



UNIVERSITÀ DEGLI STUDI DI FIRENZE

Facoltà di Ingegneria

Corso di Laurea in Ingegneria Informatica

Elaborato Ingegneria del Software

Alessio Corsi

# **Applicativo Java che simula la gestione di un sistema per le registrazioni in un Hotel**

A.A. 2019-2020

# Indice:

INTRODUZIONE:	PAG.4
Intenti del Progetto	Pag. 4
Realizzazione	Pag. 5
1.PROGETTAZIONE	PAG.6
1.1 Casi d'Uso	Pag.6
1.2 Diagramma UML	Pag.7
1.3 Mockups	Pag.8
2.REALIZZAZIONE	PAG.12
2.1 Classi e interfacce	Pag.13
2.1.1 HotelManagement	Pag.13
2.1.2 Employee	Pag.14
2.1.3 Guest	Pag.15
2.1.4 GuestRelatives	Pag.16
2.1.5 StayAccount	Pag.17
2.1.6 Room	Pag.18
2.1.7 SingleRoom	Pag.18
2.1.8 DoubleRoom	Pag.19
2.1.9 SuiteRoom	Pag.19
2.1.10 DrinkEFood	Pag.20
2.1.10 Extra	Pag.21
2.2 Design Pattern	Pag.22
2.2.1 Singleton	Pag.22

**2.2.2** Strategy Pattern Pag.23

**2.2.3** Builder Pag.25

**2.2.3** Observer Pag.26

**2.2.5** MVC Pag.27

### **3.TESTING** PAG.28

**3.1** HotelManagement Test Pag.29

**3.2** Guest Test Pag.30

**3.3** DrinkEFood Test Pag.31

**3.4** Extra Test Pag.31

**3.5** RestaurantEBar Test Pag.32

**3.6** RoomService Test Pag.32

**3.7** StayAccount Test Pag.33

### **4.SEQUENCE DIAGRAM** PAG.34

**4.1** Sequence Diagram Pag.34

# Introduzione:

## INTENTO DEL PROGETTO:

L'elaborato simula in modo semplificato il funzionamento di un sistema per la gestione di un piccolo hotel. Attraverso la pagina di Login l'applicazione registrerà i dati di accesso del dipendente dell'hotel. Una volta effettuato l'accesso sarà possibile per il gestore inserire i dati di ogni utente e di eventuali accompagnatori che sono con lui, inoltre sarà possibile assegnargli una camera per un periodo di permanenza, naturalmente fino a che ci saranno camere disponibili. Ad ogni paziente sarà quindi associata una scheda con le informazioni personali, più quelle di eventuali accompagnatori, la camera a lui assegnata e la permanenza in hotel. Sarà inoltre possibile per ogni utente inserire e tenere traccia di tutti gli eventuali acquisti fatti al bar o al ristorante dell'hotel, oppure delle richieste fatte al servizio in camera. E' presente quindi una pagina che permetterà al dipendente di selezionare gli articoli dal menù e di inserirli nel conto del cliente. Si deve precisare che non tutti gli articoli presenti sul menù potranno essere consegnati in camera e che in caso di consegna in camera di un articolo questo avrà un sovrapprezzo una volta che il sistema calcolerà il conto finale. Ci sarà quindi una pagina che permetterà di calcolare il conto tenendo presente la tipologia di camera assegnata al cliente, il periodo di permanenza all'interno della struttura e gli extra consumati durante il soggiorno.

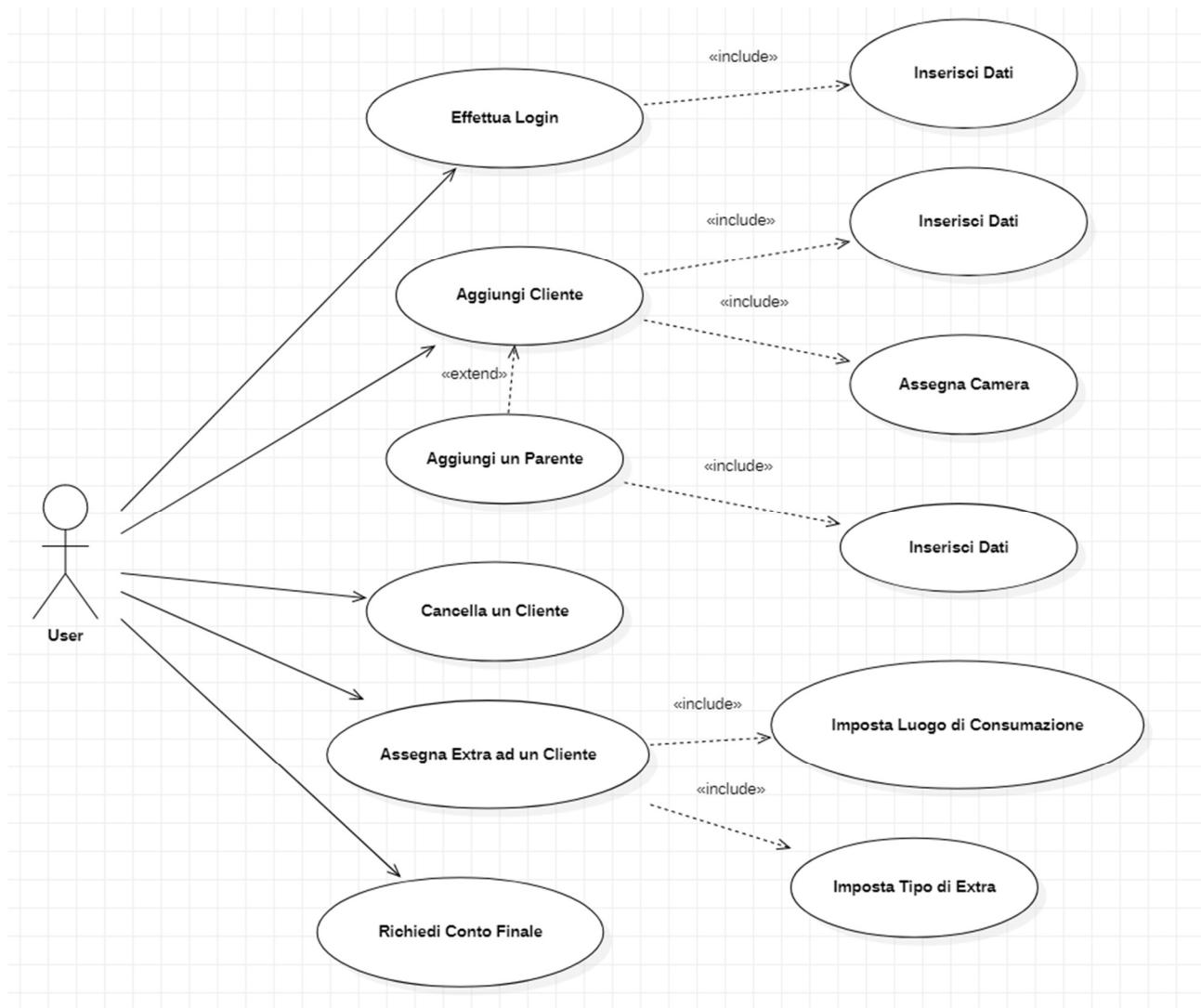
## REALIZZAZIONE:

Per la realizzazione di questa applicazione ho usato il linguaggio Java e l'IDE Eclipse. Durante la progettazione sono stati pensati e schematizzati grazie all'ausilio di un diagramma UML i modelli che vanno a comporre la struttura dell'applicazione. Inoltre è stato pensato e realizzato uno Use Case Diagram per la visualizzazione dei possibili casi d'uso del sistema. Nel realizzare la struttura dell'applicazione sono stati implementati alcuni pattern all'interno del package src/model. Abbiamo alcuni pattern comportamentali come lo Strategy, utilizzato nel calcolo del conto degli extra a seconda che essi siano stati consumati in camera o meno, oppure come l'Observer utilizzato per notificare alla classe hotel che una camera non è più fra quelle libere nel caso in cui venga assegnata ad un cliente (quindi implementato tra la classe madre Room che rappresenta le camere e la classe Hotel). Inoltre sono stati sviluppati anche due pattern creazionali. Il primo è un Builder, utilizzato per la creazione della classe Guest relativa al Cliente, mentre il secondo è un Singleton, utilizzato nella costruzione della classe che rappresenta l'hotel. Per quanto riguarda la gestione grafica dell'applicazione è stato utilizzato il framework Java.swing integrato con l'utilizzo del pattern MVC. Sono inoltre stati realizzati dei test, nello specifico test di Unit Testing, utilizzando il framework JUnit 5. L'intero progetto è inoltre disponibile su GitHub al link: <https://github.com/alessio199825/Applicativo-Hotel>

