Applicativo Java per la gestione di un ristorante

Elaborato Ingegneria del Software

Gianni Moretti, Francesco Fantechi, Francesco Bettazzi

A.A. 2020-2021

UNIVERSITA' DEGLI STUDI DI FIRENZE
Facolta di Ingegneria
Corso di Laurea in Ingegneria Informatica

Contents

1 Motivazione e Descrizione

Il presente elaborato nasce dall'idea di uno di noi ragazzi che, lavorando in una pizzeria come cameriere, si è trovato ad interagire con un applicativo simile. Abbiamo così pensato di riprodurne uno personalizzato che preveda inoltre, considerato il periodo particolare che stiamo vivendo, la possibilità di gestire azioni atte alla sicurezza dei clienti, come il loro monitoraggio per poterli rintracciare.
L'applicativo ha lo scopo di gestire le varie parti che compongono un ristorante, in modo da farle interagire e collaborare assieme. Abbiamo individuato le cinque figure professionali principali che possono trovarsi ad agire all'interno di un ristorante:

1. L'organizzatore della sala: è colui che all'arrivo di nuovi clienti ha il compito di assegnare loro un tavolo. Ciò può essere realizzato utilizzando i tavoli singoli già disposti in sala ad inizio serata o aggregandone alcuni se il numero di clienti è elevato. Ha quindi la possibilità di contrassegnare come occupati i tavoli assegnati e di rimetterli disponibili una volta terminato il servizio e igienizzato il tavolo. Inoltre, in vista delle norme di distanziamento imposte dal periodo che stiamo vivendo, l'organizzatore della sala può decidere se e quali tavoli sono o no utilizzabili.

2. Il cameriere: ha il compito di gestire i vari servizi ai tavoli, ossia di prendere le comande e mandarle alla cucina per la loro realizzazione. Le comande sono composte da un insieme di piatti presenti nel menù del locale, da dei commenti opzionali sui piatti per la cucina e da un insieme di ingredienti aggiunti/rimossi dalle varie portate su preferenza e richiesta del cliente. In caso di errore di immissione della comanda o di richiesta di modifica da parte del cliente, il cameriere può eliminare le portate sbagliate in modo da correggere correttamente il conto. Per aprire un servizio al tavolo il cameriere è tenuto a prendere un nominativo ed un recapito telefonico di uno dei presenti in modo da tenere traccia di chi ha frequentato il locale ed essere in grado di rintracciare i clienti in caso di necessità.

3. La cucina: i cuochi ottengono le comande confermate dai camerieri e possono spuntarle una volta preparate le pietanze. La cucina si può inoltre specializzare in più aree di competenza come, ad esempio, la pizzeria e il bar.

4. Il cassiere: è colui che al termine del servizio a un tavolo esegue il conto, contrassegna il tavolo come da pulire ed igienizzare e si occupa di memorizzare correttamente i dati dei clienti presenti.

5. Il gestore del ristorante: è colui che ha accesso ai dati memorizzati dei clienti. Può inoltre modificare il menù del locale aggiungendo portate e variando gli ingredienti in esse presenti, al fine, per esempio, di aggiungere una specialità del giorno.

1.1 Possibili aggiunte

Delle possibili aggiunte non implementate potrebbero essere quelle di consentire al cliete di fare degli ordini da casa attraverso un sistema di consegna a domicilio (Es: Delivero) oppure quella di consenire all'organizzatore della sala di poter gestire le prenotazioni future dei clienti attraverso app. Login dei vari dipendenti.

2 Requisiti

2.1 Use Case

Dalla descrizione del nostro modello di dominio abbiamo identificato gli attori in gioco con le varie figure professionali del ristorante e i rispettivi casi d'uso con le rispettive mansioni. Di seguito viene riportato l'Use Case Diagram risultante:

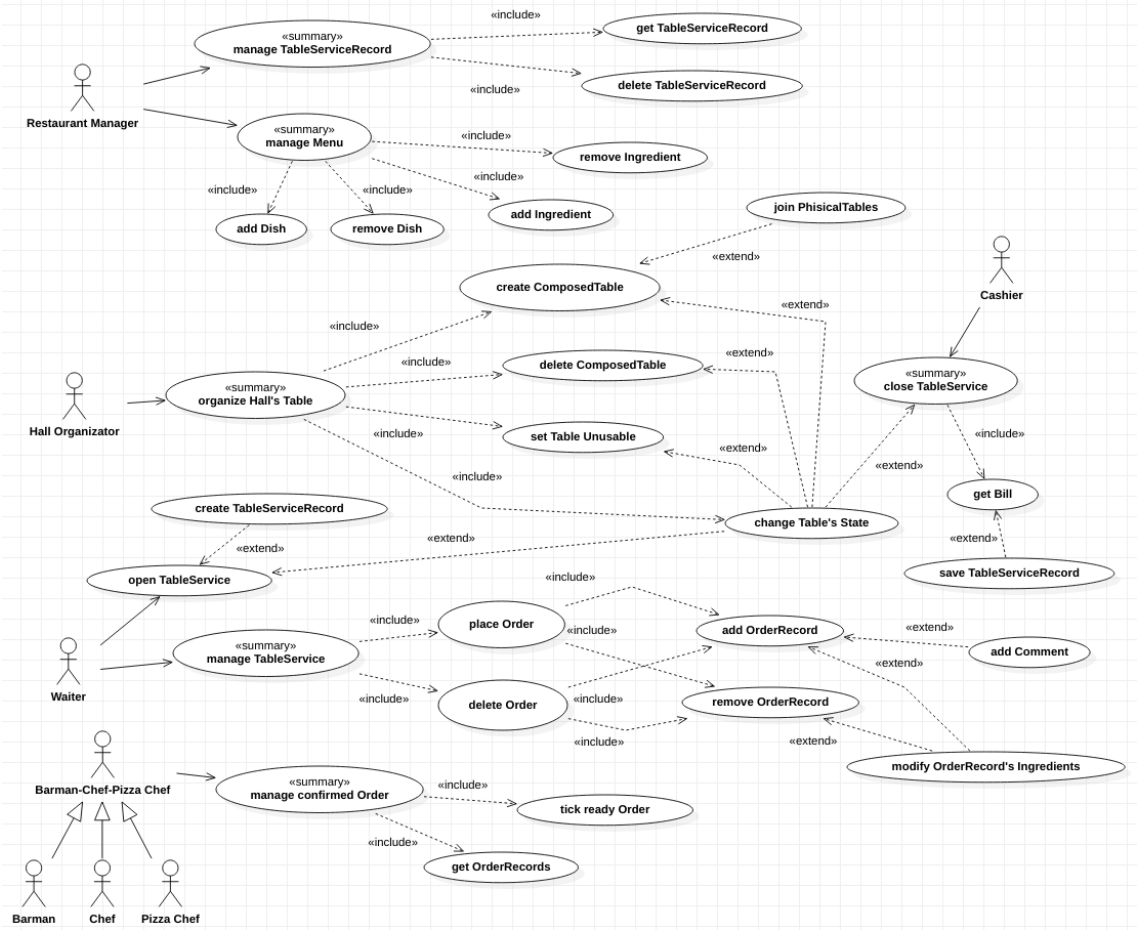

Figure 1: Use Case Diagram

2.2 Use Case Template

Riportiamo di seguito i template relativi ad alcuni dei pricipali casi d'uso individuati nel nostro progetto:

UC1	Piazza Ordine
Livello	User Goal
Descrizione	Il cameriere associa un ordine ad un tavolo attraverso l'interfaccia dell'applicativo
Attori	Cameriere
Pre-condizioni	Il cameriere deve avere un id per poter effettuare il login nell'applicativo.
Post-condizioni	Viene inviato l'ordine ai controller della cucina, del bar e della pizzeria.
Normale svolgimento	1. Il cameriere seleziona un tavolo e il sistema mostra una schermata con i piatti del menù 2. Il cameriere può selezionare il numero di coperti del tavolo 3. Il cameriere può aggiungere e rimuovere piatti all'ordine 4. Il cameriere preme AGGIUNGI ORDINE
Svolgimenti alternativi	3a. Il cameriere aggiunge un commento all'ultimo piatto selezionato 3a.1 Il cameriere compila il form apposito 3a.2 Il cameriere preme AGGIUNGI COMMENTO AL PIATTO 3b. Il cameriere aggiunge/rimuove un ingrediente al piatto 3b.1 Il cameriere preme MODIFICA INGREDIENTI AL PIATTO 3b.2 Il sistema mostra una schermata con gli ingredienti del menù 3b.3 Il cameriere aggiunge e rimuove ingredienti e preme CONFERMA 4a. Il cameriere preme RIMUOVI ORDINE 4a.1 L'ordine non viene inviato alla cucina/pizzeria/bar. 4a.2 L'ordine viene contato in negativo nel conto finale associato a quel servizio al tavolo (storno)

Figure 2: Template che descrive il caso d'uso del cameriere per piazzare un ordine ad un servizio al tavolo

UC2	Crea Tavolo Composto
Livello	User Goal
Descrizione	L'organizzatore della sala crea un tavolo composto
Attori	Organizzatore della sala
Pre-condizioni	L'organizzatore della sala deve avere un id per poter effettuare il login nell'applicativo.
Post-condizioni	Viene aggiunto il tavolo composto a quelli visibili nella vista dei tavoli
Normale svolgimento	1. L'organizzatore della sala vede la mappa dei tavoli 2. L'organizzatore della sala seleziona uno o più tavoli e preme UNISCI 3. Il sistema mostra un form per inserire l'id del nuovo tavolo creato 4. L'organizzatore della sala inserisce l'id desiderato e preme CONFERMA
Svolgimenti alternativi	4a. L'id inserito è già presente o non è valido 4a.1 Il sistema ritorna alla vista dei tavoli 4b. I tavoli fisici selezionati non sono disponibili 4b.1 Il sistema ritorna alla vista dei tavoli

Figure 3: Template che descrive il caso d'uso dell'organizzatore della sala per creare un tavolo composto

UC3	Calcola Conto
Livello	User Goal
Descrizione	Il cassiere calcola il conto di un servizio al tavolo
Attori	Cassiere
Pre-condizioni	Il cassiere deve avere un id per poter effettuare il login nell'applicativo.
Post-condizioni	Vengono salvati i dati del cliente in apposito database e viene messo lo stato del tavolo a DA PULIRE
Normale svolgimento	1. Il cassiere seleziona un tavolo dalla vista dei tavoli e preme CONTO 2. Il sistema mostra una finestra con il dettaglio del conto e il totale

Figure 4: Template che descrive il caso d'uso del cassiere per calcolare il conto di un servizio al tavolo

2.3 Mockups

Riportiamo di seguito dei possibili mockups relativi alle interfaccie grafiche della nostra app per l'interazione con i nostri attori.

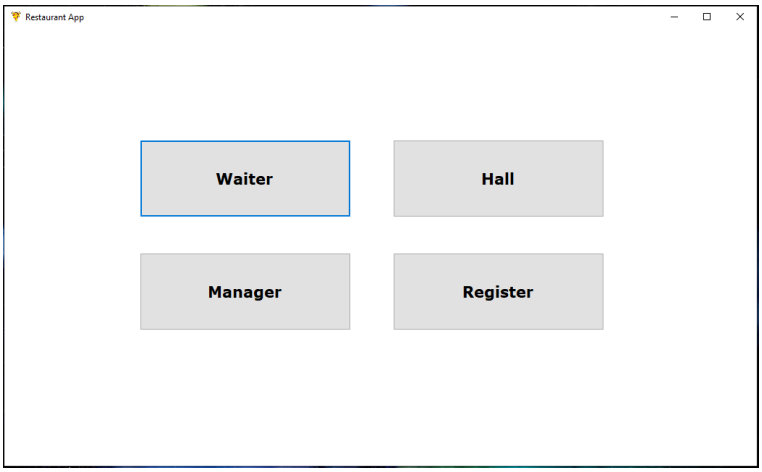

Figure 5: Mockup raffigurante un prototipo della home page dell'app

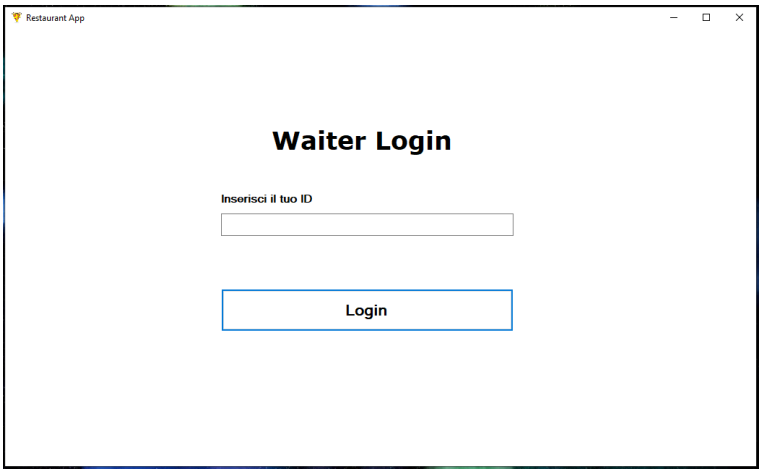

Figure 6: Mockup raffigurante un prototipo della pagina di login del cameriere

Figure 7: Mockup raffigurante un prototipo dell'interfaccia dell'organizzatore della sala

