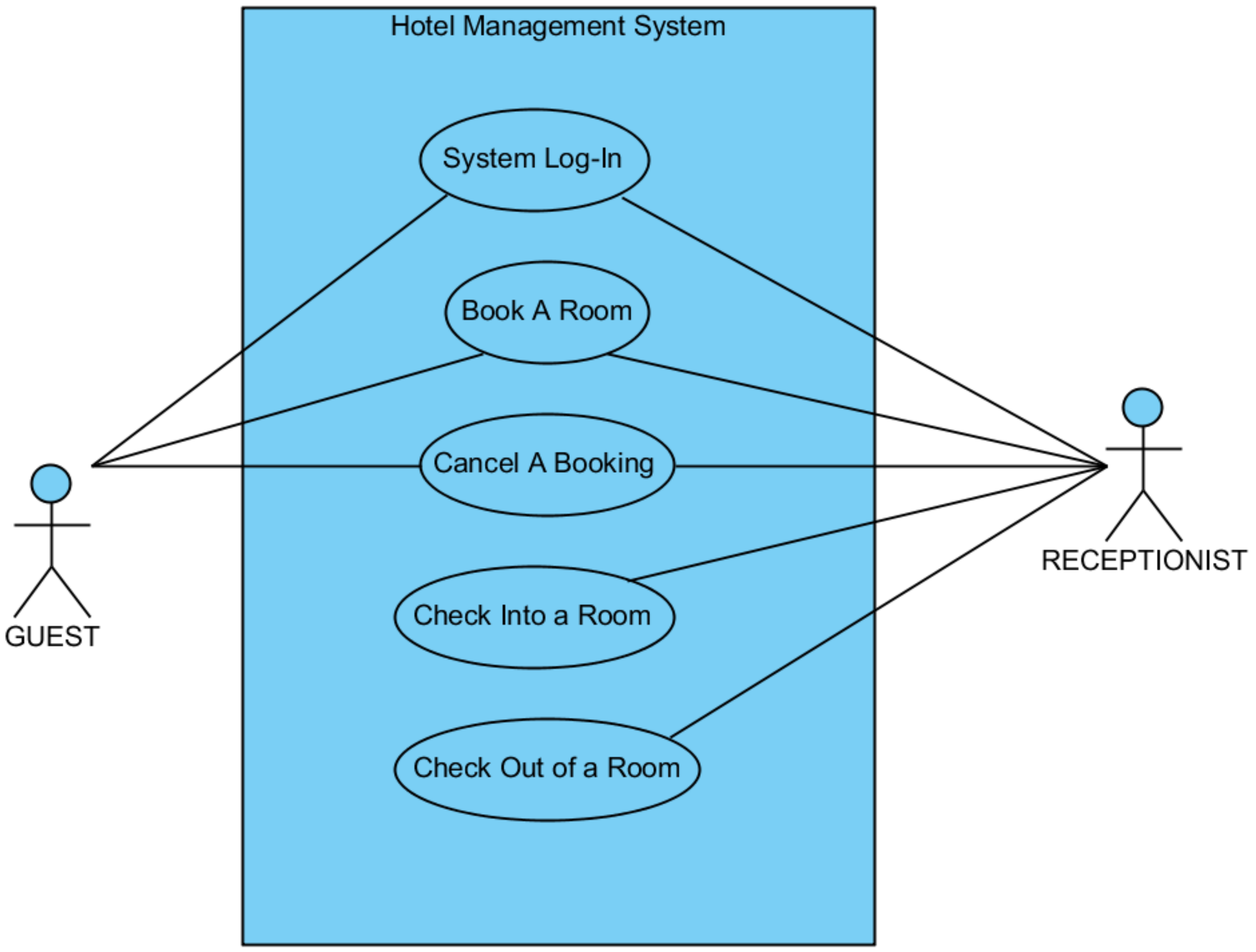
After creating all the necessary data structures that can be used for a hotel management system, there appears to be a need to create a system that can utilize these structures properly for a hotel management system. For this need, another data structure, hotel comes into the play. This data structure needs to know all its users and the rooms that they book. To provide these needs, the system needs an outside source that has the information about the guests and the receptionists, and it also needs another source that holds the information about the rooms. Most useful solution to this problem is having two separate CSV files that lists this information. After providing the system the database it needs, the system needs to have some operations that the users can utilize and bring life to our hotel management system. The two operations before the system that show themselves to the user are reader and writer operations for CSV files. These operations provide the system a secure way to keep its data. After that, the operations that can interact with the user comes in. First of these operations is the one that lets a user to log-in to the system using their e-mail address and password. This operation controls the database for given information and lets the user use the system if the given information is correct. After this confirmation, the system needs to provide the user some activities they can utilize. As mentioned in the Users of the System part, these activities are to book a room, cancel a booking, check in a room and check out a room.