# Capitolo 1: Progettazione

## 1.1 CASI D'USO:

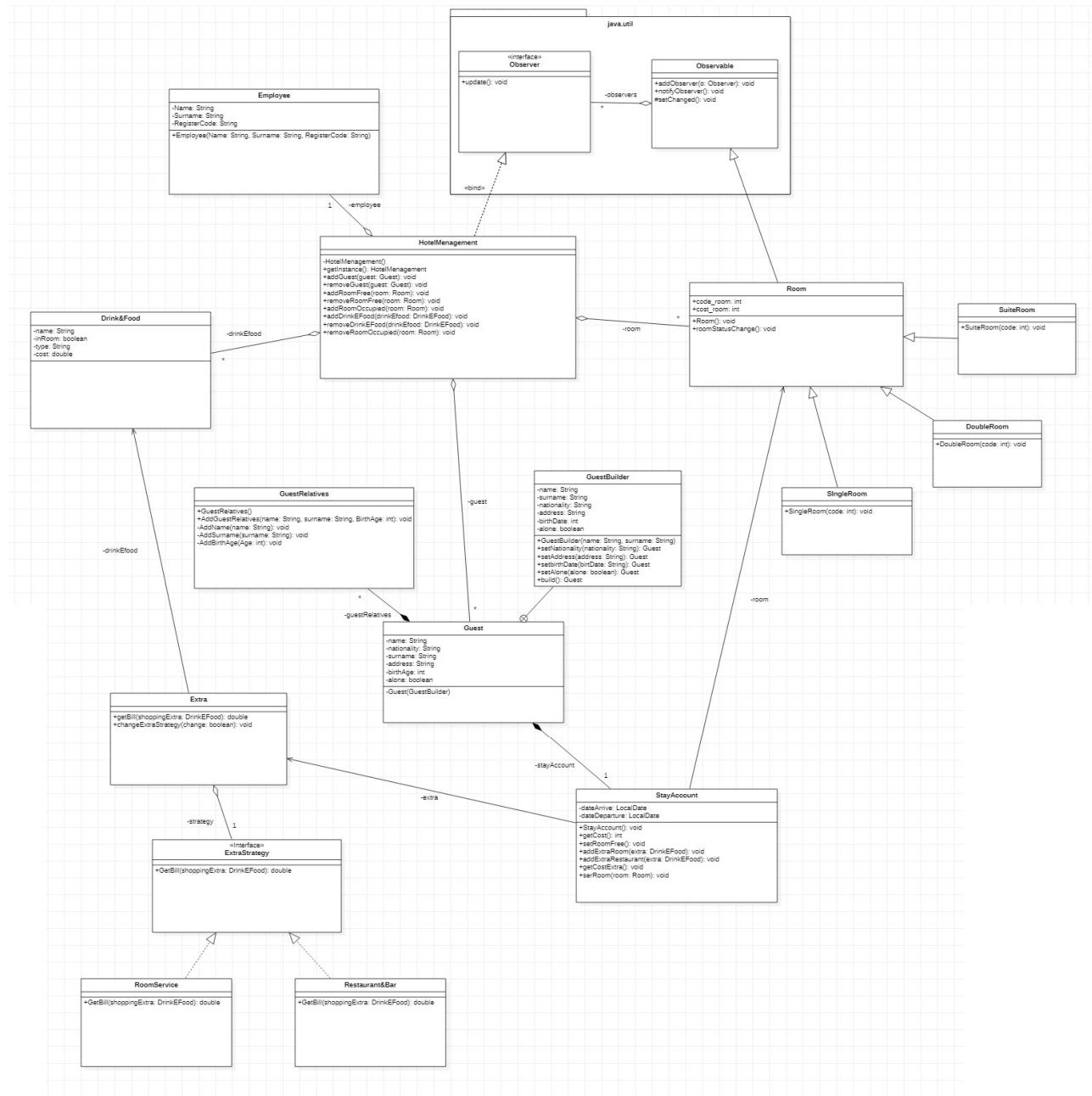
Definiamo in fase di progettazione gli attori in gioco e i vari casi di uso dell'applicazione descrivendoli attraverso uno UseCase Diagram.



**Figura 1.1:** Use Case Diagram.

## 1.2 DIAGRAMMA UML

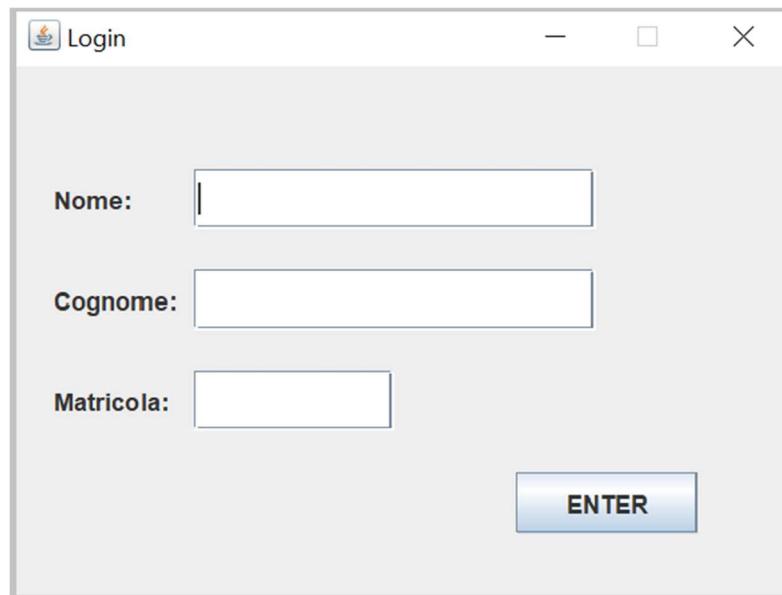
Definiamo ora, sempre in fase di progettazione, il diagramma UML che implementa la logica dei model, i quali comporranno la struttura portante della nostra applicazione.



**Figura 1.2:** Class Diagram.

### 1.3 MOCKUPS:

Seguono alcuni mockups relativi all'interfaccia dell'applicazione.



**Figura 1.3:** Pagina di Login per i dipendenti.

 HOTEL PRIMULA ROSSA

**HOME PAGE**

Impiegato: Mario Rossi

**LISTA CLIENTI:**

**LISTA FAMILIARI:**

Nome: <input type="text"/> Cognome: <input type="text"/> Nazionalità: <input type="text"/> Indirizzo: <input type="text"/> Anno di nascita: <input type="text"/>	Anno di nascita del familiare: <input type="text"/> Camera Cliente: <input type="text"/> Data arrivo: <input type="text"/> Data partenza: <input type="text"/>
--	---

**Figura 1.3:** Pagina Principale dell'applicazione, qui la vediamo inizialmente senza clienti, ma è possibile aggiungerli selezionando il tasto in basso a destra.

