Issue Refactoring - Code Smells

Hello,

We are students of the software design course, because of the content of our subject we have had to identify possible "bad smells" in repository codes and some "refactoring" that would be a solution to these bad smells that are fundamental basis for building a scalable, maintainable and readable code.

This is all for a practice and learning motive.

Thank you very much.

Code Smells

Hotel Managment

Class: DoubleRoom and SingleRoom

1. **Object-Orientation Abusers -> Alternative Classes with Different Interfaces**

```
public class SingleRoom implements Serializable {
        String name;
        String contact;
        String gender;
        ArrayList<Food> food = new ArrayList<>();

        SingleRoom() {
                this.name = "";
        }

        SingleRoom(String name, String contact, String gender) {
                this.name = name;
                this.contact = contact;
                this.gender = gender;
        }
}
```

```
public class DoubleRoom extends SingleRoom implements Serializable {
        String name2;
        String contact2;
        String gender2;

        DoubleRoom() {
                this.name = "";
                this.name2 = "";
        }

        DoubleRoom(String name, String contact, String gender, String name2, String contact2, String gender2) {
                this.name = name;
                this.contact = contact;
                this.gender = gender;
                this.name2 = name2;
                this.contact2 = contact2;
                this.gender2 = gender2;
        }
}
```

Between these two classes we can notice the smell "Alternative Classes with Different Interfaces" because if we look, the two classes are basically the same, thus having the same behavior, the only slight change we have is that the "DoubleRoom" class has one more attribute which is a person (attribute that already exists in the "SingleRoom" class). To avoid this, we can apply the refactoring method "Extract Class" to convert these two classes into one, but they can modify their behavior internally to be a "Single" or "Double" room.

Class: Holder

2. **Dispensable -> Data Class**

In the Holder class we can find that the only thing it is being used for is to define the number of rooms available in the hotel.

```
public class Holder implements Serializable {
        DoubleRoom arr1[] = new DoubleRoom[10]; // Luxury
        DoubleRoom arr2[] = new DoubleRoom[20]; // Deluxe
        SingleRoom arr3[] = new SingleRoom[10]; // Luxury
        SingleRoom arr4[] = new SingleRoom[20]; // Deluxe
}
```

The bad smell that can be noticed is that it only has its attributes that are used by other classes, being the Holder class as a container of this data, besides it does not have any additional functionality, nor can it operate independently.

Another way to notice a bad smell is that the amount of SingleRoom and DoubleRoom available are values that are burned, that we will not be able to add more rooms in case of expansion of the hotel or that the system works with another client, as a result we have a too static program.

Class: Main

3. **Dispensable -> Duplicated code**

Due to the similarities of lines and actions along the switch and the different conditionals present, there is the smell "Duplicate Code", this makes evident the use of copy-paste and it would be better if it is grouped in some way or try to present better those conditionals.

```
System.out.println(
                "\nEnter Your Choice :\n1.Show Room Details\n2.Show Room Availability \n3.Book Room\n4.Order Food\n5.Checkout Now\n6.(
choice1 = sc.nextInt();
switch (choice1) {
case 1:
        System.out.println(
                        "\nChoose Room Type :\n1.Luxury Double Bedroom \n2.Deluxe Double Bedroom \n3.Luxury Single Bedroom \n4.Deluxe
        choice2 = sc.nextInt();
        Hotel.features(choice2);
        break;
case 2:
        System.out.println(
                        "\nChoose Room Type :\n1.Luxury Double Bedroom\n2.Deluxe Double Bedroom\n3.Luxury Single Bedroom\n4.Deluxe Sir
        choice2 = sc.nextInt();
        Hotel.availability(choice2);
        break;
case 3:
        System.out.println(
                        "\nChoose Room Type :\n1.Luxury Double Bedroom\n2.Deluxe Double Bedroom\n3.Luxury Single Bedroom\n4.Deluxe Sir
        choice2 = sc.nextInt();
        Hotel.bookroom(choice2);
        break;
```