2 METHOD

2.1 Class Diagram


<<Interface>>
InterfaceUser
+getType() : UserType
+getID() : int
+getName() : String
+getMail() : String
+getPassword() : String
+setType(type : UserType) : void
setID(id : int) : void
+setName(name : String) : void
+setMail(mail : String) : void
+setPassword(password : String) : void
<<enumeration>>
UserType
<<Constant>> -RECEPTIONIST
<<Constant>> -GUEST
<<enumeration>>
ActionType
<<Constant>> -BOOK
<<Constant>> -CANCELBOOK
<<Constant>> -CHECKIN
<<Constant>> -CHECKOUT
AbstractUser
-type : UserType
-id : int
-name : String
-mail : String
-password : String
+welcomeUser() : String
+AbstractUser(type : UserType, id : int, name : String, mail : String, password : String)
+bookRoom(room : Room, booker : AbstractUser) : void
+cancelBook(room : Room) : void
+checkIn(room : Room) : void
+checkOut(room : Room) : void
+toString() : String
Room
-number : int
-isBooked : boolean
-isChecked : boolean
-owner : AbstractUser
+Room(number : int, isBooked : boolean, isChecked : boolean, owner : AbstractUser)
+getNumber() : int
+getBooked() : boolean
+getChecked() : boolean
+getOwner() : AbstractUser
+setNumber(number : int) : void
+setBooked(isBooked : boolean) : void
+setChecked(isChecked : boolean) : void
+setOwner(owner : AbstractUser) : void
Hotel
-rooms : ArrayList<Room>
-registeredUsers : ArrayList<AbstractUser>
-currentUser : AbstractUser
-usersDir : String
-roomsDir : String
-isMainTest : boolean
-commandsTest : ArrayList<String>
-commandsTestPos : int
+Hotel(roomsDirC : String, usersDirC : String)
+Hotel(roomsDirC : String, usersDirC : String, inputDirC : String)
+readFromCSV() : void
+writeToCSV() : void
+start() : void
+userLogin() : void
+actionsMenu() : void
+roomNumberInputMenu(at : ActionType) : void
+goodBye() : void
+isInteger(String s) : boolean
Receptionist
+welcomeUser() : String
+cancelBook(room : Room)
+checkIn(room : Room) : void
+checkOut(room : Room) : void
+Receptionist(id : int, name : String, mail : String, password : String)
Guest
+welcomeUser() : String
+cancelBook() : void
+checkIn() : void
+checkOut() : void
+Guest(id : int, name : String, mail : String, password : String)

2.2 Use Case Diagram



2.3 Problem Solution Approach

2.3.1 The Room Class

To provide the needs of the methods that guests, or receptionists will use, we firstly need to create rooms so they can get booked or checked in and out. And it obvious that the room class is independent of the other classes, does not need to take inheritance from another class and won't have a hierarchy. The first step to create a room class is to create necessary data fields. First of these fields is the number of the room that can represented using an integer. After that, we need to add two Boolean fields that can give the information whether the room is booked or checked. And finally, we need to know who the owner of the room is currently. (it does not matter if it is only a booker or is already checked in) Because of that we can create the user class early on but with no details, and when we finally finish up the user class, we don't need to modify this class to fix any problems. The purpose of knowing the owner of the room is to do controls when the methods created by the user class hierarchy tries to modify the fields that a room holds. The room class itself will not try to modify any values of its current owner. Also, the room class will note have capabilities other than getting and setting.

2.3.2 The User Class Hierarchy

For the user class hierarchy, I firstly created an interface called InterfaceUser. This interface holds the necessary setters and getters for a user and no data fields. This is because no matter how big this system extends, every class that is a user must have all of their appropriate setters and getters. After that, I created an abstract class called AbstractUser. This abstract implements the InterfaceUser and has all the data fields necessary for the users of this system. This class also declares an abstract method called welcomeUser. This method is going to be another proof of polymorphism. And lastly, this abstract class declares 3 abstract methods and 1 concrete method. The abstract methods are the operations that allow a user to book a room, do a check in and a check out. These are abstract because they act differently for the Guest and Receptionist classes that inherit AbstractUser. The check in/check out is trying to do its job when it is being called by a receptionist but gives only an information message when it is being called by a guest. The book cancellation method is also abstract because the receptionists can cancel every booking. On the other hand, the guests are only allowed to cancel books that are made by the same guest.

2.3.3 The Hotel Class

After creating the room class and the user class hierarchy, we got all the components ready to create a more complex hotel management system that the users can utilize. The first step of creating a hotel class is to create data fields that a hotel needs. Because of that, we can add a list of users, and a list of rooms. These two will determine what rooms our hotel has and who is going to use the system. These two lists will hold the type AbstractUser and Room mentioned above.

After that, we will create another field of the class AbstractUser that will determine who is currently using the system. This field is an AbstractUser because both guests and receptionists will utilize the same system. After that, the system need to find a file where the user information and the room information being recorded. From the System Requirements part, we decided to use the CSV file format for both of this. Since we are keeping information separate from the code, we need to know where these information is being kept. Because of that, we need two strings that is telling is where to find these files. After all the data fields being defined, we need to define operations that can make the system a complete one. First two of these operations are the ones that is going to communicate with the databases that hold user and room information. The first one will be used for reading data from two CSV files that hold the user and room info and transfer that data to its respective arrays and the second operation will be used for taking the data from arrays and transfer them to CSV files. These two methods provide the system a reliable database management system. Reading data will be used when the system started running and writing data to file will run whenever an activity is being done. After the database management part of the hotel is done, we can move on to the operations that is going to interact with the user. All these operations will run depending on the input from the users of the system. The first step of using the hotel for a user is to login to the system. For this operation, the system needs to ask the user their e-mail address and password, if this information is not in the user database, the system asks the email and password again. If it is correct, the system advances the user another step where the user can choose one of the four activities: book a room, cancel a booking, do a check in and do a checkout. Since each one of these activities require a room number, the next screen asks the user to enter room number. After that input, all the inputs get combined and depending on the user type and the room number, the lists that hold the data about the hotel gets modified and gets saved to respective CSV files. This ends up one operation loop of the system and the system asks the user to go back to menu, log out of the system or exit the program altogether.

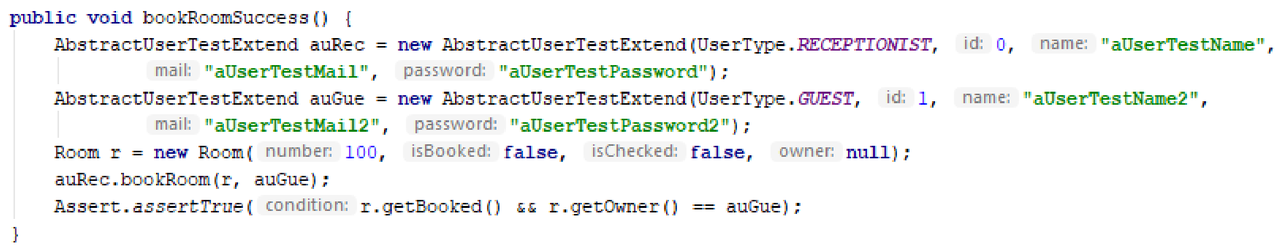
3 RESULT

3.1 Test Cases

3.1.1 Test Cases of AbstractUser

Only implemented method of the AbstractUser class is the booking method. Because of that, the unit tests will test all the possible outcomes of booking operation.