 Inserimento Cliente

Inserisci Nome:	<input type="text" value="Mario"/>		
Inserisci Cognome:	<input type="text" value="Rossi"/>		
Inserisci nazionalità:	<input type="text" value="Italiano"/>		
Inserisci indirizzo:	<input type="text" value="via Roma 13 Firer"/>	Data arrivo:	<input type="text" value="10 Agosto 20..."/>
Inserisci anno di nascita:	<input type="text" value="1985"/>	Data partenza:	<input type="text" value="20 Agosto 20..."/>
Assegna una camera:	<input type="text" value="Suite: 301"/>		
<input type="button" value="AGGIUNGI FAMILIARE"/>		<input type="button" value="SAVE"/>	

**Figura 1.4:** Pagina per l'inserimento di un utente.

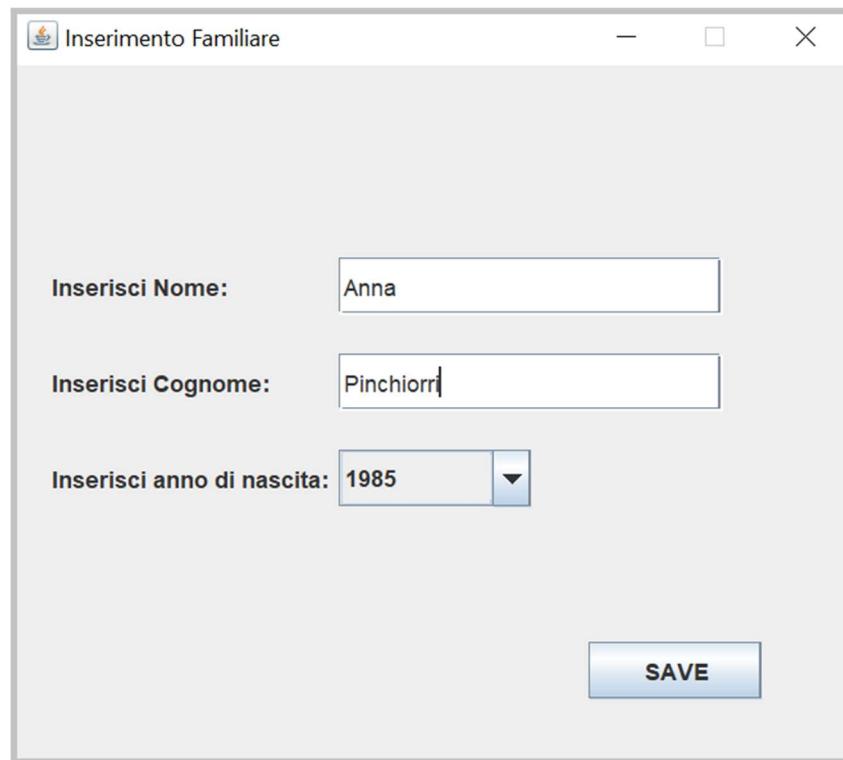
Inserimento Familiare

Inserisci Nome: Anna

Inserisci Cognome: Pinchiorri

Inserisci anno di nascita: 1985

**SAVE**



**Figura 1.5:** Pagina per l'inserimento di un familiare dell'utente.

HOTEL PRIMULA ROSSA

**HOME PAGE**

Impiegato: Mario Rossi

**LISTA CLIENTI:**

Mario Rossi
Arturo Verdi

**LISTA FAMILIARI:**

Anna Pinchiorri
-----------------

**Nome:** Mario

**Cognome:** Rossi

**Nazionalità:** italiana

**Anno di nascita del familiare:** 1985

**Indirizzo:** via Roma 13 Firenze

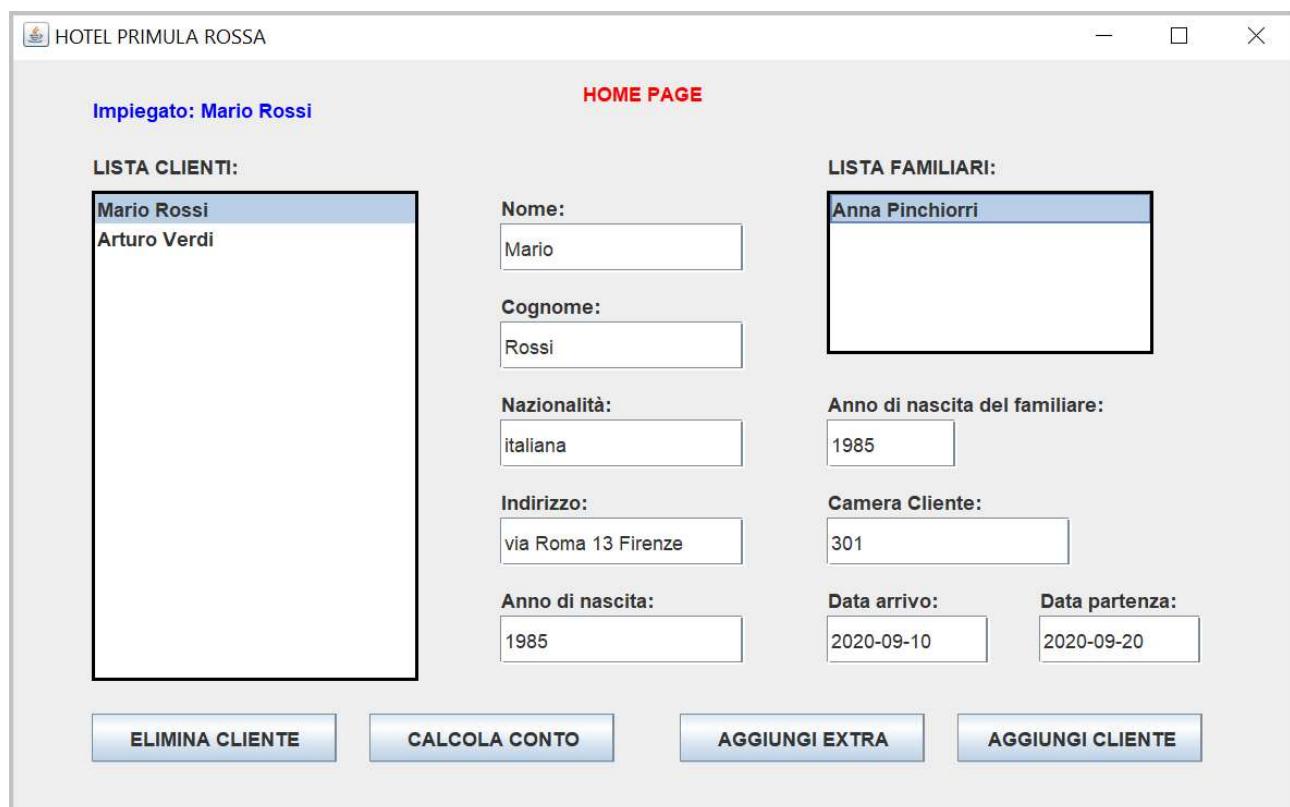
**Camera Cliente:** 301

**Anno di nascita:** 1985

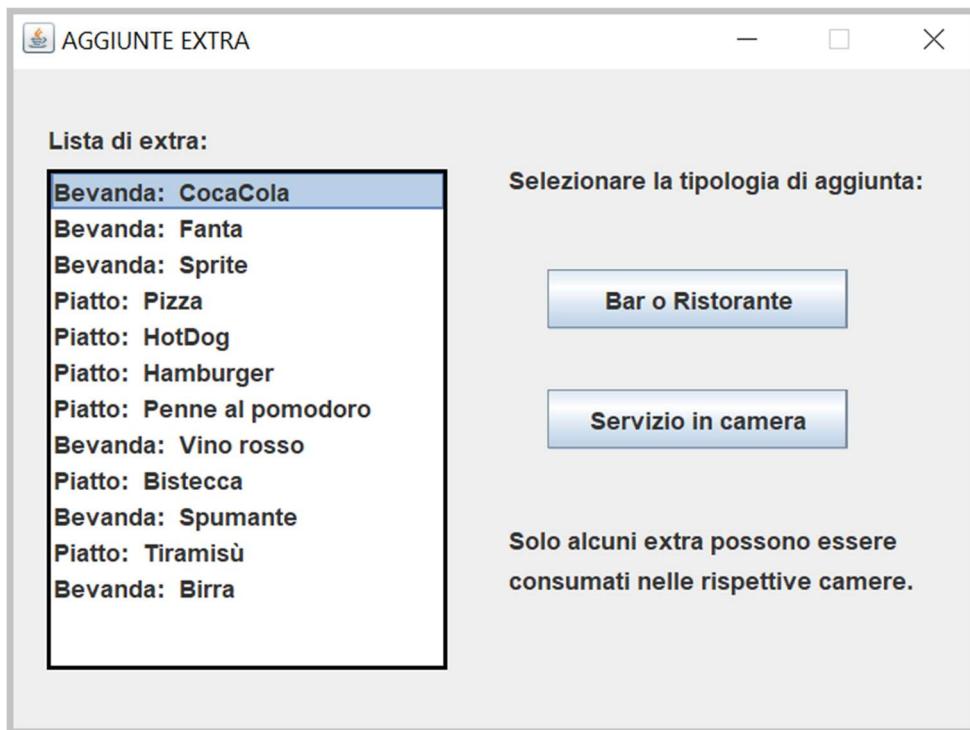
**Data arrivo:** 2020-09-10

**Data partenza:** 2020-09-20

**ELIMINA CLIENTE**   **CALCOLA CONTO**   **AGGIUNGI EXTRA**   **AGGIUNGI CLIENTE**



**Figura 1.6:** Pagina Principale dell'applicazione, qui la vediamo con due utenti già registrati, ed è ora possibile gestire i loro account.



