Relazione su Applicativo Java per la Gestione di un Ristorante

Pizzolato Antonio - Zheng Junliang

Maggio 2024

UNIVERSITÀ
DEGLI STUDI
FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
INGEGNERIA DEL SOFTWARE

Repository Github

Indice

Capitolo I

Introduzione

1 Descrizione dell'Applicativo

L'idea alla base del nostro progetto si fonda sulla creazione di un'applicazione innovativa e intuitiva, concepita per fornire una gestione semplificata e ottimizzata di un ristorante. L'applicazione è pensata per venire incontro alle esigenze dei proprietari dei ristoranti, offrendo loro uno strumento efficace per la gestione quotidiana del locale. Allo stesso tempo, l'applicazione prevede un'interfaccia dedicata ai clienti, permettendo loro di effettuare ordini in modo rapido e comodo.

1.1 Funzionalità Principali

1.1.1 Schermata di Login

Al primo accesso, l'utente si troverà di fronte a una schermata di login, dove sarà richiesto di inserire le proprie credenziali per autenticarsi. Il sistema di login è progettato per garantire la sicurezza dei dati e l'accesso riservato alle funzionalità dell'applicazione. Una volta effettuato l'accesso, le funzionalità disponibili varieranno in base al tipo di utente: amministratore o cliente.

1.1.2 Funzionalità per l'Amministratore

Le principali funzionalità riservate all'***amministratore*** sono state sviluppate per garantire una gestione completa e efficiente del ristorante. Tra queste troviamo:

- **Gestione dei Dipendenti (Staff):** L'amministratore ha la possibilità di aggiungere, modificare o rimuovere i dati dei dipendenti, monitorandone le attività e assegnando ruoli specifici all'interno del ristorante.
- **Gestione del Menu (Menu):** Questa funzionalità consente all'amministratore di creare, aggiornare e organizzare il menu del ristorante, inserendo descrizioni dettagliate dei piatti, prezzi, e eventuali variazioni stagionali o promozionali.
- **Visualizzazione degli Ordini (Orders):** L'amministratore può visualizzare in tempo reale gli ordini effettuati dai clienti, gestire lo stato degli ordini, e analizzare le statistiche di vendita per ottimizzare il servizio e le offerte del ristorante.

1.1.3 Funzionalità per il Cliente

Per quanto riguarda i ***clienti***, l'applicazione offre un'esperienza d'uso semplice e intuitiva:

- **Selezione degli Elementi dal Menu:** I clienti possono sfogliare il menu creato dall'amministratore, visualizzare i dettagli dei piatti, e selezionare gli elementi desiderati.
- **Effettuazione del Pagamento:** Una volta effettuata la selezione, il cliente può procedere al pagamento attraverso diversi metodi, quali contanti, carta di credito o buoni pasto, garantendo flessibilità e comodità.

1.2 Tecnologie Utilizzate

L'applicazione è stata sviluppata utilizzando ***JavaFX*** per la creazione delle interfacce grafiche, offrendo un'esperienza utente fluida e reattiva. Per la gestione dei dati, è stato integrato un database ***MySQL***, dove vengono memorizzati i dati relativi ai dipendenti, al menu, agli ordini e agli utenti registrati. Questo assicura che tutte le informazioni siano conservate in modo sicuro e accessibile, permettendo una gestione efficiente e centralizzata del ristorante.

Capitolo II

Requisiti

2 Use Case

Dalla descrizione del nostro modello di dominio, abbiamo identificato gli use case che possono essere incontrati dagli utenti del sistema di gestione del ristorante, sia utenti normali che amministratori. Di seguito viene riportato l'Use Case Diagram risultante:

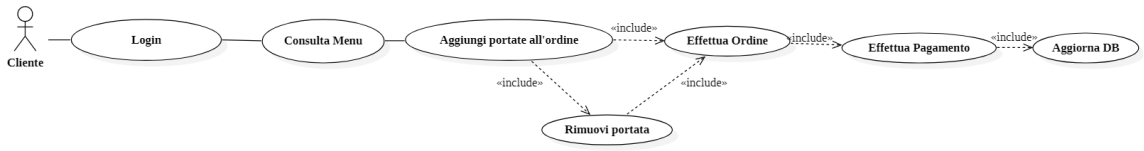

Figura 1: Use Case Diagram Client

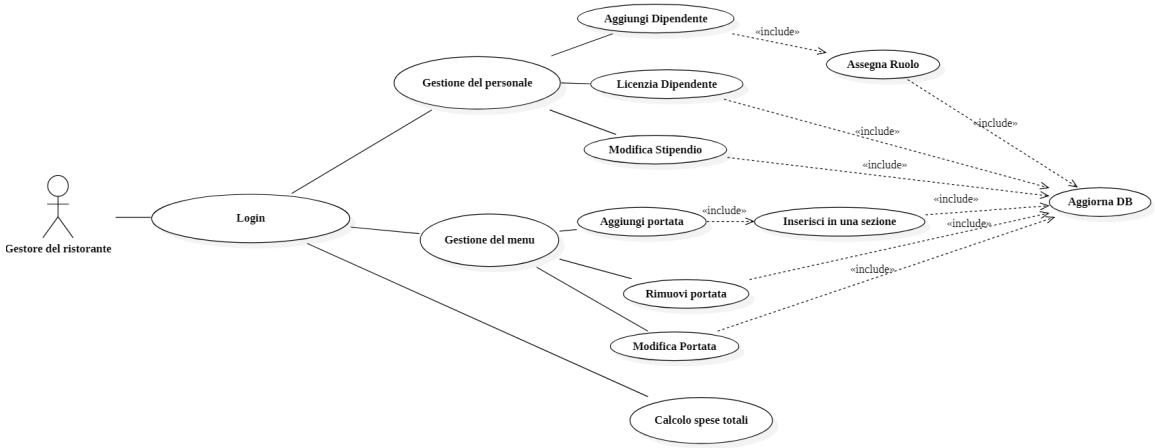

Figura 2: Use Case Diagram Admin

3 Mockups

La GUI (Interfaccia Utente Grafica) è stata implementata usando il framework ***JavaFX***. Riportiamo di seguito i mockups realizzati relativi alla GUI per l'interazione dell'utente con ordine e dell'ammistratore con la gestione del ristorante. Oltre ad agire come mockup, la GUI è funzionante in tutti i casi d'uso precedentemente elencati.