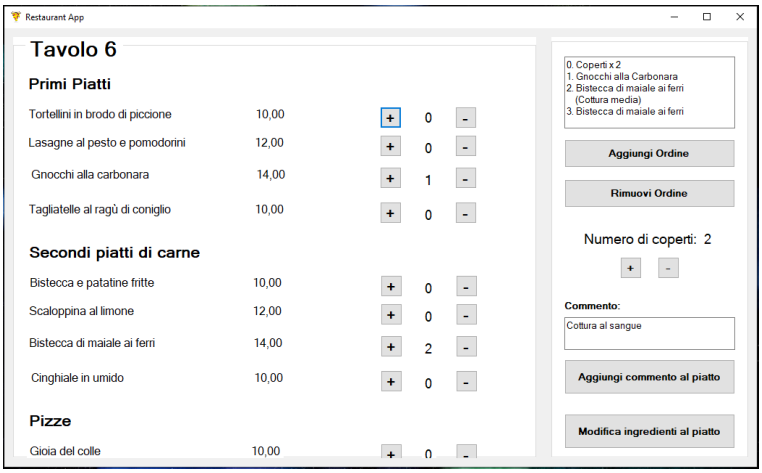

Figure 8: Mockup raffigurante un prototipo dell'interfaccia del cameriere al momento della creazione di un ordine

3 Progettazione e Implementazione

3.1 Scelte implementative e considerazioni

L'applicativo si presta per essere realizzato ad operare come un sistema distribuito, prevedendo cioè un server centrale con al suo interno dati comuni accessibili dai vari terminali attraverso un protocollo di comunicazione. Per le competenze richieste dall'elaborato, lo schema da noi riportato e implementato ha il solo scopo di illustrare le interazioni principali fra le varie parti, non è quindi atto ad una realizzazione concreta.

3.2 Class Diagram

Qui di seguito riportiamo la realizzazione del Class Diagram che descrive la nostra logica di dominio in prospettiva di implementazione:

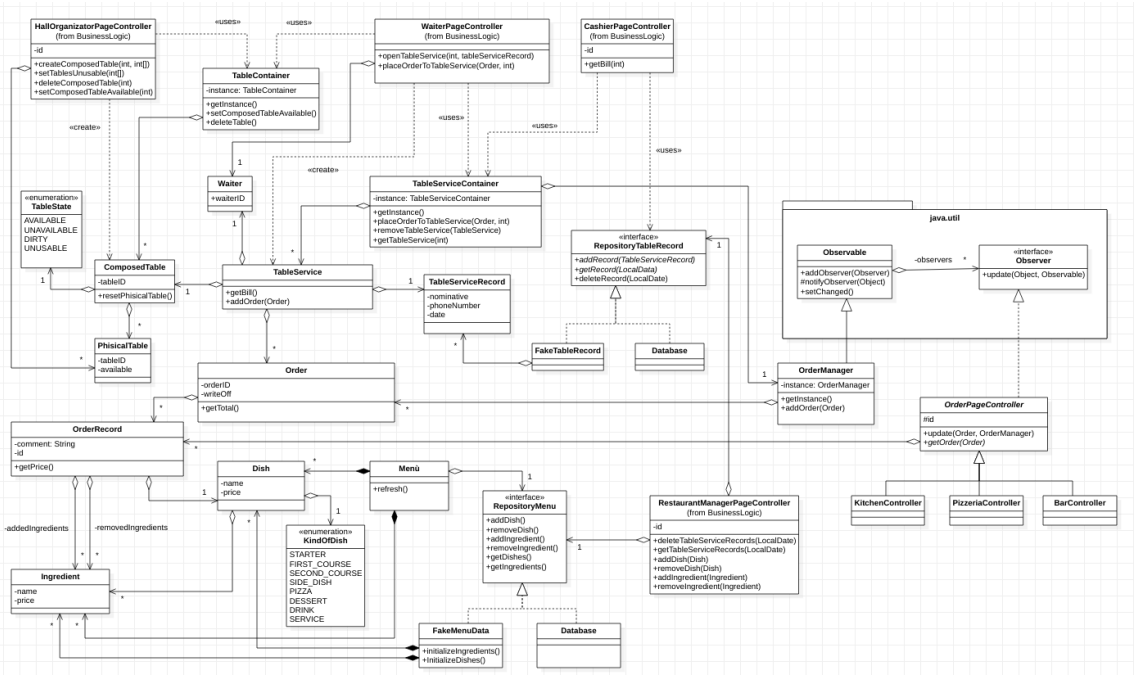

Figure 9: Class Diagram

3.3 Classi ed Interfacce

Per l'implementazione del nostro applicativo abbiamo sia definito nuove classi ed interfacce specifiche, sia utilizzato alcune di quelle già presenti nelle librerie standard di Java.
Le classi Controller sotto riportate vengono create al momento del login di un client nella relativa interfaccia grafica, sono una per ciascun attore e vengono utilizzate nei seguenti modi:

3.3.1 WaiterPageController

Tramite questo controller, il cameriere può inserire nuovi ordini relativi ad un determinato tavolo utilizzando il metodo *placeOrderToTableService()*, oppure creare un nuovo TableService con il metodo *openTableService()* indicando il tavolo interessato.

```
public class WaiterPageController {
    private Waiter waiter;

    public WaiterPageController(String ID) {
        this.waiter = new Waiter(ID);
    }

    public boolean openTableService(int idTable, TableServiceRecord tsr) {
        try {
            ComposedTable ct = TableContainer.getInstance().getTable(idTable);
            if (ct.getTableState() == TableState.AVAILABLE) {
                ct.setTableState(TableState.UNAVAILABLE);
                TableService tableService = new TableService(this.waiter, ct, tsr);
                TableServiceContainer.getInstance().addTableService(tableService);
                return true;
            }
            return false;
        } catch (IndexOutOfBoundsException e) {
            return false;
        }
    }

    public boolean placeOrderToTableService(Order order, int id) {
        return TableServiceContainer.getInstance().placeOrderToTableService(order, id);
    }
```

Figure 10: Porzione di codice della classe WaiterPageController

3.3.2 HallOraganizatorPageController

È un controller che permette di organizzare i tavoli presenti nella sala. Al suo interno contiene la lista di tavoli fisici. È possibile creare nuovi tavoli composti attraverso il metodo *createComposedTable()* specificando gli id dei tavoli fisici da utilizzare. Il tavolo composto creato viene direttamente inserito all'interno del "TableContainer". *setTablesUnusable()* ci permette invece di impostare lo stato dei tavoli, specificati con id, nello stato "Unusable".

```java
public class HallOrganizatorPageController {

    private ArrayList<PhisicalTable> tables;
    private String id;

    public HallOrganizatorPageController(ArrayList<PhisicalTable> tables, String id){
        this.id=id;
        this.tables = tables;
    }

    public boolean createComposedTable(int id, int[] IDs){
        for(ComposedTable ct: TableContainer.getInstance().getTables()) {
            if(ct.getTableID()==id)
                return false;
        }
        ComposedTable c = new ComposedTable(id);
        ArrayList<PhisicalTable> tmp = new ArrayList<PhisicalTable>();

        for (int i = 0; i < IDs.length; i++) {
            for (PhisicalTable t : tables) {
                if (t.getTableID() == IDs[i] && t.isAvailable())
                    tmp.add(t);
                else if (t.getTableID() == IDs[i] && !t.isAvailable())
                    return false;
            }
        }
        for (PhisicalTable t : tmp) {
            c.addTable(t);
        }
        TableContainer.getInstance().addTable(c);
        return true;
    }
```