**Figura 1.7:** Pagina per l'assegnazione di eventuali extra consumati dagli utenti.

The screenshot shows a window titled "Calcolo del Conto". It contains the following fields:

Nome:	Mario	Cognome:	Rossi
Camera:	301	Data arrivo:	2020-09-10
Lista di extra:		Data partenza:	2020-09-20
<b>Penne al pomodoro: 9.5</b> <b>Vino rosso: 11.0</b> <b>CocaCola: 2.5</b> <b>Pizza: 5.0</b>		In Camera:	NO
		Costo totale camera:	1400.0€
		Conto finale:	1432.0€

A button labeled "Calcola conto" is located at the bottom right. A note at the bottom states: "Si ricorda che il servizio in camera ha un addebito di 2 euro a prodotto."

**Figura 1.8:** Pagina per il calcolo del conto di ogni utente.

# Capitolo 2: Implementazione

## 2.1 CLASSI E INTERFACCE

Durante la fase di implementazione sono state definite e realizzate alcune classi ed interfacce, nonché i design patterns definiti in precedenza. Inoltre per la realizzazione del pattern Observer è stato sfruttato il package java.util. Le principali classi sono:

1. HotelManagement
2. Employee
3. Guest
4. GuestRelatives
5. StayAccount
6. DrinkEFood
7. Extra
8. Room (classe da cui ereditano tre sottoclassi Single, Double e Suite).
9. ExtraStrategy (interfaccia avente due diverse realizzazioni RoomService e Restaurant&Bar).

Oltre a queste classi ne sono state definite delle altre nei package src.model, src.controller e src.view. In particolare questi ultimi due package implementano il pattern MVC e permettono la gestione della parte grafica che viene controllata dalle classi definite nel package src.view, le quali estendono JFrame (all'interno del framework Java Swing). Inoltre viene definita un'altra classe detta ExtraRoomException per la gestione di una eccezione in particolare. La suddetta classe permette di gestire il problema che si pone quando, attraverso il metodo addExtraRoom, definito nella classe StayAccount, viene tentato l'inserimento fra i prodotti consumati in camera, da un determinato utente, di un prodotto che per definizione non può essere consegnato tramite servizio in camera.

### 2.1.1 HOTEL MANAGEMENT

La classe HotelManagement rappresenta l'Hotel, sarà quindi all'interno di questa classe che verrà memorizzato il riferimento all'oggetto che identifica il dipendente che ha effettuato l'accesso. Inoltre saranno presenti anche due liste per memorizzare le camere libere e le camere occupate, una lista che andrà a memorizzare le offerte che l'hotel propone per quanto riguarda bar, ristorante e servizio in camera, ed infine una lista che andrà a memorizzare i clienti che andranno a soggiornare all'interno della nostra struttura.

```
public class HotelManagement implements Observer{  
    private static HotelManagement hotel;  
    private static Employee employee;  
    private static List<Guest> guests;  
    private static List<Room> roomsFree;  
    private static List<Room> roomsOccupied;  
    private static List<DrinkEFood> drinksEfoods;  
  
    private HotelManagement() {  
        guests = new ArrayList<>();  
        roomsFree = new ArrayList<>();  
        roomsOccupied = new ArrayList<>();  
        drinksEfoods = new ArrayList<>();  
    }  
  
    public static HotelManagement getInstance() {  
        if(hotel == null)  
            hotel = new HotelManagement();  
        return hotel;  
    }  
  
    public List<Guest> getGuests() { return guests; }  
    public Employee getEmployee() { return employee; }  
    public void setEmployee(String name, String surname, String code) { employee = new Employee(name, surname, code); }  
    public List<Room> getRoomsFree() { return roomsFree; }  
    public List<Room> getRoomsOccupied() { return roomsOccupied; }  
    public List<DrinkEFood> getDrinksEfoods() { return drinksEfoods; }  
    public void addGuest(Guest guest) { guests.add(guest); }  
    public void removeGuest(Guest guest) {  
        if (guest != null) {  
            guests.remove(guest);  
        }  
    }  
    public void addRoomFree(Room room) { roomsFree.add(room); }  
    public void removeRoomFree(Room room) {  
        if (room != null) {  
            roomsFree.remove(room);  
        }  
    }  
    public void addRoomOccupied(Room room) { roomsOccupied.add(room); }  
    public void removeRoomOccupied(Room room) {  
        if (room != null) {  
            roomsOccupied.remove(room);  
        }  
    }  
    public void addDrinkEFood(DrinkEFood drinkEfood) { drinksEfoods.add(drinkEfood); }  
    public void removeDrinkEFood(DrinkEFood drinkEfood) {  
        if (drinkEfood != null) {  
            drinksEfoods.remove(drinkEfood);  
        }  
    }  
    public void updateHotel(Observable o) {  
        Room room = (Room) o;  
        if(room.free) {  
            removeRoomOccupied(room);  
            addRoomFree(room);  
        }  
        else {  
            removeRoomFree(room);  
            addRoomOccupied(room);  
        }  
    }  
    @Override  
    public void update(Observable o, Object arg) {  
        updateHotel(o);  
    }  
}
```

**Figura 2.1:** Codice della classe HotelManagement.

### 2.1.2 EMPLOYEE

La classe impiegato rappresenta l'istanza di un impiegato. Come già detto nella pagina precedente l'impiegato è memorizzato all'interno della classe HotelManagement. Sarà l'impiegato colui che interagirà e gestirà il nostro applicativo in ogni sua parte.

```
package model;

public class Employee {

    private final String Name;
    private final String Surname;
    private final String RegisterCode;

    public Employee(String name, String surname, String registerCodeField) {
        Name = name;
        Surname = surname;
        RegisterCode = registerCodeField;
    }

    public String getName() { return Name; }

    public String getSurname() { return Surname; }

    public String getRegisterCode() { return RegisterCode; }
}
```

**Figura 2.2:** Codice della classe Employee.

### 2.1.3 GUEST

La classe guest rappresenta un cliente che viene registrato all'interno della struttura. Verranno quindi registrati i dati anagrafici del cliente. Vengono inoltre create due istanze che andranno a gestire l'account dell'utente (StayAccount) e la relazione dell'utente con eventuali accompagnatori (GuestRelatives). La lista dei clienti, come già scritto in precedenza, si trova all'interno della classe HotelMenagement.

```
package model;

public class Guest{
    private String name;
    private String surname;
    private String nationality;
    private String address;
    private int birthAge;
    private boolean alone;
    private GuestRelatives guestRelatives;
    private StayAccount stayAccount;

    private Guest(GuestBuilder builder) {
        name = builder.name;
        surname = builder.surname;
        nationality = builder.nationality;
        address = builder.address;
        birthAge = builder.birthAge;
        alone = builder.alone;
        stayAccount = new StayAccount();
        guestRelatives = new GuestRelatives();
    }

    public String getName() { return name; }

    public String getSurname() { return surname; }

    public String getNationality() { return nationality; }

    public String getAddress() { return address; }

    public int getBirthAge() { return birthAge; }

    public boolean isAlone() { return alone; }

    public void setAlone(boolean alone) { this.alone = alone; }

    public GuestRelatives getGuestRelatives() { return guestRelatives; }

    public StayAccount getStayAccount() { return stayAccount; }
}
```