3.1 Login View

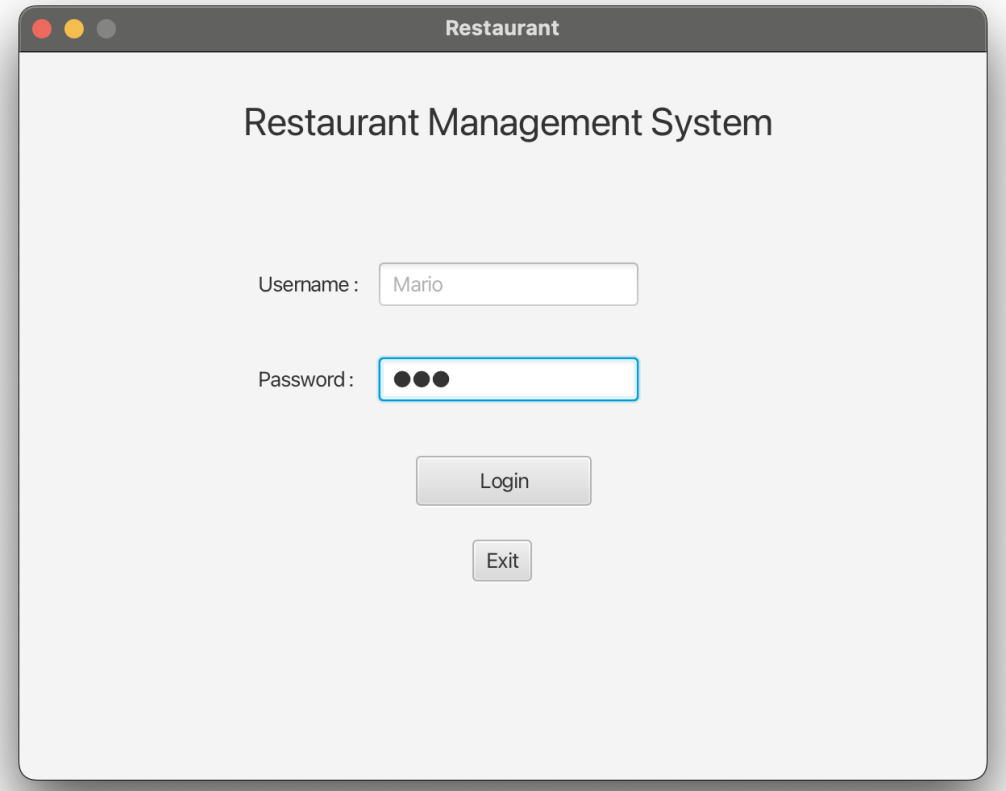

Figura 3: Login Mockup

3.2 Customer View

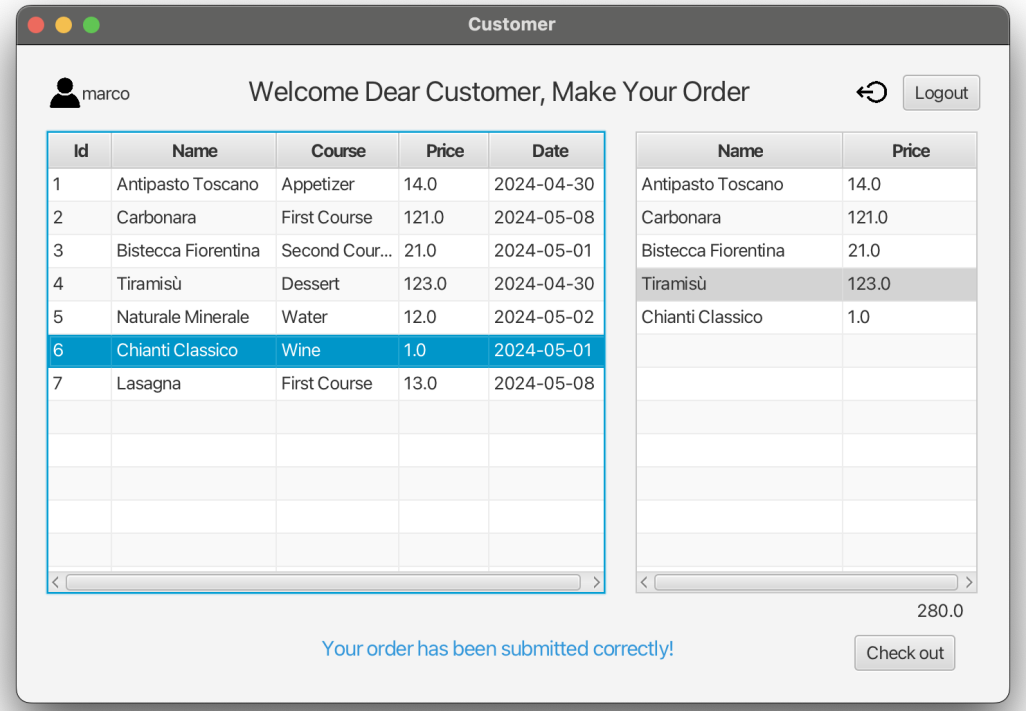

Figura 4: Customer Mockup

3.3 Adminstrator View

3.3.1 Admin View

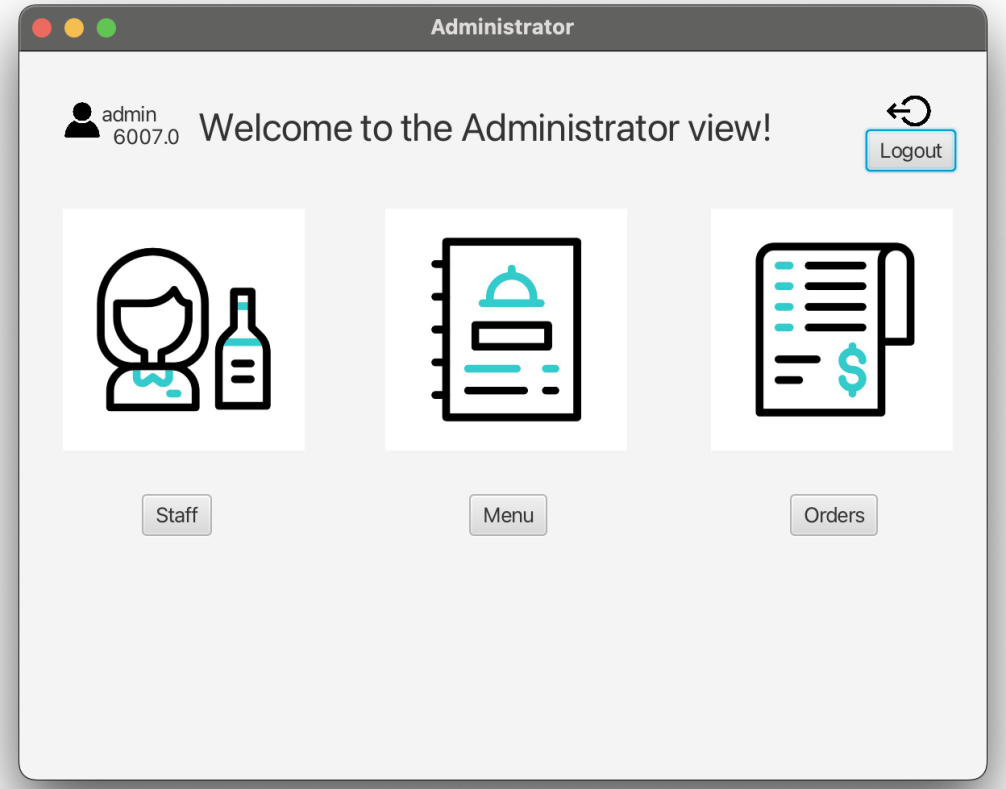

Figura 5: Admin Mockup

3.3.2 Staff View

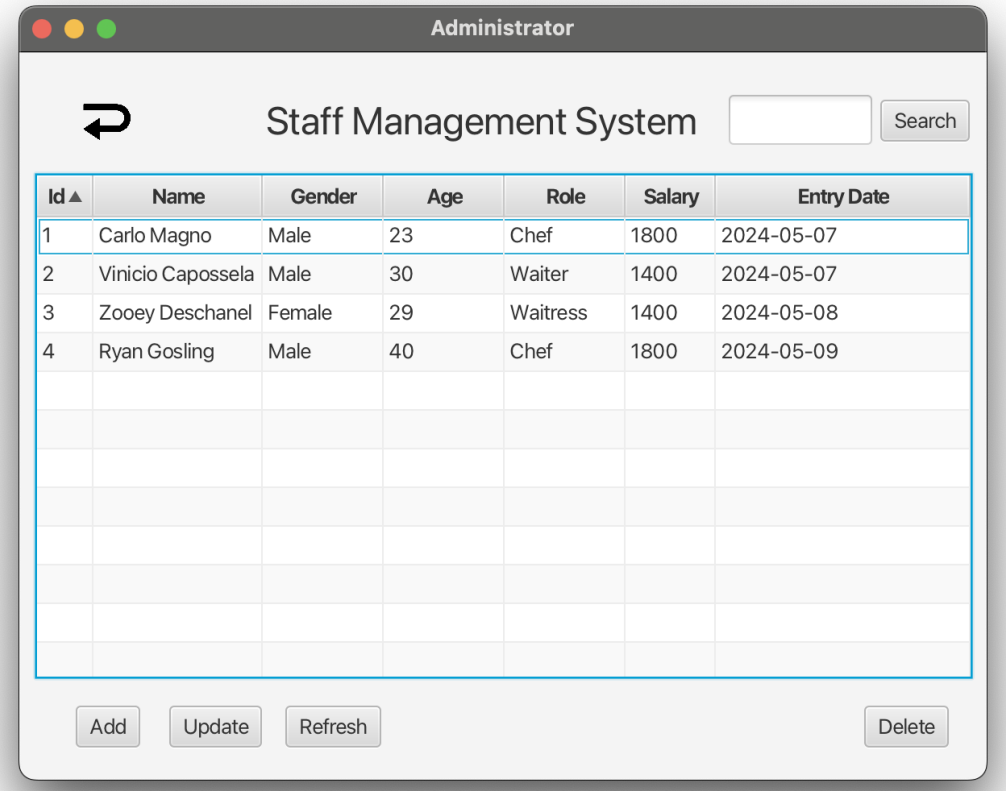

Figura 6: Staff Mockup

3.3.3 Menu View

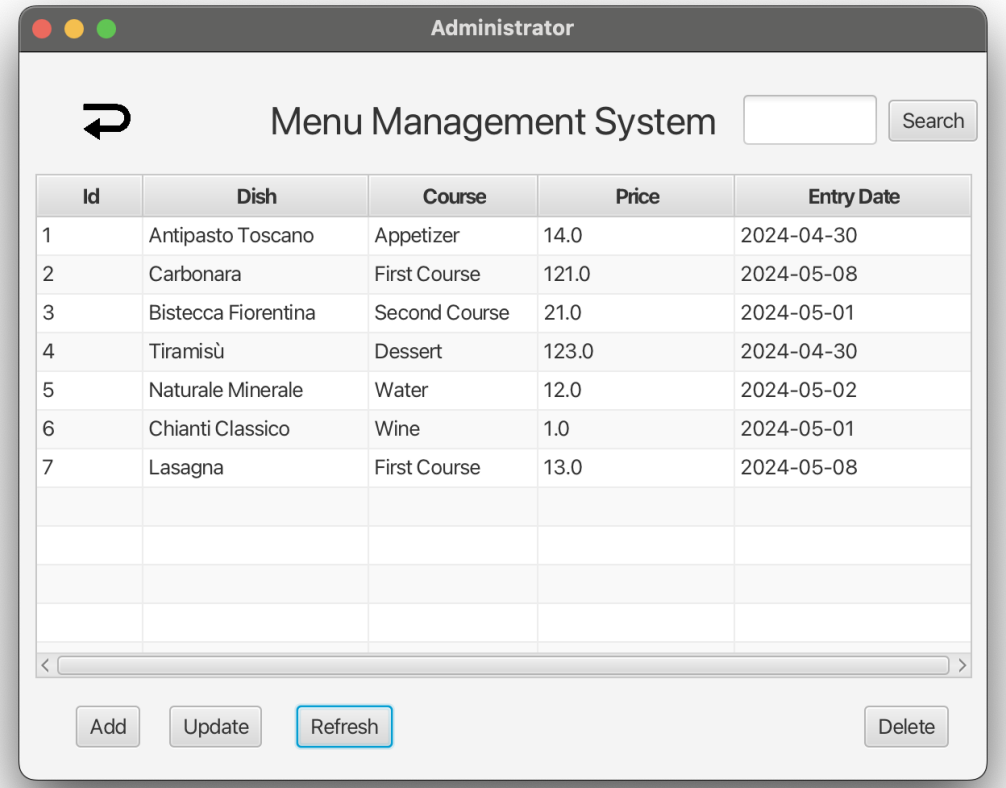

Figura 7: Menu Mockup

3.3.4 Orders View

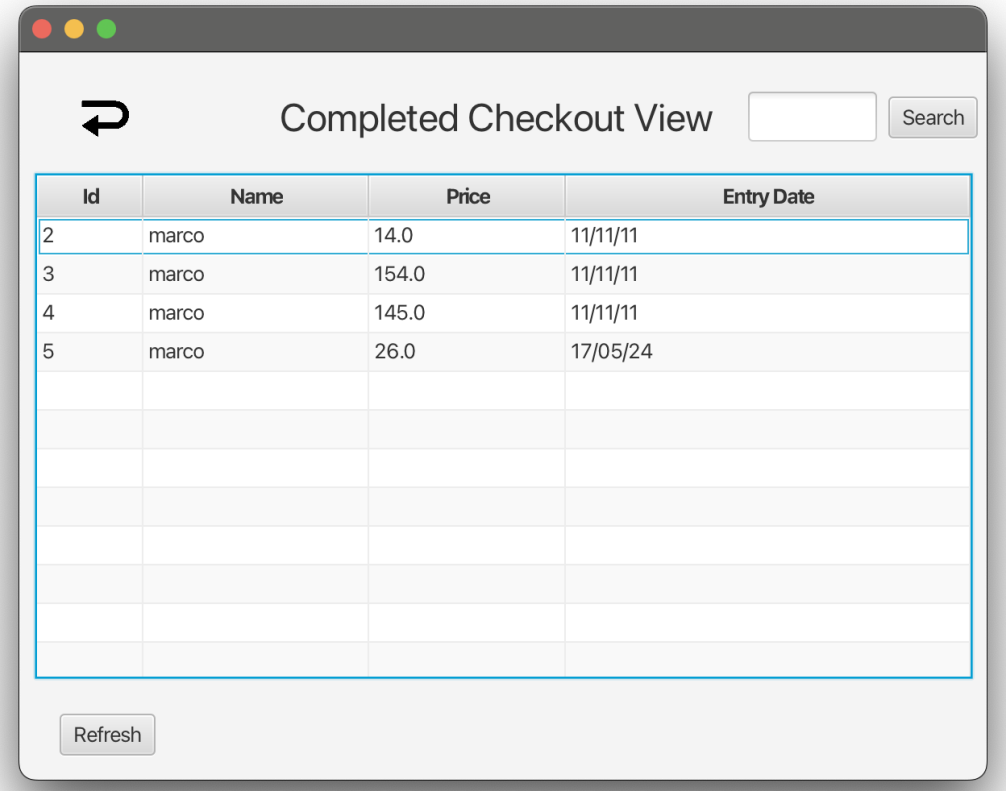

Figura 8: Orders Mockup

Capitolo III

Progettazione E Implementazione

4 Scelte implementative e considerazioni

Per implementare il nostro progetto in linguaggio Java ed eseguire i test ci siamo serviti dell'*IDE IntelliJ IDEA*. Al fine di semplificare la collaborazione abbiamo utilizzato *GitHub* come stumento di versionamento del codice. Per quanto riguarda il Class Diagram, l'Use Case Diagram e lo schema architetturale in packages ci siamo serviti del software *StarUML*.