Figure 11: Porzione di codice della classe HallOrganizatorPageController

```java
public boolean setTablesUnusable(int[] IDs) {
    ArrayList<PhisicalTable> tmp = new ArrayList<PhisicalTable>();

    for (int i = 0; i < IDs.length; i++) {
        for (PhisicalTable t : tables) {
            if (t.getTableID() == IDs[i] && t.isAvailable())
                tmp.add(t);
        }
    }
    if (tmp.size() != IDs.length)
        return false;

    for (PhisicalTable t : tmp) {
        ComposedTable c = new ComposedTable(TableState.UNUSABLE, t.getTableID());
        c.addTable(t);
        TableContainer.getInstance().addTable(c);
    }
    return true;
}
```

Figure 12: Porzione di codice della classe HallOrganizatorPageController

3.3.3 CashierPageController

Questo controller viene utilizzato dall'interfaccia del cassiere per calcolare il conto relativo ad un servizio al tavolo utilizzando *getBill()*.

```java
public class CashierPageController {
    private RepositoryTableRecord repository;
    private String id;

    public CashierPageController(RepositoryTableRecord repository, String id) {
        this.repository = repository;
        this.id=id;
    }

    public double getBill(int id) {
        try {
            TableService ts = TableServiceContainer.getInstance().getTableService(id);
            TableServiceContainer.getInstance().removeTableService(ts);
            repository.addRecord(ts.getTableServiceRecord());
            ts.getComposedTable().setTableState(TableState.DIRTY);
            return ts.getBill();
        } catch(IndexOutOfBoundsException e) {
            return -1;
        }
    }

}
```

Figure 13: Porzione di codice della classe CashierPageController

3.3.4 RestaurantManagerPageController

Viene utilizzato per la gestione del "Menù" e dei "TableServiceRecord" da parte del manager del ristorante. Al suo interno sono presenti metodi per la gestione del menù, ad esempio *addDish()* e *removeDish()* per aggiungere e rimuovere un piatto rispettivamente. Inoltre il manager del ristorante ha la possibilità di cancellare dei "TableServiceRecord" relativi ad una data con *deleteTableServiceRecords()* o di visionarli con *getTableServiceRecords()*.

```java
public boolean deleteTableServiceRecords(LocalDate date) {
    return repositoryTableRecord.deleteRecords(date);
}

public ArrayList<TableServiceRecord> getTableServiceRecords(LocalDate date) {
    return repositoryTableRecord.getRecords(date);
}

public void addDish(Dish dish) {
    repositoryMenu.addDish(dish);
}

public void removeDish(Dish dish) {
    repositoryMenu.removeDish(dish);
}
```

Figure 14: Porzione di codice della classe RestaurantManagerPageController

3.3.5 OrderPageController, BarController, KitchenController e PizzeriaController

"OrderPageController" è una classe astratta che abbiamo creato per definire i metodi necessari ai controller specifici della cucina: "BarController", "KitchenController" e "PizzeriaController". Questi ultimi, infatti, la implementano e definiscono al loro interno il metodo astratto *getOrder()* in modo tale da prendere soltanto gli "OrderRecord" a cui sono interessati. Inoltre "OrderPageController" implementa l'interfaccia "Observer" del package java.util così da poter essere avvisata dall' "OrderManager" ("Observable") quando vengono piazzati nuovi ordini e tramite il metodo *update()* aggiornare automaticamente la lista dei suoi "OrderRecord" (Vedi Sezione Observer 3.4.1).

```
public abstract class OrderPageController implements Observer{

    protected ArrayList<OrderRecord> orderRecords;
    protected String id;

    @Override
    public void update(Observable ordermanager, Object order) {
        Order newOrder = (Order)order;
        getOrder(newOrder);
    }

    public abstract void getOrder(Order order);
```

Figure 15: Porzione di codice della classe OrderPageController

```
public class PizzeriaController extends OrderPageController{

    public PizzeriaController(Observable obs, String id) {
        this.id=id;
        obs.addObserver(this);
        orderRecords = new ArrayList<OrderRecord>();
    }

    @Override
    public void getOrder(Order order) {
        for(OrderRecord ord : order.getRecords()) {
            if(ord.getDish().getKindOfDish() == KindOfDish.PIZZA) {
                orderRecords.add(ord);
            }
        }
    }
}
```

Figure 16: Porzione di codice della classe PizzeriaController

Di seguito riportiamo la descrizione di alcune delle classi principali appartenenti alla parte di Domain Model del nostro progetto:

3.3.6 PhisicalTable e ComposedTable

I "PhisicalTable" rappresentano i tavoli presenti all'interno del locale, vengono identificati tramite un id e *setAvailable()* permette di renderli utilizzabili o non utilizzabili.

```
public class PhisicalTable {
    private int tableID;
    private boolean available = true;

    public PhisicalTable(int tableID) {
        this.tableID = tableID;
    }

    public void setAvailable(boolean available) {
        this.available = available;
    }
```

Figure 17: Porzione di codice della classe PhisicalTable

"ComposedTable" è un tavolo composto da un insieme di "PhisicalTable" e viene identificato, anche lui, con un id. Il metodo *addTable()* permette di aggiungere un "PhisicalTable" al suo interno. Inoltre, è possibile assegnargli un "TableState" per contrassegnare lo stato in cui si trova.

```
public class ComposedTable {
    private int tableID;
    private ArrayList<PhisicalTable> tables = new ArrayList<>();
    private TableState tableState;

    public ComposedTable(TableState tableState, int tableID) {
        this.tableState = tableState;
        this.tableID = tableID;
    }

    public ComposedTable(int tableID) {
        this.tableState = TableState.AVAILABLE;
        this.tableID = tableID;
    }

    public void addTable(PhisicalTable pt) {
        pt.setAvailable(false);
        tables.add(pt);
    }
```

Figure 18: Porzione di codice della classe Composed

3.3.7 TableContainer

Contiene tutti i "ComposedTable" creati dall'organizzatore della sala in una lista al suo interno. Permette di aggiungere, eliminare o cambiare lo stato ai tavoli composti con i metodi *addTable()*, *deleteTable(int id)* e *setComposedTableAvailable()*.

```
public class TableContainer {

    private static TableContainer instance=null;
    private ArrayList<ComposedTable> tables;

    private TableContainer() {
        tables = new ArrayList<ComposedTable>();
    }

    public static TableContainer getInstance() {
        if(instance==null)
            instance= new TableContainer();
        return instance;
    }

    public void addTable(ComposedTable table) {
        tables.add(table);
    }
```

Figure 19: Porzione di codice della classe TableContainer

```
public boolean deleteTable(int id) {
    for(ComposedTable ct : tables) {
        if(ct.getTableID() == id) {
            ct.resetPhisicalTable();
            tables.remove(ct);
            return true;
        }
    }
    return false;
}

public boolean setComposedTableAvailable(int id) {
    for(ComposedTable ct : tables) {
        if(ct.getTableID() == id) {
            ct.setTableState(TableState.AVAILABLE);
            return true;
        }
    }
    return false;
}
```