**Figura 2.3:** Codice della classe Guest.

#### 2.1.4 GUEST RELATIVES

La classe GuestRelatives rappresenta i parenti o gli eventuali accompagnatori del cliente principale. Al suo interno vengono istanziate e gestite delle liste per la memorizzazione dei loro dati anagrafici.

```
package model;

import java.util.ArrayList;

public class GuestRelatives {

    private List<String> name;
    private List<String> surname;
    private List<Integer> birthAge;

    public GuestRelatives() {
        name = new ArrayList<>();
        surname = new ArrayList<>();
        birthAge = new ArrayList<>();
    }

    public List<String> getName() { return name; }

    public List<String> getSurname() { return surname; }

    public List<Integer> getBirthAge() { return birthAge; }

    public void AddGuestRelatives(String Name, String Surname, int Age) {
        AddName(Name);
        AddSurname(Surname);
        AddBirthAge(Age);
    }

    private void AddName(String Name) { name.add(Name); }

    private void AddSurname(String Surname) { surname.add(Surname); }

    private void AddBirthAge(int Age) { birthAge.add(Age); }
}
```

**Figura 2.4:** Codice della classe GuestRelatives.

## 2.1.5 STAY ACCOUNT

StayAccount è la classe che si occupa di memorizzare e gestire le informazioni relative al soggiorno dell'utente. Tiene quindi nota della data di arrivo e di quella di partenza, della camera assegnata e delle consumazioni fatte al bar, al ristorante oppure ordinate tramite servizio in camera (grazie a due liste definite al suo interno), ed infine gestisce anche il calcolo del conto finale da presentare all'utente (Gestendo separatamente il calcolo del costo della camera e quello del costo degli extra).

```
public class StayAccount {

    private LocalDate dateArrive;
    private LocalDate dateDeparture;
    private Room room;
    private List<DrinkEFood> roomExtra;
    private List<DrinkEFood> restaurantExtra;
    private Extra extra;

    public StayAccount() {
        roomExtra = new ArrayList<DrinkEFood>();
        restaurantExtra = new ArrayList<DrinkEFood>();
        extra = new Extra(); }

    public void setDateArrive(LocalDate dateArrive) { this.dateArrive = dateArrive; }

    public void setDateDeparture(LocalDate dateDeparture) { this.dateDeparture = dateDeparture; }

    @SuppressWarnings("deprecation")
    public void setRoom(Room room) {
        this.room = room;
        room.roomStatusChange();
        room.notifyObservers(); }

    public Room getRoom() { return room; }

    @SuppressWarnings("deprecation")
    public void setRoomFree() {
        room.roomStatusChange();
        room.notifyObservers(); }

    public int getCost() {
        long daysBetween = ChronoUnit.DAYS.between(dateArrive, dateDeparture);
        int tmpDays = (int) daysBetween;
        int tmp = room.cost_room * tmpDays;
        return tmp; }

    public void addExtraRoom(DrinkEFood extra) throws ExtraRoomException{
        if(extra.getInRoom()) {
            roomExtra.add(extra);
        } else {
            throw new ExtraRoomException();
        } }

    public void addExtraRestaurant(DrinkEFood extra) { restaurantExtra.add(extra); }

    public double getCostExtra() {
        double tmp;
        tmp = extra.getBill(restaurantExtra);
        extra.changeExtraStrategy(true);
        tmp = tmp + extra.getBill(roomExtra);
        extra.changeExtraStrategy(false);
        return tmp; }

    public List<DrinkEFood> getRoomExtra() { return roomExtra; }

    public LocalDate getDateArrive() { return dateArrive; }

    public LocalDate getDateDeparture() { return dateDeparture; }

    public List<DrinkEFood> getRestaurantExtra() { return restaurantExtra; }
}
```

Figura 2.5: Codice della classe StayAccount.

### 2.1.6 ROOM

La classe Room è la classe che definisce le varie camere dell’hotel. Si tratta della classe madre da cui ereditano le sottoclassi SingleRoom, DoubleRoom e SuiteRoom, che identificano le varie tipologie di camere presenti nell’hotel.

```
package model;

import java.util.Observable;

@SuppressWarnings("deprecation")
public class Room extends Observable{

    public int code_room;
    public int cost_room;
    public boolean free;

    public Room() {
        free = true;
    }

    public void roomStatusChange() {
        if(free) {
            free=false;
        }
        else {
            free=true;
        }
        setChanged();
    }
}
```

**Figura 2.6:** Codice della classe Room.

### 2.1.7 SINGLE ROOM

La classe SingleRoom è una delle classi che ereditano da Room. Questa classe identifica una stanza singola del nostro albergo, definendone prezzo, codice di riconoscimento e se è occupata oppure no.

```
package model;

public class SingleRoom extends Room {

    public SingleRoom(int code) {
        code_room = code;
        cost_room = 50;
    }
}
```

**Figura 2.7:** Codice della classe SingleRoom.

### 2.1.8 DOUBLE ROOM

La classe DoubleRoom è una delle classi che ereditano da Room. Questa classe identifica una stanza doppia del nostro albergo, definendone prezzo, codice di riconoscimento e se è occupata oppure no.

```
package model;

public class DoubleRoom extends Room {

    public DoubleRoom(int code) {
        code_room = code;
        cost_room = 90;
    }
}
```

**Figura 2.8:** Codice della classe DoubleRoom.

### 2.1.9 SUITE ROOM

La classe SuiteRoom è una delle classi che ereditano da Room. Questa classe identifica una suite del nostro albergo, definendone prezzo, codice di riconoscimento e se è occupata oppure no.

```
package model;

public class SuiteRoom extends Room {

    public SuiteRoom(int code) {
        code_room = code;
        cost_room = 140;
    }
}
```

**Figura 2.8:** Codice della classe SuiteRoom.

### 2.1.10 DRINK E FOOD

La classe DrinkEFood rappresenta un prodotto presente tra quelli consumabili all'interno del menù e possibilmente anche consumabile nelle camere. La classe definisce il nome dell'alimento o della bibita, il tipo (Piatto oppure bibita), il costo ed infine la possibilità o meno della consegna in camera di quel prodotto. La lista degli elementi presenti sul menù è memorizzata, come già detto in precedenza, all'interno di HotelManagement.

```
package model;

public class DrinkEFood {

    private String name;
    private String type;
    private boolean inRoom;
    private double cost;

    public DrinkEFood(String name, String type, boolean inRoom, double cost) {
        this.name = name;
        this.type = type;
        this.inRoom = inRoom;
        this.cost = cost;
    }

    public String getName() { return name; }

    public String getType() { return type; }

    public boolean getInRoom() { return inRoom; }

    public double getCost() { return cost; }

}
```

**Figura 2.9:** Codice della classe DrinkEFood.

### 2.1.11 EXTRA

La classe Extra si occupa della gestione e del calcolo di eventuali sovrapprezzi, da aggiungere al conto finale, dovuti a consumazioni al Bar e/o al Ristorante, oppure dovuti ad ordini effettuati in camera.

```
package model;

import java.util.List;

public class Extra {

    ExtraStrategy extraStrategy;

    public Extra() {
        extraStrategy = new RestaurantEBar();
    }

    public double getBill(List<DrinkEFood> shoppingExtra) {
        return extraStrategy.getBill(shoppingExtra);
    }

    public void changeExtraStrategy(boolean change) {
        if(change) {
            extraStrategy = new RoomService();
        }
        else {
            extraStrategy = new RestaurantEBar();
        }
    }
}
```

**Figura 2.10:** Codice della classe Extra.

## 2.2 DESIGN PATTERN

Sono stati implementati ed utilizzati alcuni Design Pattern per la realizzazione di questo progetto. I pattern utilizzati sono:

1. Pattern Singleton
2. Strategy Pattern
3. Builder
4. Observer
5. MVC

### 2.2.1 SINGLETON

Il Singleton è un pattern creazionale che permette di avere una e una sola istanza di una determinata classe. Per permettere ciò il costruttore della nostra classe viene dichiarato privato e viene definito un metodo statico getInstance che restituirà un'unica istanza della classe. Nel progetto la classe HotelManagement che rappresenta l'hotel è stata realizzata come Singleton.

```
private HotelManagement() {  
    guests = new ArrayList<>();  
    employees = new ArrayList<>();  
    roomsFree = new ArrayList<>();  
    roomsOccupied = new ArrayList<>();  
    drinksEfoods = new ArrayList<>();  
}  
  
public static HotelManagement getInstance() {  
    if(hotel == null)  
        hotel = new HotelManagement();  
  
    return hotel;  
}
```

**Figura 2.11:** Frammento di codice della classe HotelManagement.

## 2.2.2 STRATEGY PATTERN

Lo strategy è un pattern comportamentale utilizzato per richiamare un metodo, all'interno di un oggetto, che avrà un comportamento diverso a seconda di una strategia scelta a run time, mediante il metodo changeExtraStrategy (nel nostro caso). Nel nostro caso ogni volta che il sistema dovrà calcolare il costo degli extra consumati da ogni cliente verrà utilizzato il metodo getBill presente all'interno della classe Extra. Il metodo getBill permetterà, appunto, di calcolare il costo degli extra che potranno essere di due tipi, consumati in camera oppure in loco (il calcolo sarà quindi differente per le due tipologie di extra). Per permettere questa distinzione nel tipo di calcolo è presente all'interno della classe Extra un campo di tipo ExtraStrategy che, a seconda di che tipo di calcolo è richiesto, permetterà di cambiare la strategia (attraverso il metodo changeExtraStrategy presente in Extra) e di richiamare uno dei due metodi presenti all'interno delle classi RoomService e RestaurantEBar.

```
package model;

import java.util.List;

public class Extra {

    ExtraStrategy extraStrategy;

    public Extra() {
        extraStrategy = new RestaurantEBar();
    }

    public double getBill(List<DrinkEFood> shoppingExtra) {
        return extraStrategy.getBill(shoppingExtra);
    }

    public void changeExtraStrategy(boolean change) {
        if(change) {
            extraStrategy = new RoomService();
        }
        else {
            extraStrategy = new RestaurantEBar();
        }
    }
}
```

**Figura 2.12:** Codice della classe Extra.

```

package model;

import java.util.List;

public interface ExtraStrategy {
    public double getBill(List<DrinkFFood> shoppingExtra);
}

```

**Figura 2.13:** Codice della classe ExtraStrategy.

```

package model;

import java.util.List;

public class RestaurantEBar implements ExtraStrategy{

    @Override
    public double getBill(List<DrinkFFood> shoppingExtra) {

        double tmp = 0;
        if(shoppingExtra.size() == 0) {
            return 0;
        }
        else {
            for(int i = 0; i < shoppingExtra.size(); i++) {
                tmp = tmp + shoppingExtra.get(i).getCost();
            }
        }
        return tmp;
    }
}

```

**Figura 2.14:** Codice della classe RestaurantEBar.

```

package model;

import java.util.List;

public class RoomService implements ExtraStrategy{

    @Override
    public double getBill(List<DrinkFFood> shoppingExtra) {

        double tmp = 0;
        if(shoppingExtra.size() == 0) {
            return 0;
        }
        else {
            for(int i = 0; i < shoppingExtra.size(); i++) {
                tmp = tmp + shoppingExtra.get(i).getCost();
                tmp = tmp + 2;
            }
        }
        return tmp;
    }
}

```

**Figura 2.15:** Codice della classe RoomService.

### 2.2.3 BUILDER

Il builder è un pattern creazionale utilizzato per la creazione di oggetti che hanno numerosi attributi. Nel nostro caso il builder viene utilizzato per facilitare la creazione di ogni istanza della classe Guest, viene perciò creata una classe GuestBuilder a cui si delega la creazione di ogni cliente attraverso il metodo statico build.

```
public static class GuestBuilder {
    private final String name;
    private final String surname;
    private String nationality;
    private String address;
    private int birthAge;
    private boolean alone;

    public GuestBuilder(String name, String surname) {
        this.name=name;
        this.surname=surname;
    }

    public Guest build() { return new Guest(this); }

    public GuestBuilder setNationality(String nationality) {
        this.nationality = nationality;
        return this;
    }

    public GuestBuilder setAddress(String address) {
        this.address = address;
        return this;
    }

    public GuestBuilder setBirthAge(int birthAge) {
        this.birthAge = birthAge;
        return this;
    }

    public GuestBuilder setAlone(boolean alone) {
        this.alone = alone;
        return this;
    }
}
```

**Figura 2.16:** Codice della classe GuestBuilder.

#### 2.2.4 OBSERVER

L'observer è un pattern comportamentale utilizzato quando uno o più oggetti (Observer) devono monitorare un altro oggetto (Subject). Il subject mantiene al suo interno una lista di oggetti observers ed è suo compito notificare ai suoi osservatori ogni volta che esso cambia stato. Per l'implementazione sono state utilizzate la classe Observable e l'interfaccia Observer entrambe contenute nel package java.util. In particolare la classe che va ad estendere la classe Subject, in questo progetto, è la classe Room, mentre la classe che va ad estendere la classe Observer è la classe HotelManagement.

```
@Override  
public void update(Observable o, Object arg) {  
    updateHotel(o); }
```

**Figura 2.17:** Frammento di codice della classe HotelManagement, ovvero la classe che implementa l'Observer. Si va qui a fare l'Override del metodo update che andrà a richiamare il metodo updateHotel nel momento in cui una modifica è stata apportata al Subject.

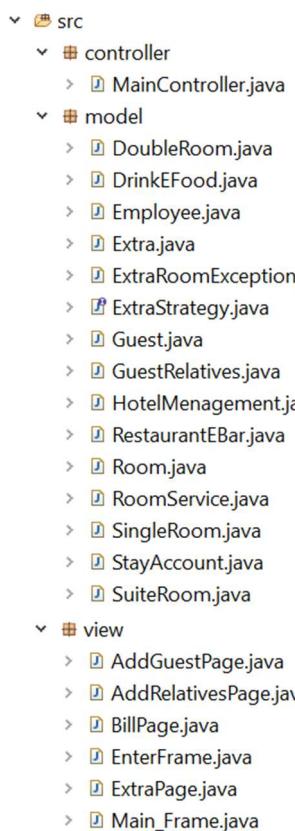
```
public void roomStatusChange() {  
    if(free) {  
        free=false;  
    }  
    else {  
        free=true;  
    }  
    setChanged();  
}
```

**Figura 2.18:** Frammento di codice della classe Room, ovvero la classe che estende il Subject. Si vede qui che una volta effettuata una modifica si va a chiamare setChanged(). Sarà poi richiamato in seguito NotifyObservers() per andare a notificare la modifica all'Observer (nel nostro caso a HotelManagement).

## 2.2.5 MVC

MVC (Model-View-Controller) è un pattern architetturale che, separando la nostra applicazione in più parti (generalmente si separa la logica di dominio, ovvero i model, dalla logica legata alla visualizzazione grafica dell'applicazione, ovvero le View), permette una maggiore efficienza di gestione del codice. Per fare ciò, in questo applicativo, il codice è stato suddiviso in 3 package principali:

1. **Model:** il modello fornisce la struttura principale dell'applicazione. Fornisce cioè tutti i metodi che permettono di accedere e rielaborare i dati della nostra applicazione. In questo caso il model coincide con le classi del progetto contenute nel package src.model.
2. **Controller:** il controller svolge il ruolo di mediatore tra i model e le view. Riceve i comandi dall'utente ed in base ad essi modifica lo stato degli altri due componenti. In questo caso il controller coincide con la classe MainController presente nel package src.controller.
3. **View:** le classi view hanno il compito di visualizzare i dati del model e di interagire con l'utente. In questo caso tutte le classi riguardanti l'interfaccia grafica utente (GUI) sono contenute nel package src.view.