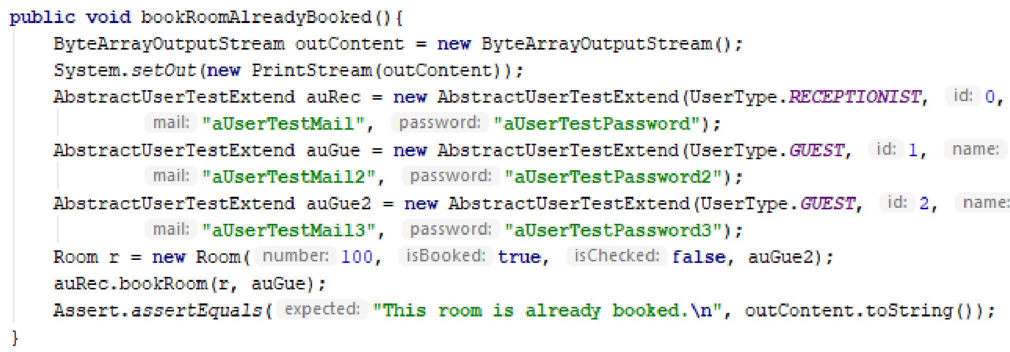
1. **A Successful Booking**



```
public void bookRoomSuccess() {
    AbstractUserTestExtend auRec = new AbstractUserTestExtend(UserType.RECEPTIONIST,  id: 0,  name: "aUserTestName",
             mail: "aUserTestMail",  password: "aUserTestPassword");
    AbstractUserTestExtend auGue = new AbstractUserTestExtend(UserType.GUEST,  id: 1,  name: "aUserTestName2",
             mail: "aUserTestMail2",  password: "aUserTestPassword2");
    Room r = new Room( number: 100,  isBooked: false,  isChecked: false,  owner: null);
    auRec.bookRoom(r, auGue);
    Assert.assertTrue( condition: r.getBooked() && r.getOwner() == auGue);
}
```

In this case, a room gets created without an owner, so it is not booked, and a receptionist books the room for a user. This test passes, so it shows the room booking for an empty room works correctly.

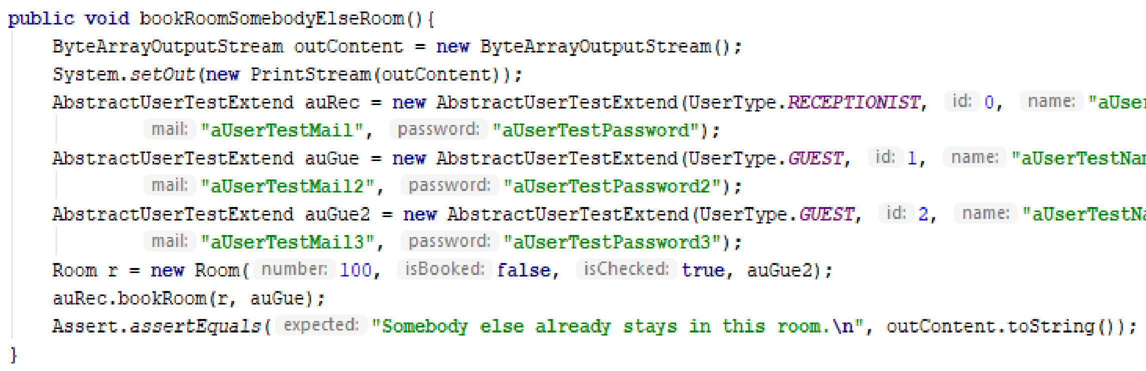
2. **Trying to Book a Room That is Already Booked**



```
public void bookRoomAlreadyBooked(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    AbstractUserTestExtend auRec = new AbstractUserTestExtend(UserType.RECEPTIONIST,  id: 0,  name: "aUserTestName",
             mail: "aUserTestMail",  password: "aUserTestPassword");
    AbstractUserTestExtend auGue = new AbstractUserTestExtend(UserType.GUEST,  id: 1,  name: "aUserTestName2",
             mail: "aUserTestMail2",  password: "aUserTestPassword2");
    AbstractUserTestExtend auGue2 = new AbstractUserTestExtend(UserType.GUEST,  id: 2,  name: "aUserTestName3",
             mail: "aUserTestMail3",  password: "aUserTestPassword3");
    Room r = new Room( number: 100,  isBooked: true,  isChecked: false, auGue2);
    auRec.bookRoom(r, auGue);
    Assert.assertEquals( expected: "This room is already booked.\n", outContent.toString());
}
```

In this case, a room gets created with an owner that is booked the room and a receptionist tries to book the room for another user. This action fails and the information message saying, "This room is already booked." Appears on the screen and the test controls for this message. The test passes.

3. **Trying to Book a Room That is Already Checked In**



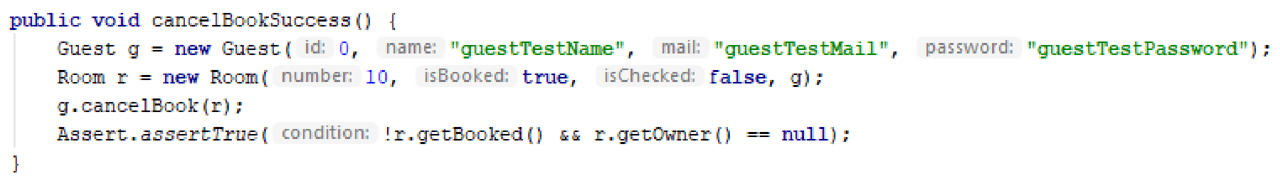
```
public void bookRoomSomebodyElseRoom(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    AbstractUserTestExtend auRec = new AbstractUserTestExtend(UserType.RECEPTIONIST,  id: 0,  name: "aUserTestName",
             mail: "aUserTestMail",  password: "aUserTestPassword");
    AbstractUserTestExtend auGue = new AbstractUserTestExtend(UserType.GUEST,  id: 1,  name: "aUserTestName2",
             mail: "aUserTestMail2",  password: "aUserTestPassword2");
    AbstractUserTestExtend auGue2 = new AbstractUserTestExtend(UserType.GUEST,  id: 2,  name: "aUserTestName3",
             mail: "aUserTestMail3",  password: "aUserTestPassword3");
    Room r = new Room( number: 100,  isBooked: false,  isChecked: true, auGue2);
    auRec.bookRoom(r, auGue);
    Assert.assertEquals( expected: "Somebody else already stays in this room.\n", outContent.toString());
}
```