Figure 20: Porzione di codice della classe TableContainer

3.3.8 TableService

"TableService" è una classe che raggruppa al suo interno tutte le informazioni inerenti al servizio di un tavolo. Al suo interno troviamo infatti: lista degli ordini piazzati, "TableServiceRecord", il "ComposedTable" a cui è associato e il cameriere che lo ha creato e sta effettuando il servizio.

```
public class TableService {
    private ComposedTable composedTable;
    private TableServiceRecord tableServiceRecord;
    private Waiter waiter;
    private ArrayList<Order> orders;
```

Figure 21: Porzione di codice della classe TableService

3.3.9 TableServiceContainer

Contiene tutti i "TableService" creati dai camerieri. Permette di gestire i "TableService" contenuti al suo interno con diversi metodi come, ad esempio, *addTableService()* per aggiungere un TableService o *placeOrderToTableService()* per aggiungere un ordine al TableService del tavolo id.

```
public void addTableService(TableService tableService){
    tableServices.add(tableService);
}

public TableService getTableService(int idTable) throws IndexOutOfBoundsException {
    for (TableService t : tableServices) {
        if (t.getComposedTable().getTableID() == idTable)
            return t;
    }
    throw new IndexOutOfBoundsException();
}

public boolean placeOrderToTableService(Order order, int id) {
    try {
        TableService ts = getTableService(id);
        ts.addOrder(order);
        if(!order.isWriteOff())
            OrderManager.getInstance().addOrder(ts.getOrders().get(ts.getOrders().size()-1));
        return true;
    } catch (IndexOutOfBoundsException e) {
        return false;
    }

}

public boolean removeTableService(TableService ts) {
    return tableServices.remove(ts);
}
```

Figure 22: Porzione di codice della classe TableServiceContainer

3.3.10 OrderRecord

Questa classe viene utilizzata per inserire i piatti del menù nelle comande e, inoltre, rende possibile l'aggiunta di un commento o la rimozione/aggiunta di ingredienti al piatto. All'interno si trovano infatti due liste di "Ingredients", una contenente gli ingredienti aggiunti "ArrayList¡Ingredient¿ addedIngredients" e l'altra gli ingredienti rimossi "ArrayList¡Ingredient¿ removedIngredients". Attraverso il metodo *getPrice()* calcola il prezzo del piatto corrispondente sommando e sottraendo il prezzo degli ingredienti rispettivamente aggiunti e rimossi. Ogni "OrderRecord" è identificato da un id, lo stesso dell' "Order" che lo contiene.

```
public double getPrice() {
    double total=0;
    total+=dish.getPrice();
    for(Ingredient ingredient:addedIngredients) {
        total+=ingredient.getPrice();
    }
    for (Ingredient ingredient : removedIngredients) {
        total-=ingredient.getPrice();
    }
    return total;
}

public boolean addIngredient(Ingredient ingredient) {
    if(dish.getKindOfDish()==KindOfDish.DRINK || dish.getKindOfDish()==KindOfDish.SERVICE)
        return false;
    if(!dish.getIngredients().contains(ingredient) && !addedIngredients.contains(ingredient)) {
        addedIngredients.add(ingredient);
        return true;
    }
    return false;
}

public boolean removeIngredient(Ingredient ingredient) {
    if(dish.getKindOfDish()==KindOfDish.DRINK || dish.getKindOfDish()==KindOfDish.SERVICE)
        return false;
    if((dish.getIngredients().contains(ingredient) || addedIngredients.contains(ingredient)) &&
        !removedIngredients.contains(ingredient)) {
        removedIngredients.add(ingredient);
        return true;
    }
    return false;
}
```

Figure 23: Porzione di codice della classe OrderRecord

3.3.11 OrderManager

Si occupa della gestione degli ordini. Al suo interno è presente una lista di ordini e con il metodo *addOrder()* se ne possono aggiungere di nuovi. "OrderManager" estende la classe astratta "Observable" del package java.util, così che ogni qualvolta venga aggiunto un nuovo "Order" alla lista venga richiamato *notifyObservers()* che avvisa gli "Observer" ("OrderPageController") del nuovo ordine aggiunto (Vedi Sezione Observer 3.4.1).

```java
public class OrderManager extends Observable {

    private ArrayList<Order> orders;
    private static OrderManager instance=null;

    private OrderManager() {
        orders = new ArrayList<Order> ();
    }

    public static OrderManager getInstance() {
        if(instance==null) {
            instance= new OrderManager();
        }
        return instance;
    }

    public void addOrder(Order order) {
        orders.add(order);
        setChanged();
        notifyObservers(order);
    }

}
```

Figure 24: Porzione di codice della classe OrderManager

3.3.12 RepositoryMenu e FakeMenuData

"RepositoryMenu" è un'interfaccia che espone tutti i metodi necessari per la gestione del "Menù" come, ad esempio, *addDish()* e *removeDish()*. Con *getDishes()* e *getIngredients()* è possibile ottenere le liste contenenti tutti i piatti e tutti gli ingredienti del menù. Grazie a questa interfaccia è possibile utilizzare diversi tipi di oggetti per lo storage di dati senza dover cambiare il codice che utilizza il menù, introducendo così un ulteriore livello di astrazione.

```java
public interface RepositoryMenu {

    public void addDish(Dish dish);
    public void removeDish(Dish dish);
    public void addIngredient(Ingredient ingredient);
    public void removeIngredient(Ingredient ingredient);
    public ArrayList<Dish> getDishes();
    public ArrayList<Ingredient> getIngredients();

}
```

Figure 25: Porzione di codice della classe RepositoryMenu

"FakeMenuData" implementa "RepositoryMenu" ed è stato creato per avere un oggetto di storage fittizio che ci permettesse di eseguire i test senza dover implementare sul momento piatti e ingredienti.

```java
public class FakeMenuData implements RepositoryMenu {

    private ArrayList<Dish> dishes;
    private HashMap<String, Ingredient> ingredients;

    public FakeMenuData() {
        dishes = new ArrayList<>();
        ingredients = new HashMap<String, Ingredient>();
        initializeIngredients();
        initializeDishes();
    }
```

Figure 26: Porzione di codice della classe FakeMenuData

```java
private void initializeIngredients() {
    Ingredient i= new Ingredient("Bufala", 2);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Pomodoro", 0.5);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Salsiccia", 2);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Stracchino", 2);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Friarielli", 1.5);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Banana", 0.5);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Gelato al Cioccolato", 1.5);
    ingredients.put(i.getName(), i);
    i= new Ingredient("Prosciutto Crudo", 2.5);
    ingredients.put(i.getName(), i);
}

private void initializeDishes() {
    Dish d= new Dish("Coccoli", 7.5, KindOfDish.STARTER);
    d.addIngredient(ingredients.get("Stracchino"));
    d.addIngredient(ingredients.get("Prosciutto Crudo"));
    dishes.add(d);

    d= new Dish("Pasta alla Norma", 11, KindOfDish.FIRST_COURSE);
    d.addIngredient(ingredients.get("Melanzane"));
    d.addIngredient(ingredients.get("Bufala"));
    d.addIngredient(ingredients.get("Pomodoro"));
    dishes.add(d);
```