5 Project Directory

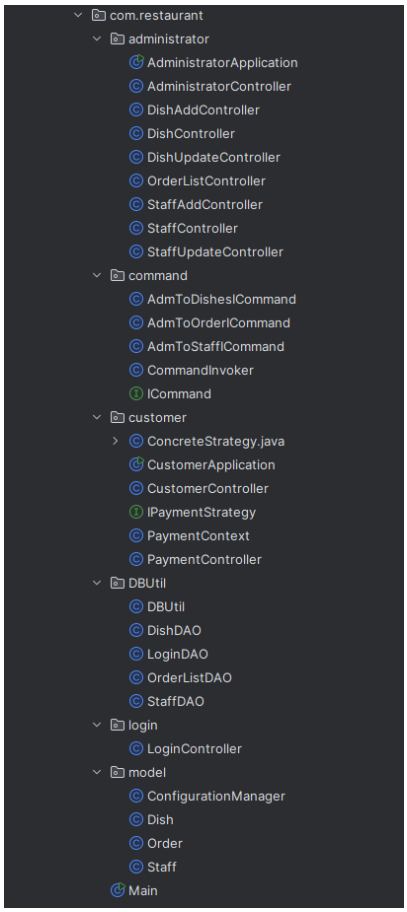

Figura 9: Src

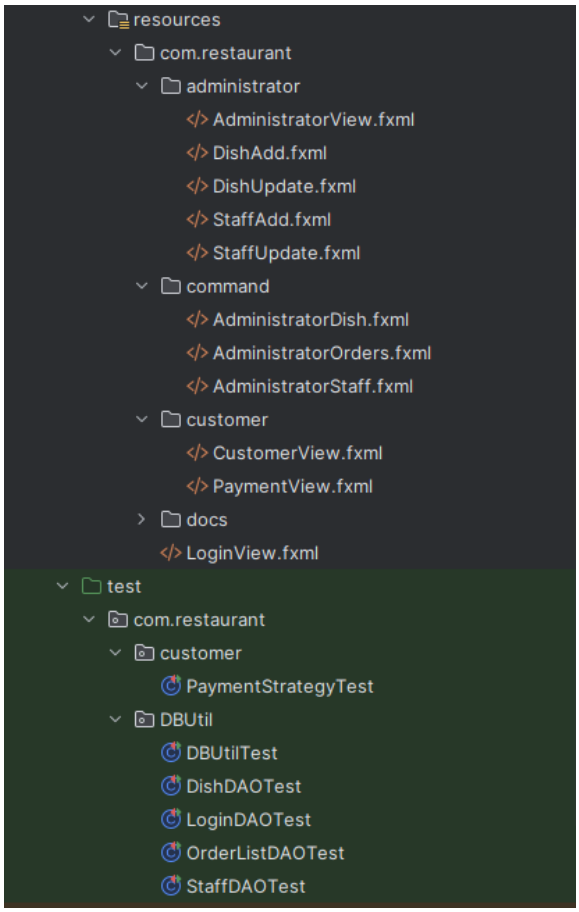

Figura 10: Resources and Test

6 Class Diagram

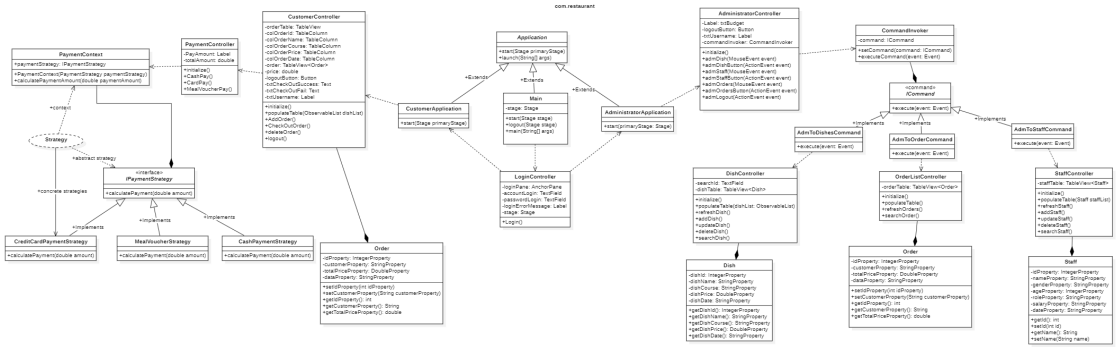

Figura 11: com.restaurant

6.1 Main

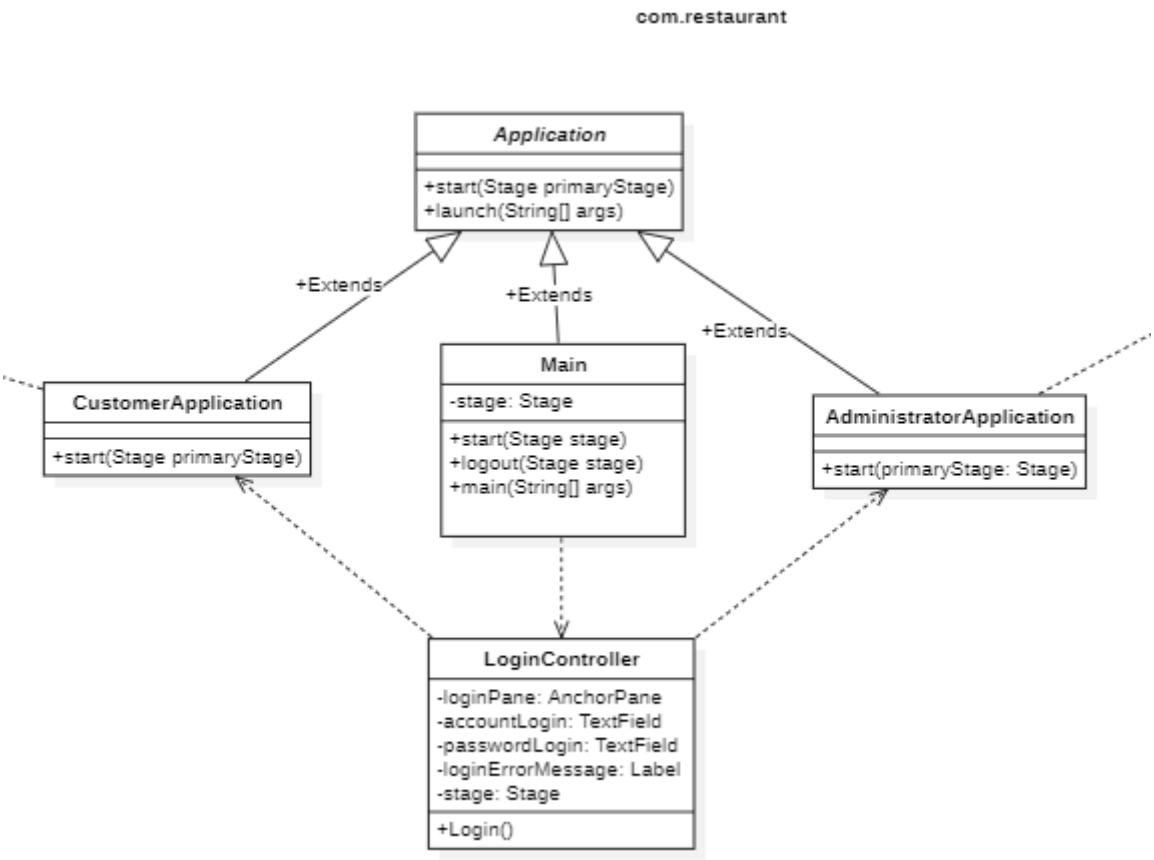

Figura 12: Main Diagram

6.1.1 Descrizione del Main

Il main del progetto è scritto utilizzando `JavaFX` e funge da punto di ingresso per l'applicazione. Questo segue il pattern architetturale Model-View-Controller (MVC) per separare le diverse responsabilità all'interno dell'applicazione. La classe principale estende `Application`, una classe di base fornita da JavaFX per le applicazioni JavaFX. Il metodo `main` lancia l'applicazione invocando il metodo `launch(args)`. Una volta avviata l'applicazione, il metodo `start` viene chiamato. Questo metodo è responsabile dell'inizializzazione della scena principale. In questa fase, viene caricata la vista del login (un file `.fxml`) e il relativo controller viene associato a tale vista.

6.1.2 Interazione con il Controller

Il `LoginController` gestisce la logica di autenticazione. Quando l'utente inserisce le credenziali, il controller verifica la validità di queste ultime. A seconda del ruolo dell'utente (amministratore o cliente), il controller avvia l'applicazione corrispondente.

6.1.3 Conclusione