In this case, a room gets created with an owner that is already staying in that room and a receptionist tries to book the room for another user. This action fails and the information message saying, “Somebody else already stays in this room.” Appears on the screen and the test controls for this message. The test passes.

3.1.2 Test Cases of Guest

Methods implemented for the guest class are cancel book, check in and check out. Even though the check in and out are implemented, they are there to show the user an information message.

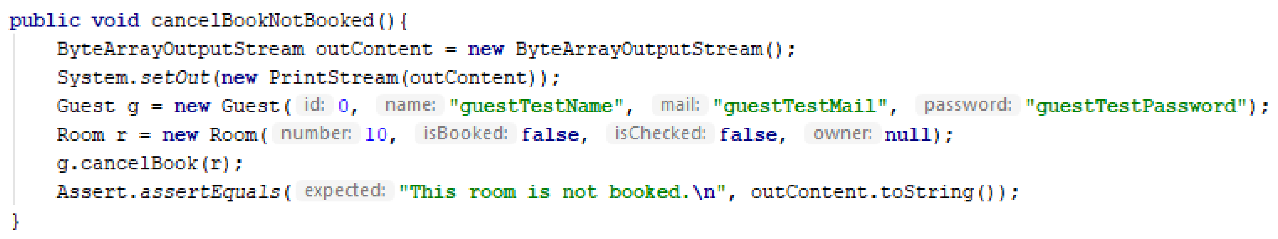
1. **Successfully Cancelling a Booking**



```java
public void cancelBookSuccess() {
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room r = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    g.cancelBook(r);
    Assert.assertTrue( condition: !r.getBooked() && r.getOwner() == null);
}
```

In this case, a room gets created with a guest booked it. When the guest who booked that room tries to cancel their booking, the operation successes. The test controls if the room hasn’t got an owner anymore and it give true.

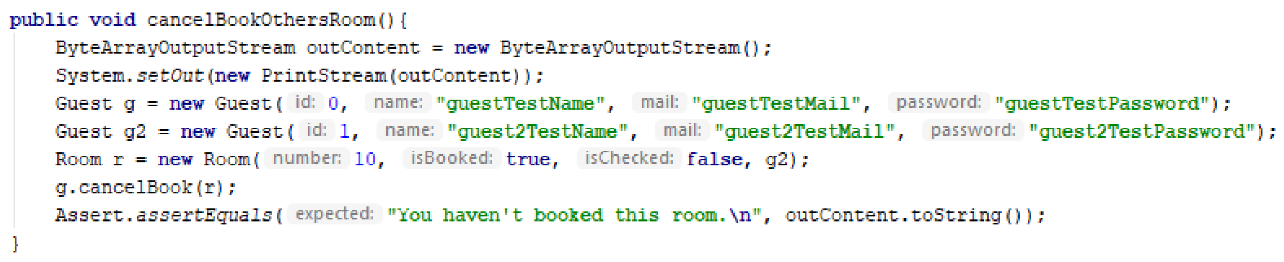
2. **Cancelling the Booking of a Room That is Not Booked**



```java
public void cancelBookNotBooked(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room r = new Room( number: 10,  isBooked: false,  isChecked: false,  owner: null);
    g.cancelBook(r);
    Assert.assertEquals( expected: "This room is not booked.\n", outContent.toString());
}
```

In this case, a room gets created with nobody booked it. When a guest tries to cancel the booking of that room, the action fails and the information message saying, “This room is not booked.” Appears on the screen and the test controls for this message.

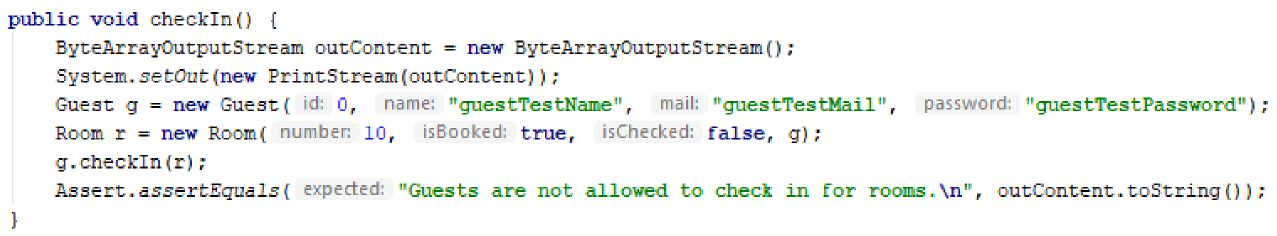
3. **Cancelling the Booking of Another Guests Room**



```java
public void cancelBookOthersRoom(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Guest g2 = new Guest( id: 1,  name: "guest2TestName",  mail: "guest2TestMail",  password: "guest2TestPassword");
    Room r = new Room( number: 10,  isBooked: true,  isChecked: false, g2);
    g.cancelBook(r);
    Assert.assertEquals( expected: "You haven't booked this room.\n", outContent.toString());
}
```

In this case, a room gets created with a user booked it. When another user tries to cancel the booking of this room, the action fails and the information saying, “You haven’t booked this room.” Appears on the screen and the test controls for this message.