Figure 27: Porzione di codice della classe FakeMenuData

3.4 Design Patterns

All'interno del progetto abbiamo avuto la necessità di introdurre alcuni design patterns noti per favorire la gestione di alcune dipendenze tra classi in modo agile ed elegante. I patterns utilizzati sono:

1. Observer
2. Singleton
3. MVC

3.4.1 Observer

Il pattern comportamentale Observer è utilizzato per instaurare una relazione uno a molti fra oggetti cosicchè quando un oggetto cambia stato, tutte le sue dipendenze vengano informate e aggiornate automaticamente.
Nella nostra logica di dominio abbiamo avuto la necessità di introdurre questo pattern per notificare alla cucina l'invio da parte del cameriere di una nuova comanda. Nonostante siano state deprecate, ci siamo serviti delle classi "Observable" e "Observer" e dei loro rispettivi metodi *notifyObservers()* e *update()* forniti dalla libreria java.util. Abbiamo inoltre deciso di implementare il pattern in modalità push, la quale prevede che l'oggetto osservato notifichi gli osservatori inviando direttamente il cambiamento al momento della sua variazione.
Come è possibile infatti apprezzare dal Class Diagram di Figura 9, la classe "OrderManager" svolge il ruolo di oggetto osservato ("Observable") che notifica l'arrivo di una nuova comanda inviandola direttamente alla cucina, alla pizzeria e al bar che svolgono il ruolo di osservatori ("Observer"). Questo avviene attraverso il metodo *addOrder()* che preso come parametro il nuovo ordine piazzato dal cameriere lo invia agli "Observers" attraverso il metodo *notifyObservers()*. Questi ultimi aggiorneranno poi il loro stato attraverso il metodo *update()* (Vedi Sequence Diagram in Figura 36).

3.4.2 Singleton

Il pattern Creazionale Singleton è utilizzato per avere un'unica istanza di una determinata classe.
Nel nostro progetto abbiamo infatti introdotto questo pattern per garantire l'unicità delle istanze delle classi "TableContainer", "TableServiceContainer" e "OrderManager", le quali interagiscono con più attori e classi e si ha quindi la necessità che siano sempre le stesse. Come è possibile osservare nel Sequence Diagram di Figura 36, gli attori non possiedono i riferimenti all'unica istanza di queste classi, ma li ottengono al momento che necessitano di utilizzarle attraverso il rispettivo metodo *getInstance()*.

3.4.3 MVC

Il pattern Architetturale Model-View-Controller è utilizzato quando si ha la necessità di accedere e modificare dei dati attraverso interazioni differenti con i client. Per far questo si possono dunque individuare le tre componenti principali:

- Model: È la parte che definisce il modello dei dati e le operazioni che possono essere eseguite su queste presentandole alla View e al Controller. Il Model può inoltre notificare ai vari componenti della View eventuali aggiornamenti in seguito a richieste del Controller al fine di presentare ai Client dati sempre aggiornati.
- View: Corrisponde alle varie interfacce dell'applicazione con cui gli utenti si possono trovare ad interagire per eseguire delle azioni sul sistema.
- Controller: Svolge il ruolo da mediatore fra il View e il Model, trasformando le interazioni dell'utente sul primo in azioni sul secondo. Per farlo sfrutta inoltre un meccanismo che prende il nome di business logic. Questo consiste nel mettere in vita determinati oggetti del Model solo nel momento del loro effettivo bisogno, ossia quando devono interagire in risposta alle richieste del client.

Nel nostro applicativo abbiamo sentito la necessità di introdurre questo pattern in quanto i nostri attori interagiscono fra di loro e con il sistema al fine di scambiare dati e informazioni attraverso una molteplicità di viste differenti. Riportiamo quindi di seguito l'architettura del nostro applicativo suddivisa nelle tre componenti sopra descritte:

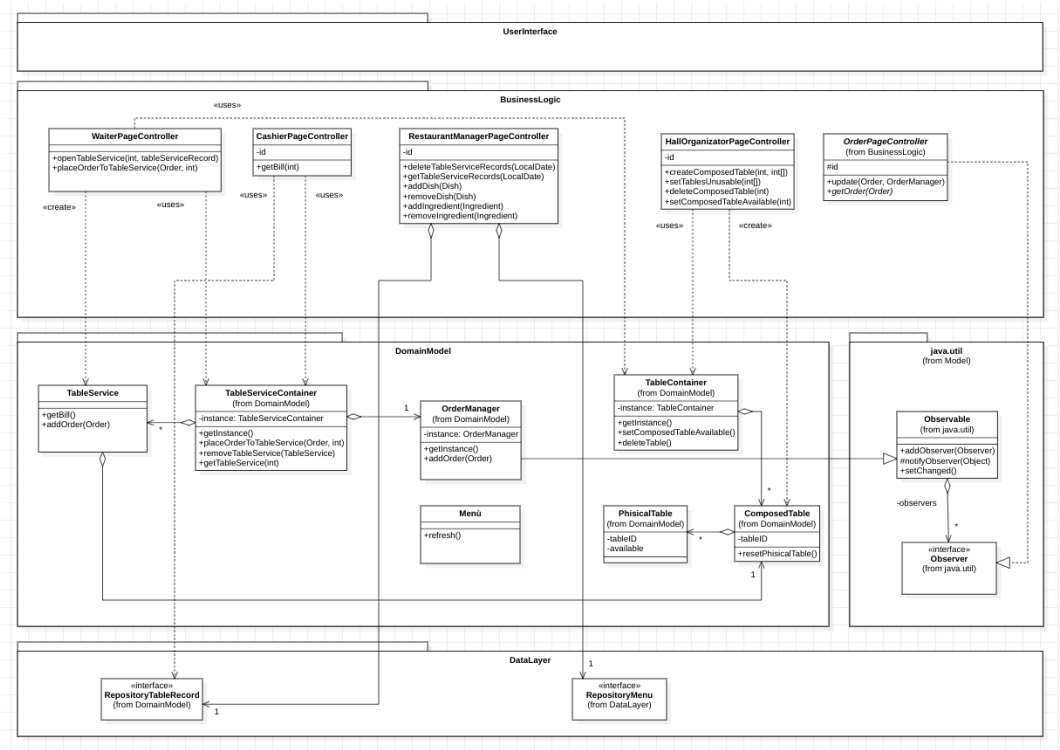

Figure 28: Raffigurazione dell'architettura dell'applicativo secondo lo schema Model-View-Controller

3.5 Disposizione delle classi nei package

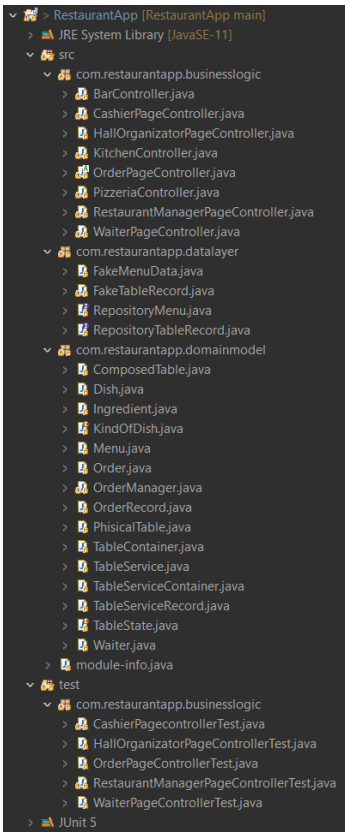

Figure 29: Raffigurazione della disposizione delle classi del progetto nei package

4 UnitTest

Per testare la corretta interazione e collabborazione fra le parti abbiamo realizzato i seguenti test cases per alcune delle classi principali dell'applicativo:

Nel progetto è stato utilizzato il framework JUnit 5.0.