Class: NotAvailable

4. **Dispensable -> Lazy Class**

Because this class has as only objective to have a toString method (although it extends exception), it is considered a "Lazy Class" because it does basically nothing, the responsibility that this one has can be replaced easily with a String variable that can be printed if the room that you want to reserve is not available (with the help of a conditional).

```
package main;

public class NotAvailable extends Exception {
        @Override
        public String toString() {
                return "Not Available !";
        }
}
```

Refactoring

5. Extract Superclass

This refactoring is simple, we can make a single class "Room" which can have a list with objects of a class of type "Person" which will have the attributes of name, gender and others required (This helps to group them better and efficiently in the array), it will also have an attribute isDouble, to know if it is a double room. Thus, it will not matter to have a different class to know if it is double, it can be done with a simple validation according to the number of people in the array:

```java
public class Persona{

    private String nombre;
    private String genero;
    private String contacto;


    public Persona(){
      this.nombre = "";
    }

    public Persona(String nombre, String
    genero,String contacto){

      this.nombre = nombre;
      this.genero = genero;
      this.contacto = contacto;


    }

    //getters and setters


}
```

```java
public class Room implements Serializable{

    private List<Persona> lista ;

    private boolean isDouble;


    public void agregarPersona(Persona p){

        lista.add(p);
    }

    public Persona quitarPersona(Persona p){
        int indice = lista.indexOf(p);
        Persona persona = lista.remove(indice);

        return persona;
    }

    public void isDouble(){

        int numeroP = lista.size();
        if(numeroP<2){
          this.isDouble = false;
        }

        this.isDouble = true;

    }
```

We will not worry about the Person class having "Data Class" as more functionalities could be added to this class later.

6. Consolidate Duplicate Conditional Fragments
7. Consolidate Conditional Expression

In the "Main" class there are code fragments that belong to the conditionals which are repeated unnecessarily causing "Duplicate Code" this can be fixed with "Consolidate Duplicate Conditional Fragments" removing those actions that are repeated in several conditionals (like prints); and with Consolidate Conditional Expression, grouping certain actions in a method since several conditionals return or make the same call to a method.

```
System.out.println(
                "\nEnter Your Choice :\n1.Show Room Details\n2.Show Room Availability \n3.Book Room\n4.Order Food\n5.Checkout Now\n6.Quit\n");
choice1 = sc.nextInt();
switch (choice1) {
case 1:
        System.out.println(
                        "\nChoose Room Type :\n1.Luxury Double Bedroom \n2.Deluxe Double Bedroom \n3.Luxury Single Bedroom \n4.Deluxe Single Bedroom \n");
        choice2 = sc.nextInt();
        Hotel.features(choice2);
        break;
case 2:
        System.out.println(
                        "\nChoose Room Type :\n1.Luxury Double Bedroom\n2.Deluxe Double Bedroom\n3.Luxury Single Bedroom\n4.Deluxe Single Bedroom\n");
        choice2 = sc.nextInt();
        Hotel.availability(choice2);
        break;
```

In the following image two things were done: first, a general method was made to analyze the variable "choice1" to be able to group the choice between parts: if it is 6, the quit() method is called to exit; if it is between 1 and 3, another method is called to try to perform an action regarding the rooms (the print mentions them); and if it is none of these, it calls another method (anotherAction()), to analyze and decide which action to perform.

Another thing that was done is to remove the actions that were repeated in several fragments of the switch, for example, the print that showed the different types of rooms was placed in only one part and where it is assured that the option that was chosen has to do with the rooms, also the fact that there is no longer the repetition of call to methods of choice1 (for example choice1.example1, choice1.example2, ...) and in this way a cleaner code is noticed.

```
System.out.println(
                "\nEnter Your Choice :\n1.Show Room Details\n2.Show Room Availability \n3.Book Room\n4.Order Food\n5.Checkout
                Now\n6.Quit\n");
        choice1 = sc.nextInt();



        analizarEleccion(choice1);



public void analizarEleccion(choice1){

    if(choice1 == 6){
        quit();
    }



    else if(0<choice1<4){

        System.out.println(
                "\nChoose Room Type :\n1.Luxury Double Bedroom \n2.Deluxe Double Bedroom \n3.Luxury Single Bedroom \n4.Deluxe
                Single Bedroom \n");

        accionHabitacion(choice1);
    }

    else {

        otraAccion(choice1);
    }
}
```