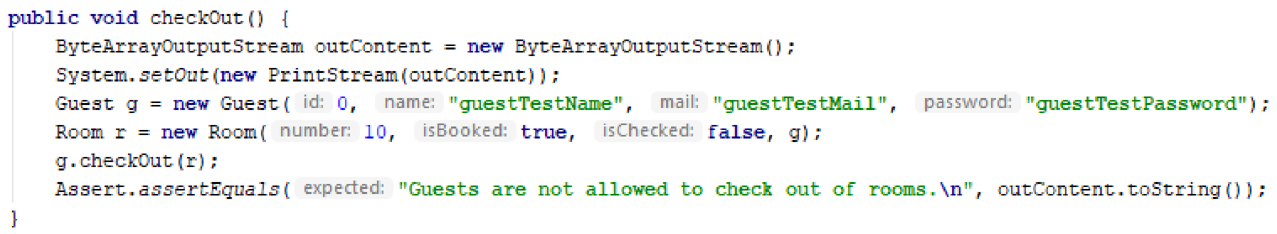
4. Checking in as a User



```
public void checkIn() {
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room r = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    g.checkIn(r);
    Assert.assertEquals( expected: "Guests are not allowed to check in for rooms.\n", outContent.toString());
}
```

In this case, a room gets created with a user booked it. When a guest tries to check in for that room, the action fails and the information message saying, “Guests are not allowed to check in for rooms.” Appears on the screen and the test controls for this message.

5. Checking out as a User

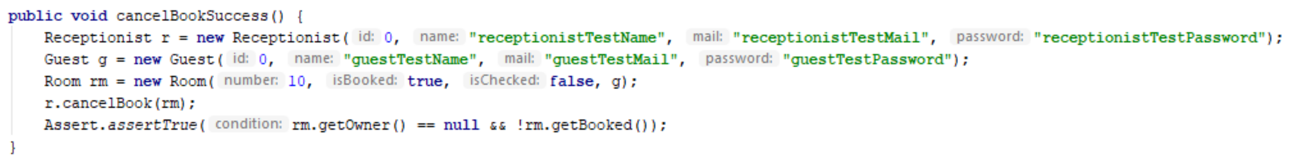


```
public void checkOut() {
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room r = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    g.checkOut(r);
    Assert.assertEquals( expected: "Guests are not allowed to check out of rooms.\n", outContent.toString());
}
```

In this case, a room gets created with a user checked in it. When a guest trues to check out for that room, the action fails and information message saying, “Guests are not allowed to check out for rooms.” Appears on the screen and the test controls for this message.

3.1.3 Test Cases of Receptionist

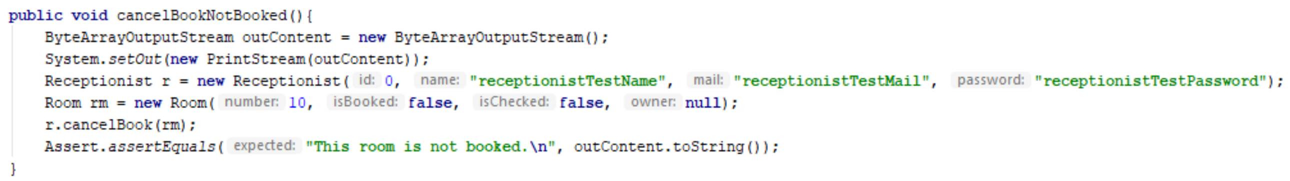
1. Successfully Cancelling a Booking



```
public void cancelBookSuccess() {
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room rm = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    r.cancelBook(rm);
    Assert.assertTrue( condition: rm.getOwner() == null && !rm.getBooked());
}
```

In this case, a room gets created with a user booked it. When a receptionist tries to cancel that rooms booking, the action succeeds and the test controls if the room hasn’t got an owner.

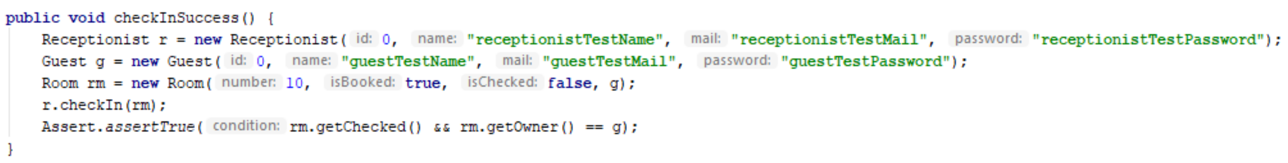
2. Cancelling the Booking of a Room That is not Booked



```
public void cancelBookNotBooked(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Room rm = new Room( number: 10,  isBooked: false,  isChecked: false,  owner: null);
    r.cancelBook(rm);
    Assert.assertEquals( expected: "This room is not booked.\n", outContent.toString());
}
```

In this case, a room gets created without any user booked it. When a receptionist tries to cancel that rooms booking, the action fails and an information message saying, “This room is not booked.” Appears on the screen and the test controls for this message.

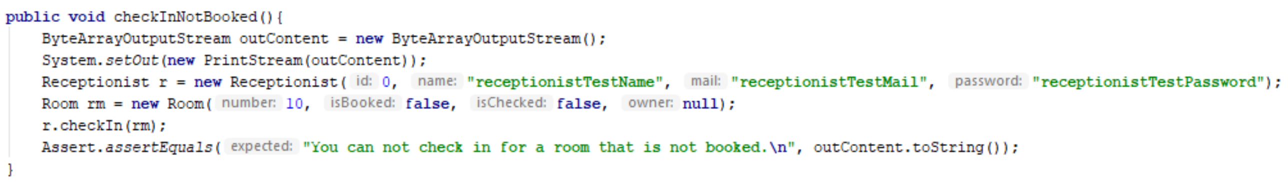
3. Successfully Checking in a Guest



```
public void checkInSuccess() {
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room rm = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    r.checkIn(rm);
    Assert.assertTrue( condition: rm.getChecked() && rm.getOwner() == g);
}
```

In this case, a room gets created with a user booking it. When a receptionist tries to check in that guest to his/her room, the action succeeds and the test controls if the room is checked and the owner is same as the previous one.