4.1 WaiterPageControllerTest

Nella seguente classe di test, è stata testato il controller del cameriere. In particolare dopo aver istanziato alcuni tavoli composti, vengono testate le funzioni *openTableService()* e *placeOrderToTableService()*. Ciò che ci ha spinto a scegliere questo test, è stato il fatto questi metodi sono tra i più essenziali tra quelli che stanno dietro la logica dell'applicativo.

```
@Test
@DisplayName("Ensures that table service opening takes place correctly")
void testOpenTableService() {
    assertTrue(WPC.openTableService(40, new TableServiceRecord("Francesco", "7013028")), "Table service created");
    assertFalse(WPC.openTableService(40, new TableServiceRecord("Francesco", "7165028")) ,"Table not available");
    assertFalse(WPC.openTableService(16, new TableServiceRecord("Gianni", "7015778")) ,"Table not found");
    assertFalse(WPC.openTableService(70, new TableServiceRecord("Francesco", "7015028")) ,"Table unusable");
}

@Test
@DisplayName("Placing an order to table service should work")
void testPlaceOrderToTableService() {

     Order order = new Order();

     Dish dish = menu.getDishes().get(0);
     Ingredient ingredient = menu.getIngredients().get(1);

     OrderRecord orderRecord = new OrderRecord(dish);
     orderRecord.setComment("impasto integrale");

     assertTrue(orderRecord.addIngredient(ingredient),"Ingredient added");
     assertFalse(orderRecord.addIngredient(dish.getIngredients().get(0)),"Ingredient already inside");
     assertTrue(orderRecord.removeIngredient(ingredient),"Ingredient removed");
     assertFalse(orderRecord.removeIngredient(ingredient),"Ingredient not inside");

     order.addOrderRecord(orderRecord);
     WPC.openTableService(60, new TableServiceRecord("Gianni", "7019928"));

     assertTrue(WPC.placeOrderToTableService(order, 60),"Order placed");
     assertFalse(WPC.placeOrderToTableService(order, 20),"try to place an Order to unexistent TableService");
}
```

Figure 30: Porzione di codice del test case del cameriere

4.2 RestaurantManagerPageControllerTest

Il manager del ristorante può gestire sia i dati relativi ai clienti che il menù. Nella classe di test sottostante ci siamo concentrati sui metodi che visualizzano e cancellano i dati dei clienti. Gli altri metodi relativi al menù non li abbiamo testati per due ragioni principali: la prima è che non è presente un database sottostante implementato, la seconda è che sono analoghi ai precedenti.

```
@Test
@DisplayName("Get Table Service should work")
void testGetTableServiceRecords() {

    assertTrue(records.equals(RMPC.getTableServiceRecords(LocalDate.now())), "get table service record");
}

@Test
@DisplayName("Delete Table Service should work")
void testDeleteTableServiceRecords() {
    ArrayList<TableServiceRecord> tmp = new ArrayList<>();
    assertFalse(tmp.equals(RMPC.getTableServiceRecords(LocalDate.now())), "check if the record list is not empty");
    assertTrue(RMPC.deleteTableServiceRecords(LocalDate.now()), "delete records");
    assertTrue(tmp.equals(RMPC.getTableServiceRecords(LocalDate.now())), "check if the record list is empty");
    assertFalse(RMPC.deleteTableServiceRecords(LocalDate.now()), "no data to delete");
}
```

Figure 31: Porzione di codice del test case del manager del ristorante

4.3 CashierPageControllerTest

Il controller del cassiere può calcolare il conto associato ad un servizio al tavolo. Nella classe di test sottostante è stato testato il corretto funzionamento del metodo *getBill()*. In particolare vengono istanziati alcuni ordini associati a due diversi servizi al tavolo. Sono presenti piatti con ingredienti aggiunti e rimossi e ordini di storno. Quello che il cassiere deve controllare è che vengano calcolati i conti in modo corretto e che lo stato dei tavoli venga messo a "DA PULIRE".

```
@Test
@DisplayName("getBill should return the exact price")
void testGetBill() {
    records = new ArrayList<>();
    records.add(new TableServiceRecord("Francesco", "123456789"));
    records.add(new TableServiceRecord("Gianni", "987654321"));

    assertEquals(44, CPC.getBill(40), "Bill with added and removed ingredients");
    assertEquals(25.5, CPC.getBill(60), "Bill with write off");
    assertEquals(TableState.DIRTY, TableContainer.getInstance().getTable(40).getTableState(), "Assert first DIRTY");
    assertEquals(TableState.DIRTY, TableContainer.getInstance().getTable(60).getTableState(), "Assert second DIRTY");

    assertEquals(2, repository.getRecords(LocalDate.now()).size(), "check saved records");
}
```

Figure 32: Porzione di codice del test case del cassiere

4.4 HallOrganizatorPageController

Nella seguente classe di test, abbiamo testato tutti i metodi principali del controller dell'organizzatore della sala. In particolare dopo aver istanziato alcuni tavoli fisici, viene verificato il corretto funzionamento dei metodi *createComposedTable(), deleteComposedTable(), setComposedTableAvailable()* e *setTablesUnusable()*. La ragione che ci ha spinto ad effettuare questi test è, ancora una volta, il fatto che questi metodi sono tra i più essenziali tra quelli che stanno dietro la logica dell'applicativo.

```
@Test
@DisplayName("Set Composed Table Unusable should work")
void testSetTableUnusable() {
    assertTrue(HOPC.setTablesUnusable(new int[] { 1, 3, 5 }), "set table unusable");
    HOPC.createComposedTable(12, new int[] { 9 });
    assertFalse(HOPC.setTablesUnusable(new int[] { 1, 9 }), "set table unusable with unusable and unavailable");
    assertFalse(HOPC.setTablesUnusable(new int[] { 13 }), "try to set table unusable with unexistent table");
}

@Test
@DisplayName("Create Composed Table should work")
void testCreateComposedTable() {
    assertTrue(HOPC.createComposedTable(40, new int[] { 0, 2, 4 }), "create composed table");
    assertFalse(HOPC.createComposedTable(50, new int[] { 1, 3, 6 }), "create composed table with unusable");
    assertFalse(HOPC.createComposedTable(60, new int[] { 2, 7, 8 }), "create composed table with unavailable");
    assertFalse(HOPC.createComposedTable(40, new int[] { 6, 7}), "create composed with existent ID");
}

@Test
@DisplayName("Delete Composed Table should work")
void testDeleteComposedTable() {
    HOPC.createComposedTable(80, new int[] { 6, 7});
    assertTrue(HOPC.deleteComposedTable(80), "delete composed table");
    assertTrue(tables.get(6).isAvailable(), "reset phisical table works #1");
    assertTrue(tables.get(7).isAvailable(), "reset phisical table works #2");
    assertFalse(HOPC.deleteComposedTable(70), "delete unexistent composed table");
}
```