Questa prima parte del class diagram rappresenta il flusso iniziale dell'applicazione, a partire dal punto di ingresso fino alla gestione delle credenziali di login e il successivo avvio dell'applicazione amministrativa o dell'applicazione cliente. Questo approccio basato su MVC consente una chiara separazione delle responsabilità, facilitando la manutenzione e l'estensibilità del progetto.

6.2 Administrator

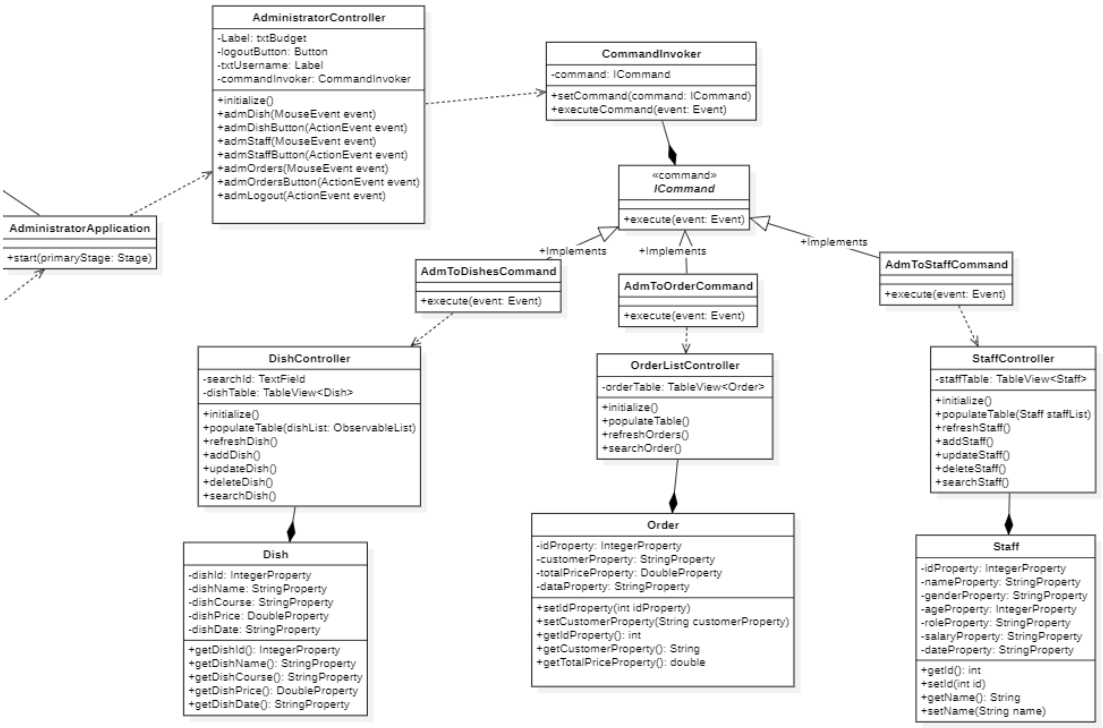

Figura 13: Administrator Diagram

Dopo il login, l'applicazione carica la vista principale dell'amministratore (`AdminView.fxml`) e associa il rispettivo controller, `AdministratorController`.

6.2.1 AdministratorController

Il `AdministratorController` gestisce la logica operativa dell'interfaccia amministrativa. Questo controller utilizza il pattern Command per delegare le operazioni specifiche ai rispettivi comandi:

- `AdmToDishCommand`
- `AdmToStaffCommand`
- `AdmToOrderCommand`

6.2.2 AdmToDishCommand

Questo comando richiama il `DishController`, che gestisce tutte le operazioni relative ai piatti (dish) all'interno del sistema. Il `DishController` è responsabile della creazione, aggiornamento, eliminazione e visualizzazione dei piatti.

6.2.3 AdmToStaffCommand

Questo comando richiama il `StaffController`, che gestisce tutte le operazioni relative al personale (staff). Il `StaffController` si occupa della gestione dei dipendenti, inclusi l'assunzione, la modifica delle informazioni e il licenziamento.