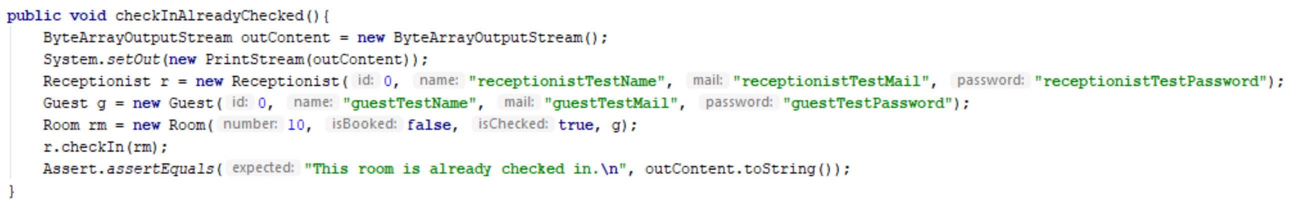
4. Checking in For a Room That is Not Booked



```
public void checkInNotBooked(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Room rm = new Room( number: 10,  isBooked: false,  isChecked: false,  owner: null);
    r.checkIn(rm);
    Assert.assertEquals( expected: "You can not check in for a room that is not booked.\n", outContent.toString());
}
```

In this case, a room gets created without any user booked it. When a receptionist tries to check in for that room, the action fails and an information message saying, “You can not check in for a room that is not booked.” appears on the screen and the test controls for this message.

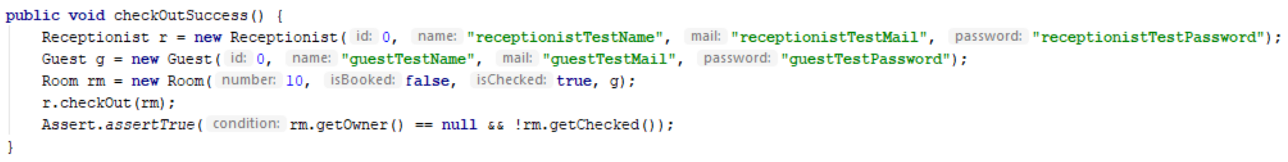
5. Checking in For a Room That is Already Checked in



```
public void checkInAlreadyChecked(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room rm = new Room( number: 10,  isBooked: false,  isChecked: true, g);
    r.checkIn(rm);
    Assert.assertEquals( expected: "This room is already checked in.\n", outContent.toString());
}
```

In this case, a room gets created with a guest already checked in for that room. When a receptionist tries to check in for that room, the action fails and an information message saying, “This room is already checked in.” appears on the screen and the test controls for this message.

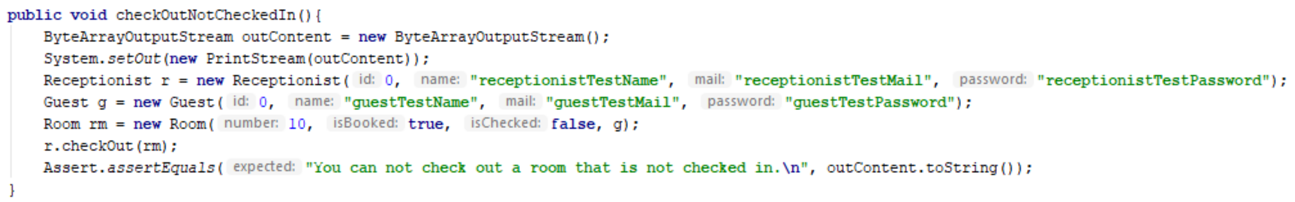
6. Checking out for a Room Successfully



```
public void checkOutSuccess() {
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room rm = new Room( number: 10,  isBooked: false,  isChecked: true, g);
    r.checkOut(rm);
    Assert.assertTrue( condition: rm.getOwner() == null && !rm.getChecked());
}
```

In this case, a room gets created with a guest already checked in for that room. When a receptionist tries to check out for that room, the action succeeds and the test controls if the room hasn’t got an owner anymore and it is available for bookings again.

7. **Checking out for a Room That is not Checked in**

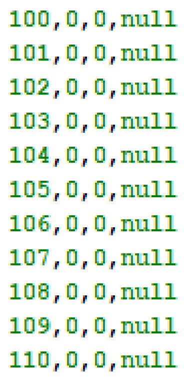


```java
public void checkOutNotCheckedIn(){
    ByteArrayOutputStream outContent = new ByteArrayOutputStream();
    System.setOut(new PrintStream(outContent));
    Receptionist r = new Receptionist( id: 0,  name: "receptionistTestName",  mail: "receptionistTestMail",  password: "receptionistTestPassword");
    Guest g = new Guest( id: 0,  name: "guestTestName",  mail: "guestTestMail",  password: "guestTestPassword");
    Room rm = new Room( number: 10,  isBooked: true,  isChecked: false, g);
    r.checkOut(rm);
    Assert.assertEquals( expected: "You can not check out a room that is not checked in.\n", outContent.toString());
}
```

In this case, a room gets created with a guest booked it. When a receptionist tries to check out for that room, the action fails and an information message saying, “You can not check out a room that is not checked in.” appears on the screen and the test controls for this message.

3.2 Running Results

List of the rooms before running the program:



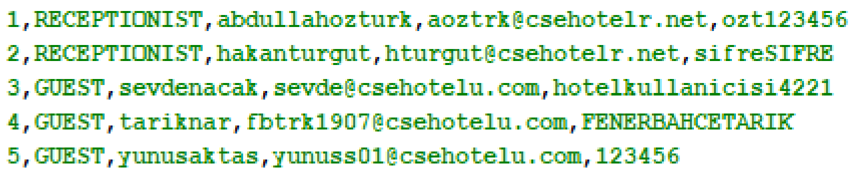
```
100,0,0,null
101,0,0,null
102,0,0,null
103,0,0,null
104,0,0,null
105,0,0,null
106,0,0,null
107,0,0,null
108,0,0,null
109,0,0,null
110,0,0,null
```

The numbers in this list means the following:

1. Number of the room
2. Is the room booked (0 for not booked, 1 for booked)
3. Is the room checked in (0 for not checked in, 1 for checked in)
4. Who is the owner of this room (null if nobody, a username if somebody else)

From that information, we can see that in the initial state, the hotel is empty and has no guests booked a room or staying in it.