**Figura 2.19:** Schema dell'organizzazione del codice fra i diversi package.

# Capitolo 3: Testing

## 3 UNIT TESTING

Lo Unit Testing è la verifica di singole porzioni di codice che avviene tramite test mirati. Nel caso dell'Object-Oriented Programming (OOP) tipicamente si testano le singole classi e i singoli metodi. Nel progetto ciò è stato fatto utilizzando il framework JUnit nella versione 5.0. Per ogni classe principale del progetto sono state create le rispettive classi di Test contenenti i test case relativi ai metodi della classe da testare. In ogni metodo dei test vengono verificate delle asserzioni (asserts) elementari attraverso vari metodi offerti da JUnit stesso come assertEquals, assertTrue, assertFalse, assertNull ed altre varianti.

Le classi di Test realizzate nel progetto sono:

1. HotelManagementTest
2. EmployeeTest
3. GuestTest
4. GuestRelativesTest
5. DrinkEFoodTest
6. ExtraTest
7. RestaurantEBarTest
8. RoomServiceTest
9. StayAccountTest

### 3.1 HOTEL MENAGEMENT TEST

Testo il corretto funzionamento dei singoli metodi della classe HotelMenagement ed in più testo anche il corretto funzionamento del Pattern Singleton (creando una istanza di HotelMenagement e assicurandomi che nel richiamare la funzione getInstance() l'istanza restituita sia sempre la stessa). Testo infine il corretto funzionamento del pattern Builder creando tramite quest'ultimo una istanza di Guest, ovvero di un cliente, ed assicurandomi che i dati siano assegnati correttamente.

```
HotelMenagement hotel;

@BeforeEach
void setUp() {
    hotel = HotelMenagement.getInstance();
    hotel.setEmployee("Mario", "Rossi", "001"); }

@Test
void getInstance() {
    assertEquals(hotel, HotelMenagement.getInstance()); }

@Test
void getEmployee() {
    assertEquals(hotel.getEmployee().getName(), "Mario");
    assertEquals(hotel.getEmployee().getSurname(), "Rossi");
    assertEquals(hotel.getEmployee().getRegisterCode(), "001"); }

@Test
void addRoom() {
    hotel.addRoomFree(new DoubleRoom(102));
    hotel.addRoomFree(new SingleRoom(101));
    assertEquals(hotel.getRoomsFree().get(0).code_room, 102);
    assertEquals(hotel.getRoomsFree().get(1).code_room, 101); }

@Test
void addRoom() {
    hotel.addRoomFree(new DoubleRoom(102));
    hotel.addRoomFree(new SingleRoom(101));
    assertEquals(hotel.getRoomsFree().get(0).code_room, 102);
    assertEquals(hotel.getRoomsFree().get(1).code_room, 101); }

@Test
void BuilderTest() {
    hotel.addGuest(new Guest.GuestBuilder("Mauro", "Bianchi")
        .setNationality("italiana")
        .setAddress("via Roma, Firenze")
        .setBirthAge(1978)
        .setAlone(true).build());
    assertEquals(hotel.getGuests().get(0).getName(), "Mauro");
    assertEquals(hotel.getGuests().get(0).getSurname(), "Bianchi");
    assertEquals(hotel.getGuests().get(0).getNationality(), "italiana");
    assertEquals(hotel.getGuests().get(0).getAddress(), "via Roma, Firenze");
    assertEquals(hotel.getGuests().get(0).getBirthAge(), 1978);
    assertTrue(hotel.getGuests().get(0).isAlone()); }

@Test
void addDrinkEFood() {
    hotel.addDrinkEFood( new DrinkEFood("pizza", "food", true, 8));
    assertEquals(hotel.getDrinksEfoods().get(0).getName(), "pizza");
    assertEquals(hotel.getDrinksEfoods().get(0).getType(), "food");
    assertTrue(hotel.getDrinksEfoods().get(0).getInRoom());
    assertEquals(hotel.getDrinksEfoods().get(0).getCost(), 8); }
```

Figura 3.1: Codice della classe HotelMenagementTest.

### 3.2 GUEST TEST

Testo il corretto funzionamento dei singoli metodi della classe Guest andando a settare i vari attributi della suddetta classe ed assicurandomi il corretto funzionamento dei metodi definiti.

```
Guest guest;

@BeforeEach
void setUp() {
    guest = new Guest.GuestBuilder("Luca", "Verdi")
        .setNationality("italiana")
        .setAddress("via Roma, Firenze")
        .setBirthAge(1978)
        .setAlone(true).build(); }

@Test
void getName() {
    assertEquals(guest.getName(), "Luca"); }

@Test
void getSurname() {
    assertEquals(guest.getSurname(), "Verdi"); }

@Test
void getRelatives() {
    guest.getGuestRelatives().AddGuestRelatives("Simone", "Neri", 1987);
    assertEquals(guest.getGuestRelatives().getName().get(0), "Simone");
    assertEquals(guest.getGuestRelatives().getSurname().get(0), "Neri");
    assertEquals(guest.getGuestRelatives().getBirthAge().get(0), 1987); }

@Test
void getStayAccount() {
    guest.getStayAccount(). setDateArrive(LocalDate.of(2019, 10, 19));
    assertEquals(guest.getStayAccount().getDateArrive(), LocalDate.of(2019, 10, 19)); }
```

**Figura 3.2:** Codice della classe GuestTest.

### 3.3 DRINK E FOOD TEST

Testo il corretto funzionamento dei singoli metodi della classe DrinkEFood andando a creare una istanza della suddetta classe ed assicurandomi tramite i metodi definiti al suo interno che i dati ed essa assegnati siano stati definiti correttamente.

```
class DrinkEFoodTest {  
    DrinkEFood extra;  
  
    @BeforeEach  
    void setUp() {  
        extra = new DrinkEFood("pizza", "food", true, 5); }  
  
    @Test  
    void getName() {  
        assertEquals(extra.getName(), "pizza"); }  
  
    @Test  
    void getType() {  
        assertEquals(extra.getType(), "food"); }  
  
    @Test  
    void getCost() {  
        assertEquals(extra.getCost(), 5); }  
  
    @Test  
    void getInRoom() {  
        assertTrue(extra.getInRoom()); }  
}
```

**Figura 3.3:** Codice della classe DrinkEFoodTest.

### 3.4 EXTRA TEST

Testo il corretto funzionamento dei singoli metodi della classe Extra, inoltre vado a testare anche il corretto funzionamento del Pattern Strategy. Il Pattern Strategy viene testato assegnando al conto di un utente un prodotto consumato in camera ed uno consumato al tavolo, viene in seguito calcolato il conto finale delle consumazioni e viene confrontato (tramite un assertEquals) con una previsione della cifra finale (ovvero consumazione al tavolo + consumazione in camera con maggiorazione di 2€).

```
class ExtraTest {  
    Extra extra;  
    List<DrinkFood> foodRestaurant = new ArrayList<>();  
    List<DrinkEFood> foodRoom = new ArrayList<>();  
  
    @BeforeEach  
    void setUp() {  
        extra = new Extra();  
        foodRestaurant.add( new DrinkEFood("pollo", "food", true, 10));  
        foodRestaurant.add( new DrinkEFood("pizza", "food", false, 5));  
        foodRoom.add( new DrinkEFood("fanta", "drink", true, 2));  
        foodRoom.add( new DrinkEFood("pasta", "food", true, 8));  
    }  
  
    @Test  
    void strategyTest() {  
        assertEquals(extra.getBill(foodRestaurant), 15);  
        extra.changeExtraStrategy(true);  
        assertEquals(extra.getBill(foodRoom), 14);  
    }  
}
```

**Figura 3.4:** Codice della classe ExtraTest.

### 3.5 RESTAURANT E BAR TEST

Testo il corretto funzionamento della classe RestaurantEBar andando a creare una istanza della suddetta classe e controllando il corretto funzionamento del metodo definito al suo interno.

```
class RestaurantEBarTest {

    RestaurantEBar restaurant;
    List<DrinkEFood> extra = new ArrayList<>();

    @BeforeEach
    void setUp() {
        restaurant = new RestaurantEBar();
    }

    @Test
    void roomServiceTest() {
        extra.add( new DrinkEFood("pizza", "food", true, 5));

        assertEquals(restaurant.getBill(extra), 5);
    }
}
```