6.2.4 AdmToOrderCommand

Questo comando richiama il `OrderController`, che gestisce tutte le operazioni relative agli ordini (order). Il `OrderController` sovrintende alla gestione degli ordini, inclusa la creazione, l'aggiornamento e la cancellazione degli ordini.

6.2.5 Conclusione

Questa parte del class diagram mostra come l'applicazione amministrativa organizzi le diverse funzionalità attraverso il pattern MVC e il pattern Command. L'uso dell'`AdministratorController` per coordinare le operazioni e delegarle ai rispettivi controller tramite comandi dedicati permette una gestione modulare e facilmente estensibile dell'applicazione. In questo modo, ogni aspetto dell'amministrazione (piatti, personale, ordini) è ben isolato e può essere sviluppato e mantenuto indipendentemente.

6.3 Customer

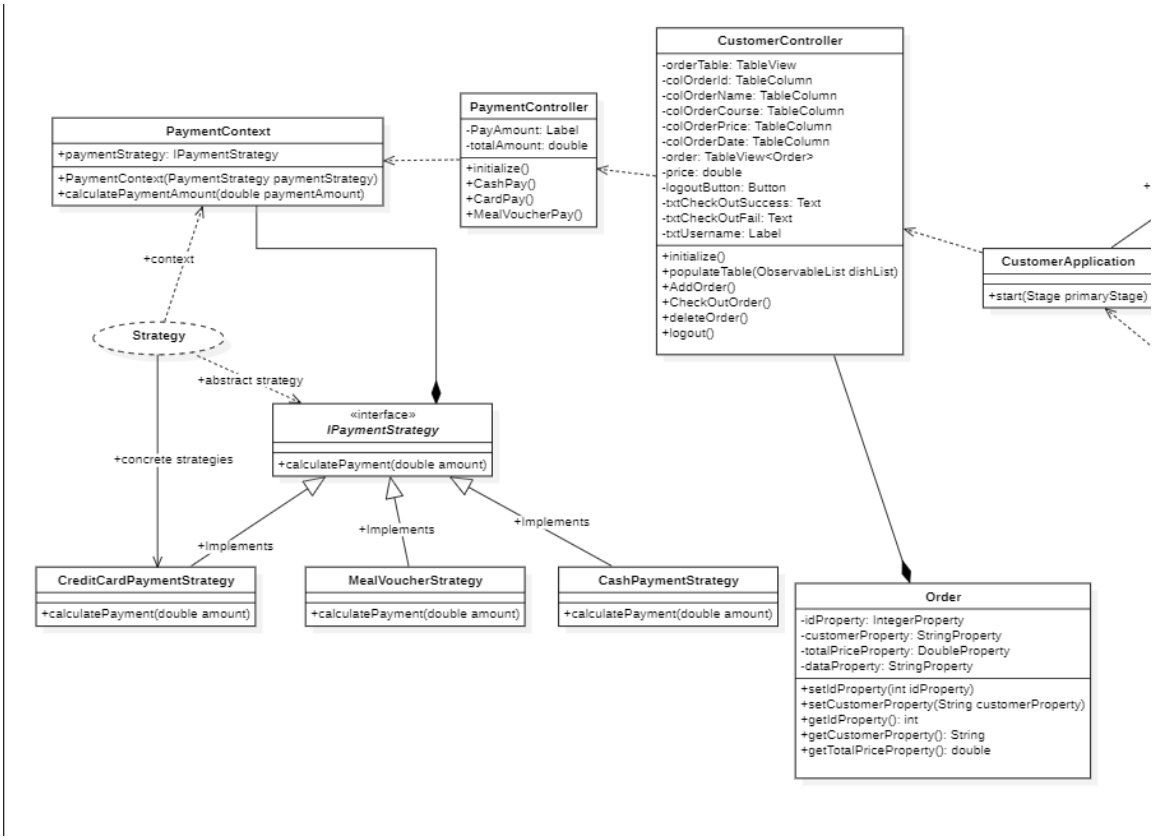

Figura 14: Customer Diagram

Dopo il login, l'applicazione carica la vista principale del cliente (`CustomerView.fxml`) e associa il rispettivo controller, `CustomerController`.

6.3.1 CustomerController

Il `CustomerController` gestisce la logica operativa dell'interfaccia cliente. Questo controller permette ai clienti di effettuare ordini e gestire i pagamenti.

6.3.2 Gestione degli Ordini

Il `CustomerController` fornisce funzionalità per permettere ai clienti di creare, visualizzare e modificare i loro ordini. Questo include la selezione dei piatti, la quantità e altre specifiche dell'ordine.

6.3.3 Gestione dei Pagamenti

Per gestire i pagamenti, il `CustomerController` utilizza il pattern Strategy. Questo pattern permette di selezionare dinamicamente l'algoritmo di pagamento da utilizzare in base alla scelta del cliente. Le tre strategie di pagamento implementate sono:

- `CreditCardPaymentStrategy`

- `CashPaymentStrategy`
- `MealVoucherPaymentStrategy`

6.3.4 CreditCardPaymentStrategy

Questa strategia gestisce i pagamenti effettuati tramite carta di credito. Include la logica per validare le informazioni della carta e processare la transazione attraverso un gateway di pagamento.

6.3.5 CashPaymentStrategy

Questa strategia gestisce i pagamenti in contanti. Include la logica per registrare un pagamento in contanti e aggiornare lo stato dell'ordine di conseguenza.

6.3.6 MealVoucherPaymentStrategy

Questa strategia gestisce i pagamenti tramite buoni pasto. Include la logica per validare i buoni pasto e registrare il pagamento corrispondente.

6.3.7 Conclusione

Questa parte del class diagram mostra come l'applicazione cliente organizzi le diverse funzionalità attraverso il pattern MVC e il pattern Strategy. L'uso del `CustomerController` per coordinare le operazioni e delegarle alle rispettive strategie di pagamento permette una gestione modulare e facilmente estensibile dell'applicazione. In questo modo, ogni aspetto dell'ordinazione e del pagamento è ben isolato e può essere sviluppato e mantenuto indipendentemente, offrendo al cliente una esperienza d'uso flessibile e intuitiva.