List of the users:



```
1,RECEPTIONIST,abdullahozturk,aoztrk@csehotelr.net,ozt123456
2,RECEPTIONIST,hakanturgut,hturgut@csehotelr.net,sifreSIFRE
3,GUEST,sevdenacak,sevde@csehotelu.com,hotelkullanicisi4221
4,GUEST,tariknar,fbtrk1907@csehotelu.com,FENERBAHCETARIK
5,GUEST,yunusaktas,yunuss01@csehotelu.com,123456
```

The information in this list means the following:

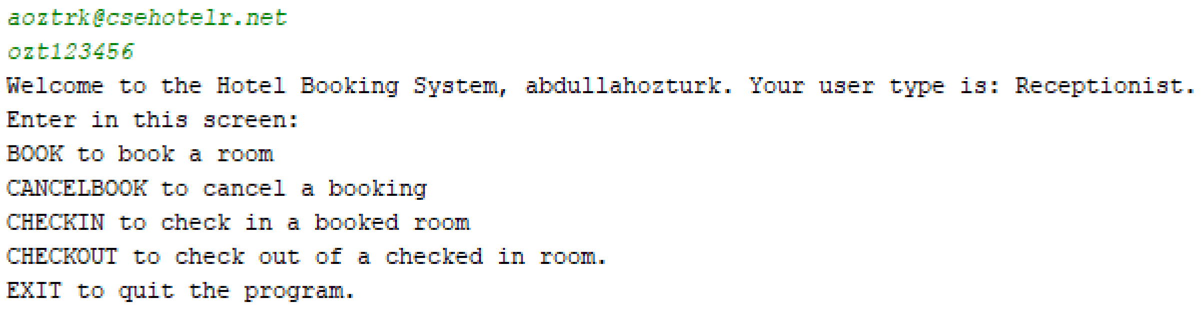
1. Identification number of the user
2. Type of the user
3. Username of the user
4. E-Mail address of the user
5. Password of the user

After running the system, when we give the following inputs:

aoztrk@csehotelr.net

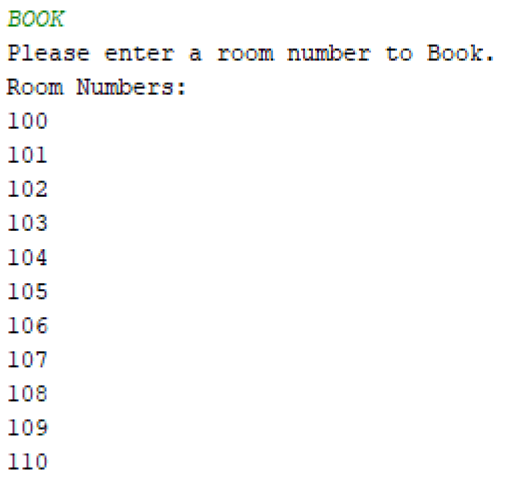
ozt123456

We get:



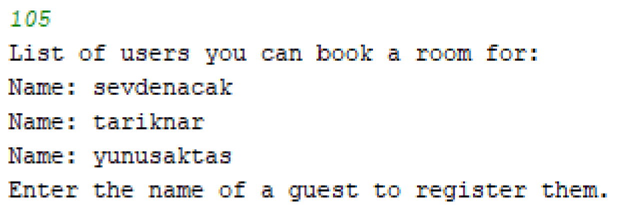
```
aoztrk@csehotelr.net
ozt123456
Welcome to the Hotel Booking System, abdullahozturk. Your user type is: Receptionist.
Enter in this screen:
BOOK to book a room
CANCELBOOK to cancel a booking
CHECKIN to check in a booked room
CHECKOUT to check out of a checked in room.
EXIT to quit the program.
```

This screen welcomes the user and asks for an action to be performed. When we choose BOOK, we get:



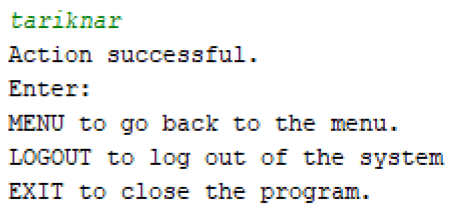
```
BOOK
Please enter a room number to Book.
Room Numbers:
100
101
102
103
104
105
106
107
108
109
110
```

This screen lets the user select a room number to book. When we choose a room number, we get:



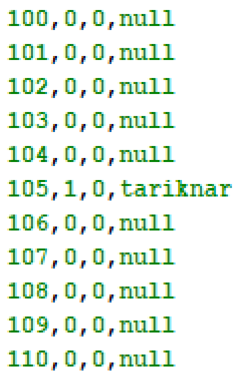
```
105
List of users you can book a room for:
Name: sevdenacak
Name: tariknar
Name: yunusaktas
Enter the name of a guest to register them.
```

This appears because current user of the system is a receptionist. If the current user was a guest, after choosing a room number, the system would response as if the guest wanted to book a room for themselves. If we enter tariknar in this screen, we get:



```
tariknar
Action successful.
Enter:
MENU to go back to the menu.
LOGOUT to log out of the system
EXIT to close the program.
```

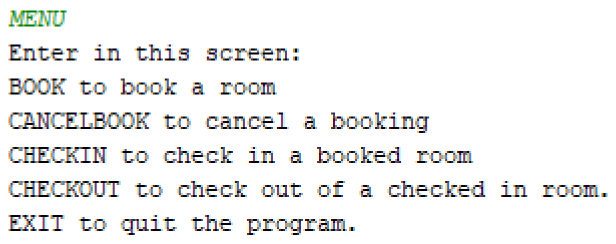
This screen informs the receptionist that the booking is successful. At this point, the user can choose to back to the menu to perform more actions, log out of the system and let other users login or exit the program.
After these actions, list of rooms become:



```
100,0,0,null
101,0,0,null
102,0,0,null
103,0,0,null
104,0,0,null
105,1,0,tariknar
106,0,0,null
107,0,0,null
108,0,0,null
109,0,0,null
110,0,0,null
```

From this list, it is easy to see that the room booking system is working since the room 105 is booked and its owner is tariknar.