Figure 33: Porzione di codice del test case dell'organizzatore della sala

```
@Test
@DisplayName("Set Composed Table Available should work")
void testSetComposedTableAvailable() {
    ComposedTable ct = new ComposedTable(TableState.DIRTY, 45);
    ct.addTable(tables.get(6));
    ct.addTable(tables.get(7));
    ct.addTable(tables.get(8));
    TableContainer.getInstance().addTable(ct);

    assertTrue(HOPC.setComposedTableAvailable(45),"set composed table available");
    assertEquals(TableState.AVAILABLE, ct.getTableState(),"check the effectice state of composed table");
    assertFalse(HOPC.setComposedTableAvailable(50),"set unexistent composed table available");
}
```

Figure 34: Porzione di codice del test case dell'organizzatore della sala

4.5 OrderPageControllerTest

Nella seguente classe di test viene testato il corretto funzionamento del pattern Observer. In particolare vengono creati due ordini, di cui uno di storno, che vengono piazzati da un cameriere su un servizio al tavolo. Il test va a controllare che i controller della cucina, della pizzeria e del ristorante siano stati aggiornati correttamente. Successivamente viene piazzato un ulteriore ordine per controllare che i controller vengano aggiornati nel modo giusto.

```
@Test
@DisplayName("Observer should work")
void test() {
    ArrayList<OrderRecord> kitchen = new ArrayList<>();
    for (int i = 0; i < 4; i++)
        kitchen.add(new OrderRecord(menu.getDishes().get(i)));

    ArrayList<OrderRecord> pizzeria = new ArrayList<>();
    pizzeria.add(new OrderRecord(menu.getDishes().get(4)));

    ArrayList<OrderRecord> bar = new ArrayList<>();
    for (int i = 5; i < 8; i++)
        bar.add(new OrderRecord(menu.getDishes().get(i)));

    assertEquals(4, KC.getRecords().size(), "kitchen records is of right size");
    KC.getRecords().retainAll(kitchen);
    assertTrue(KC.getRecords().isEmpty(), "check kitchen records");

    assertEquals(1, PC.getRecords().size(), "pizzeria records is of right size");
    PC.getRecords().retainAll(pizzeria);
    assertTrue(PC.getRecords().isEmpty(), "check pizzeria records");

    assertEquals(3, BC.getRecords().size(), "bar records is of right size");
    BC.getRecords().retainAll(bar);
    assertTrue(BC.getRecords().isEmpty(), "check bar records");

    Order order2= new Order();
    Dish d= menu.getDishes().get(0);
    OrderRecord or2= new OrderRecord(d);
    order2.addOrderRecord(or2);
    WPC.placeOrderToTableService(order2,40);

    assertEquals(1, KC.getRecords().size(), "kitchen has received right order");
    assertTrue(PC.getRecords().isEmpty(), "pizzeria hasn't received any order");
    assertTrue(BC.getRecords().isEmpty(), "bar hasn't received any order");

}
```

Figure 35: Porzione di codice del test case della cucina

5 Sequence Diagram

Di seguito riportiamo un possibile scenario di interazione dei nostri attori:

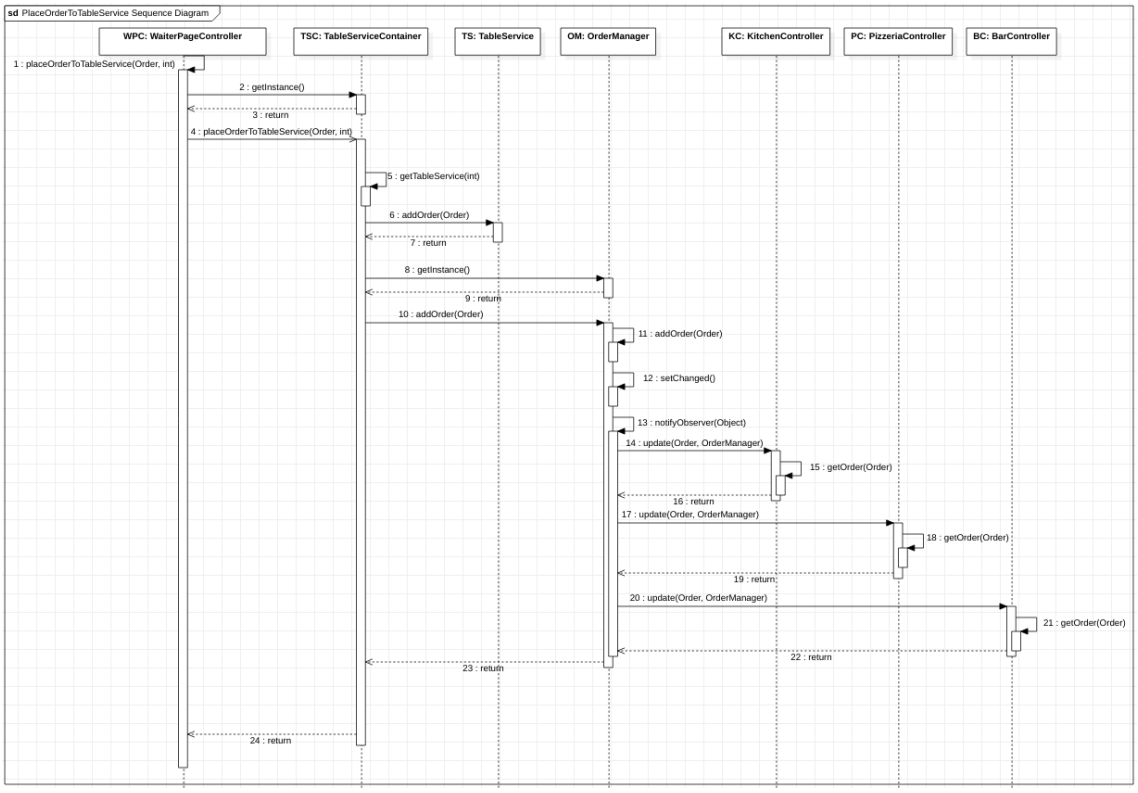

Figure 36: Sequence Diagram che documenta il flusso del controllo nel momento in cui il cameriere piazza un nuovo ordine a un tavolo