6.4 DBUtil

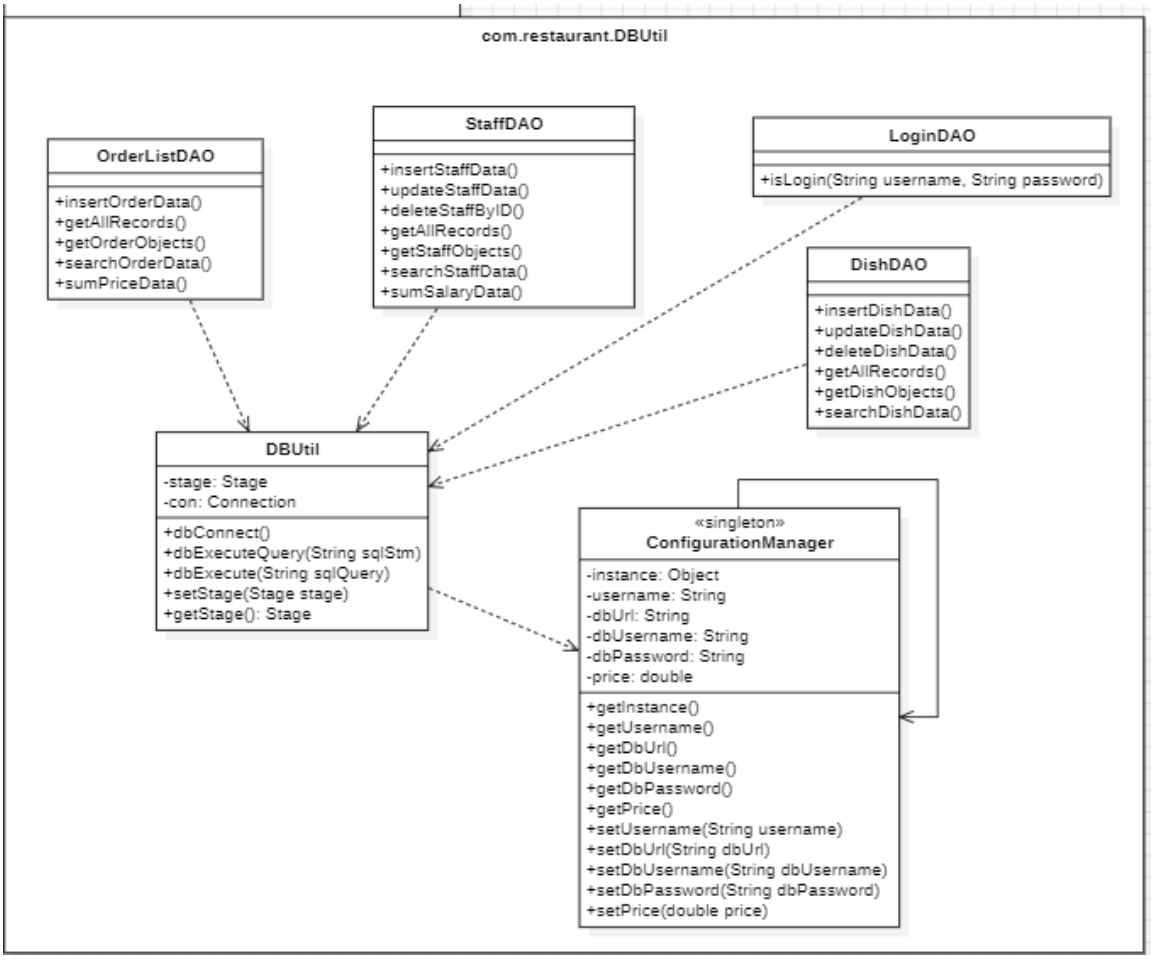

Figura 15: DBUtil Diagram

Questa parte del class diagram descrive come l'applicazione gestisce la connessione al database e l'esecuzione delle query SQL.

6.4.1 DBUtil

La classe `DBUtil` è responsabile della gestione della connessione al database, inclusi connessione, disconnessione ed esecuzione delle query SQL.

6.4.2 Connessione e Disconnessione

La classe `DBUtil` fornisce metodi per stabilire e chiudere la connessione con il database. Questi metodi utilizzano le informazioni di configurazione fornite dal `ConfigurationManager`.

6.4.3 Esecuzione delle Query SQL

La classe `DBUtil` include metodi per eseguire query SQL, inclusi `SELECT`, `INSERT`, `UPDATE` e `DELETE`. Questi metodi accettano query SQL come stringhe e parametri opzionali per le query parametrizzate.

6.4.4 DAO (Data Access Objects)

Le classi DAO utilizzano `DBUtil` per eseguire le operazioni CRUD (Create, Read, Update, Delete) sul database. Le principali classi DAO nel progetto sono:

- `OrderListDAO`
- `StaffDAO`
- `LoginDAO`
- `DishDAO`

6.4.5 OrderListDAO

Questa classe gestisce tutte le operazioni relative agli ordini (order) nel database. Utilizza `DBUtil` per eseguire query SQL e gestire i dati degli ordini.

6.4.6 StaffDAO

Questa classe gestisce tutte le operazioni relative al personale (staff) nel database. Utilizza `DBUtil` per eseguire query SQL e gestire i dati del personale.

6.4.7 LoginDAO

Questa classe gestisce tutte le operazioni relative all'autenticazione degli utenti nel database. Utilizza `DBUtil` per verificare le credenziali di login e gestire i dati degli utenti.

6.4.8 DishDAO

Questa classe gestisce tutte le operazioni relative ai piatti (dish) nel database. Utilizza `DBUtil` per eseguire query SQL e gestire i dati dei piatti.

6.4.9 ConfigurationManager

Il `ConfigurationManager` è una classe singleton che contiene le informazioni di configurazione del database, come `dbURL`, `dbUsername` e `dbPassword`. Inoltre, tiene traccia dell'utente corrente (cliente o amministratore) che sta utilizzando l'applicazione.

6.4.10 Conclusione

Questa parte del class diagram illustra come l'applicazione gestisce la connessione al database e le operazioni CRUD attraverso le classi `DBUtil` e DAO. L'uso del `ConfigurationManager` come singleton garantisce che le informazioni di configurazione del database siano gestite in modo centralizzato e sicuro. Questo approccio modulare e strutturato facilita la manutenzione e l'estensibilità dell'applicazione, assicurando un'interazione affidabile ed efficiente con il database.

7 Design Pattern

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:

- **MVC**
- **Singleton**
- **Commander**
- **Strategy**
- **DAO**

7.1 MVC

7.1.1 Informazioni generali sull'MVC

Nel contesto della progettazione e implementazione del nostro sistema di gestione per ristoranti, abbiamo adottato il design pattern ***Model-View-Controller*** (MVC) per strutturare l'applicazione in modo modulare e mantenibile. Il pattern MVC, noto per la sua capacità di separare le preoccupazioni, è una scelta consolidata per lo sviluppo di interfacce utente complesse, in quanto promuove una chiara divisione delle responsabilità tra i componenti del sistema. Suddivide l'applicazione in tre componenti principali:

- **Model (Modello):** Il modello rappresenta la logica dell'applicazione e gestisce i dati. Esso è responsabile della definizione delle strutture dei dati, delle regole di business, delle logiche di accesso ai dati e delle interazioni con il database. Nel nostro progetto, il modello comprende le classi che gestiscono i dipendenti, il menu e gli ordini, interfacciandosi con il database MySQL per memorizzare e recuperare informazioni.
- **View (Vista):** La vista è la componente che si occupa della presentazione dei dati. Essa genera l'interfaccia utente e rende visibili le informazioni del modello. Le viste sono responsabili della ricezione degli input utente e della visualizzazione dei dati in modo intuitivo e user-friendly. Nel nostro sistema, abbiamo utilizzato JavaFX per creare interfacce grafiche accattivanti e funzionali sia per gli amministratori che per i clienti.

- **Controller (Controllore):** Il controllore funge da intermediario tra il modello e la vista. Esso gestisce l'input dell'utente, elabora le richieste e aggiorna il modello e la vista di conseguenza. Il controller interpreta le azioni dell'utente (come clic, inserimenti di dati, ecc.), invoca i metodi appropriati del modello e aggiorna la vista per riflettere i cambiamenti nello stato dell'applicazione.