**Figura 3.5:** Codice della classe RestaurantEBarTest.

### 3.6 ROOM SERVICE TEST

Testo il corretto funzionamento della classe RoomService andando a creare una istanza della suddetta classe e controllando il corretto funzionamento del metodo definito al suo interno.

```
class RoomServiceTest {

    RoomService roomService;
    List<DrinkEFood> extra = new ArrayList<>();

    @BeforeEach
    void setUp() {
        roomService = new RoomService();
    }

    @Test
    void roomServiceTest() {
        extra.add( new DrinkEFood("pizza", "food", true, 5));

        assertEquals(roomService.getBill(extra), 7);
    }
}
```

**Figura 3.6:** Codice della classe RoomServiceTest.

### 3.7 STAY ACCOUNT TEST

Testo il corretto funzionamento dei singoli metodi della classe StayAccount, il funzionamento della gestione dell’eccezione (ExtraRoomException) tramite il metodo assertThrows (ovvero vado ad aggiungere ai prodotti consegnati in camera di un utente un prodotto che non può essere consegnato in camera e mi assicuro che venga lanciata la relativa eccezione) ed infine testo anche il Pattern Observer. Il Pattern Observer viene testato assegnando una camera ad un utente ed assicurandosi che la camera, registrata dentro a HotelManagement, venga spostata e messa nella lista delle camere occupate.

```
HotelManagement hotel;
Guest guest;

@BeforeEach
void setUp() {
    hotel = HotelManagement.getInstance();
    guest = new Guest.GuestBuilder("Mauro", "Bianchi")
        .setNationality("italiana")
        .setAddress("via Roma, Firenze")
        .setBirthAge(1978)
        .setAlone(true).build(); }

@SuppressWarnings("deprecation")
@Test
void ObserverTest() {
    Room room = new SingleRoom(101);
    room.addObserver(hotel);
    hotel.addRoomFree(room);

    guest.getStayAccount().setRoom(room);

    assertEquals(guest.getStayAccount().getRoom().code_room, 101);
    assertEquals(hotel.getRoomsOccupied().get(0).code_room, 101);

    guest.getStayAccount().setRoomFree();
    assertEquals(hotel.getRoomsFree().get(0).code_room, 101); }

@SuppressWarnings("deprecation")
@Test
void TestExtraCost() throws ExtraRoomException {
    Room room = new SingleRoom(101);
    room.addObserver(hotel);
    hotel.addRoomFree(room);
    guest.getStayAccount().setRoom(room);

    DrinkEFood extra = new DrinkEFood("pizza", "food", false, 8);
    guest.getStayAccount().addExtraRestaurant(extra);
    assertEquals(guest.getStayAccount().getRestaurantExtra().get(0).getName(), "pizza");

    DrinkEFood extraRoom = new DrinkEFood("Spumante", "drink", true, 15);
    guest.getStayAccount().addExtraRoom(extraRoom);

    assertEquals(guest.getStayAccount().getCostExtra(), 25);
    assertThrows(ExtraRoomException.class, () -> guest.getStayAccount().addExtraRoom(extra)); }

@SuppressWarnings("deprecation")
@Test
void TestCost() {
    Room room = new SingleRoom(101);
    room.addObserver(hotel);
    hotel.addRoomFree(room);
    guest.getStayAccount().setRoom(room);

    guest.getStayAccount(). setDateArrive(LocalDate.of(2019, 10, 19));
    guest.getStayAccount(). setDateDeparture(LocalDate.of(2019, 10, 28));

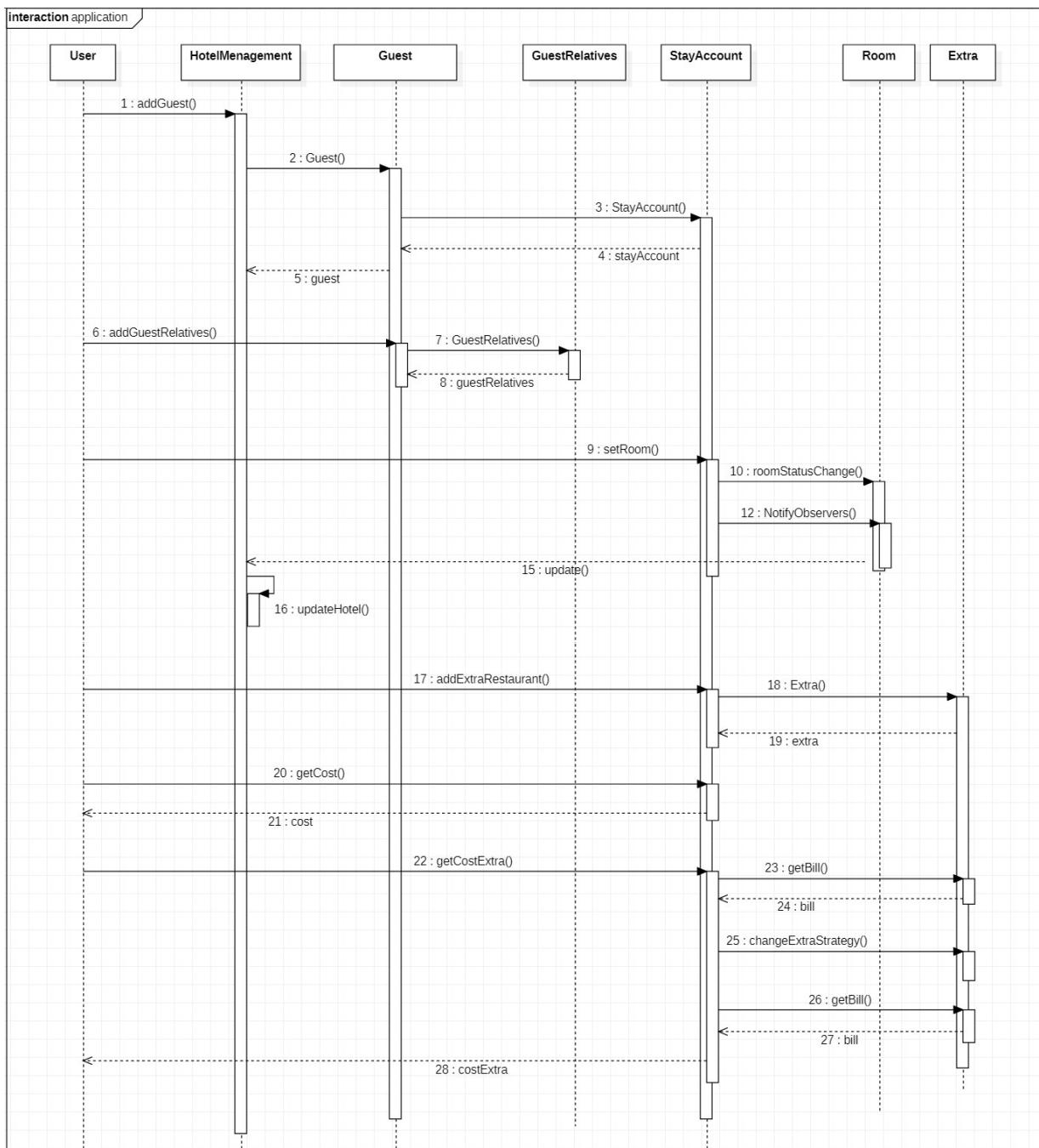
    assertEquals(guest.getStayAccount().getDateArrive(), LocalDate.of(2019, 10, 19));
    assertEquals(guest.getStayAccount().getDateDeparture(), LocalDate.of(2019, 10, 28));
    assertEquals(guest.getStayAccount().getCost(), 450); }
```

**Figura 3.7:** Codice della classe StayAccountTest.

# Capitolo 4: Sequence Diagram

## 4 SEQUENCE DIAGRAM

Viene ora proposta, grazie ad un Sequence Diagram, una simulazione del flusso del programma. Nella simulazione viene registrato un cliente, gli viene assegnata una camera e, considerando anche una eventuale consumazione, viene calcolato il conto finale che il cliente dovrà pagare.



**Figura 4.1:** Sequence Diagram.