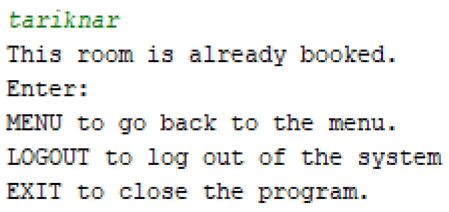
After that, if we choose to go back to the menu and perform more actions, we type in MENU and we get:



```
MENU
Enter in this screen:
BOOK to book a room
CANCELBOOK to cancel a booking
CHECKIN to check in a booked room
CHECKOUT to check out of a checked in room.
EXIT to quit the program.
```

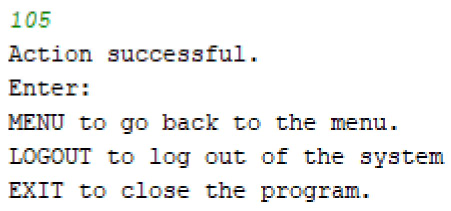
After this point we are going to skip some screenshots because they will be the same as previous ones. Only the situations that are not showed above will be show.

From this menu, if we choose to book the room 105 for another guest, let’s say yunusaktas, we get:



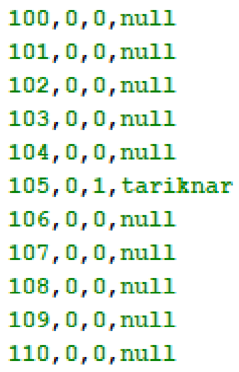
```
tariknar
This room is already booked.
Enter:
MENU to go back to the menu.
LOGOUT to log out of the system
EXIT to close the program.
```

This gives us the error message that this room is already booked so nothing happens to that room. After trying that, if we choose to check in for the room 105, we get:



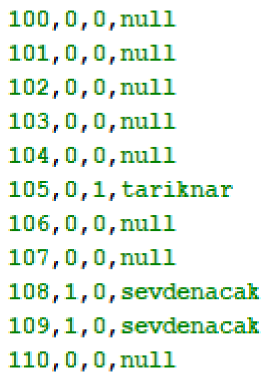
```
105
Action successful.
Enter:
MENU to go back to the menu.
LOGOUT to log out of the system
EXIT to close the program.
```

This means that tariknar is currently staying in this room and the list of rooms become:



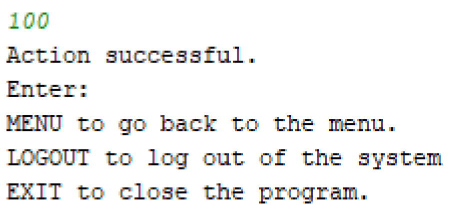
```
100,0,0,null
101,0,0,null
102,0,0,null
103,0,0,null
104,0,0,null
105,0,1,tariknar
106,0,0,null
107,0,0,null
108,0,0,null
109,0,0,null
110,0,0,null
```

Notice that the place of 1 has changed because the room is not booked anymore but it is checked in. After this, if we choose to go back to the menu and book the room 108 and room 109 for another user, let’s say sevdenacak, the list of rooms become:



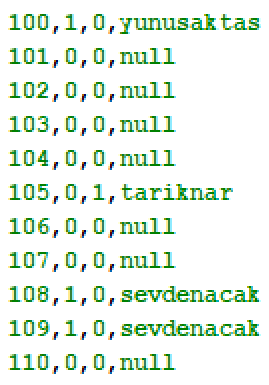
```
100,0,0,null
101,0,0,null
102,0,0,null
103,0,0,null
104,0,0,null
105,0,1,tariknar
106,0,0,null
107,0,0,null
108,1,0,sevdenacak
109,1,0,sevdenacak
110,0,0,null
```

After doing these operations, when we log out of the system and log in as a user, let’s say yunusaktas and book a room for ourself, we get:



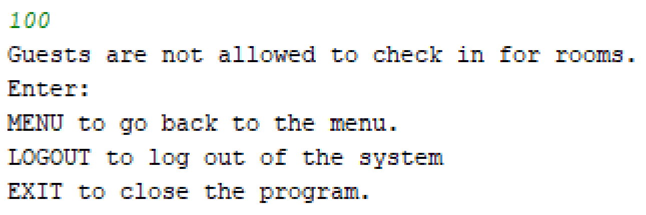
```
100
Action successful.
Enter:
MENU to go back to the menu.
LOGOUT to log out of the system
EXIT to close the program.
```

This time the system does not ask for a username to be entered because the current user is a guest and guests can only book rooms for themselves and the list of rooms become:



```
100,1,0,yunusaktas
101,0,0,null
102,0,0,null
103,0,0,null
104,0,0,null
105,0,1,tariknar
106,0,0,null
107,0,0,null
108,1,0,sevdenacak
109,1,0,sevdenacak
110,0,0,null
```

If we try to do our check in as a user, we get:



```
100
Guests are not allowed to check in for rooms.
Enter:
MENU to go back to the menu.
LOGOUT to log out of the system
EXIT to close the program.
```

And the system continues to run like that until in the menu or after doing an operation, the user types EXIT. The file provided with the project called testInputStream.txt runs commands like that and at the fills the hotel with guests, does their check ins and outs, and eventually clears the hotel like it is same as the beginning. If a user tries to type in the content of testInputStream.txt to the executable file line by line, they can track the changes of roominfo.csv.