7.1.2 Implementazione del Pattern MVC nel Progetto

Abbiamo implementato il pattern nel seguente modo:

- **Model: Dish, Order e Staff** sono esempi semplici di classi dei modelli che abbiamo designato.
  In *Dish* abbiamo creato il modello del menu, o per meglio dire della creazione del menu e quindi ivi sono definite le proprietà associata ad ogni pietanza, ad esempio a quale portata appartiene ed il suo prezzo. Rispettivamente lo stesso avviene per Order e Staff.
  Per semplicità mostreremo come esempio il codice riguardante *Dish*:

Listing 1: Codice Java Implementazione del modello *Dish*

```
1  public class Dish {
2
3      private final Integer id;
4      private final String dish;
5      private final String course;
6      private final Double price;
7      private final String date;
8
9      public Dish(){
10         this.id = new Integer();
11         this.dish = new String();
12         this.course = new String();
13         this.price = new Double();
14         this.date = new String();
15     }
16
17     //for id
18
19     public int getId() {
20         return id;
21     }
22
23     public void setId(int id) {
24         this.id = id;
25     }
26
27     //for dish name
28
29     public String getDish() {
30         return dish;
31     }
32
33     public void setDish(String dish) {
34         this.dish = dish;
35     }
```

```
36
37      //for course
38
39      public String getCourse() {
40          return course;
41      }
42
43      public void setCourse(String course) {
44          this.course = course;
45      }
46
47      //for price
48
49      public double getPrice() {
50          return price;
51      }
52
53      public void setPrice(double price) {
54          this.price = price;
55      }
56
57      //for date
58
59      public String getDate() {
60          return date;
61      }
62
63      public void setDate(String date) {
64          this.date = date;
65      }
66  }
```

- **View:** La View, implementata attraverso JavaFX, è stata implementata attraverso i file *.fxml*, anche in questo caso per comodità e coerenza analizzeremo l'interfaccia riguardante la parte dei piatti del menu mostrata all'amministratore, creata attraverso vari bottoni interagibili e una finestra interamente personalizzata per ogni esigenza richiesta da un qualsiasi gestore di un ristorante.

Listing 2: Codice Java Implementazione della Vista *AdministratorDish.fxml*

```
1  <AnchorPane prefHeight="452.0" prefWidth="600.0" xmlns="http://
       javafx.com/javafx/21" xmlns:fx="http://javafx.com/fxml/1" fx:
       controller="com.restaurant.administrator.
       AdministratorController">
2      <ImageView layoutX="-217.0" layoutY="-213.0" scaleX="0.045"
           scaleY="0.045" scaleZ="0.045">
3          <Image url="@../docs/admin_icon.png" />
4      </ImageView>
5      <Button fx:id="logoutButton" layoutX="525.0" layoutY="49.0"
           mnemonicParsing="false" onAction="#admLogout" text="Logout"
           />
6      <Button layoutX="279.0" layoutY="275.0" mnemonicParsing="false"
            onAction="#admDishButton" text="Menu" />
7      <Button layoutX="76.0" layoutY="275.0" mnemonicParsing="false"
           onAction="#admStaffButton" text="Staff" />
```

```
8     <Button layoutX="478.0" layoutY="275.0" mnemonicParsing="false"
          onAction="#admOrdersButton" text="Orders" />
9     <Label fx:id="txtUsername" layoutX="51.0" layoutY="31.0"
         prefHeight="17.0" prefWidth="64.0" text="Test" />
10    <Label fx:id="txtBudget" layoutX="58.0" layoutY="43.0"
         prefHeight="21.0" prefWidth="65.0" text="0.0" />
11    <ImageView fx:id="gifStaff" fitHeight="150.0" fitWidth="200.0"
         layoutX="27.0" layoutY="98.0" onMouseClicked="#admStaff"
         onMouseEntered="#gifHoverStaff" onMouseExited="#gifExitStaff
         " pickOnBounds="true" preserveRatio="true" visible="false">
12       <Image url="@../docs/waitress (anim).gif" />
13    </ImageView>
14    <ImageView fx:id="imgStaff" fitHeight="150.0" fitWidth="200.0"
         layoutX="27.0" layoutY="98.0" onMouseEntered="#gifHoverStaff
         " onMouseExited="#gifExitStaff" pickOnBounds="true"
         preserveRatio="true">
15       <Image url="@../docs/waitress (anim).jpg" />
16    </ImageView>
17    <ImageView fx:id="gifDishes" fitHeight="150.0" fitWidth="200.0"
          layoutX="227.0" layoutY="98.0" onMouseClicked="#admDish"
         onMouseEntered="#gifHoverDishes" onMouseExited="#
         gifExitDishes" pickOnBounds="true" preserveRatio="true"
         visible="false">
18       <Image url="@../docs/menu(anim).gif" />
19    </ImageView>
20    <ImageView fx:id="imgDishes" fitHeight="150.0" fitWidth="200.0"
          layoutX="227.0" layoutY="98.0" onMouseEntered="#
         gifHoverDishes" onMouseExited="#gifExitDishes" pickOnBounds=
         "true" preserveRatio="true">
21       <Image url="@../docs/menu.jpg" />
22    </ImageView>
23    <ImageView fx:id="gifOrders" fitHeight="150.0" fitWidth="200.0"
          layoutX="429.0" layoutY="98.0" onMouseClicked="#admOrders"
         onMouseEntered="#gifHoverOrders" onMouseExited="#
         gifExitOrders" pickOnBounds="true" preserveRatio="true"
         visible="false">
24       <Image url="@../docs/bill(anim).gif" />
25    </ImageView>
26    <ImageView fx:id="imgOrders" fitHeight="150.0" fitWidth="200.0"
          layoutX="429.0" layoutY="98.0" onMouseEntered="#
         gifHoverOrders" onMouseExited="#gifExitOrders" pickOnBounds=
         "true" preserveRatio="true">
27       <Image url="@../docs/bill.jpg" />
28    </ImageView>
29    <Label layoutX="111.0" layoutY="30.0" prefHeight="35.0"
         prefWidth="381.0" text="Welcome to the Administrator view!">
30       <font>
31          <Font size="24.0" />
32       </font>
33    </Label>
34    <ImageView fitHeight="35.0" fitWidth="31.0" layoutX="536.0"
         layoutY="22.0" pickOnBounds="true" preserveRatio="true">
35       <Image url="@../docs/logoutArrow.png" />
36    </ImageView>
37 </AnchorPane>
```

- **Controller:** Infine, la classe dei controllori, come ad esempio *DishController*, interagisce con la View attraverso la dicitura **@FXML**, con la quale, ad esempio controlliamo ciò che avviene quando viene cliccato un bottone oppure cosa prendere dal database per aggiungerlo alla tabella, attraverso metodi di ricerca, cancellazione, aggiunta e ricerca. Di seguito mostriamo ancora una volta il codice riguardante l'implementazione per il menu.

Listing 3: Codice Java Implementazione del Controller *DishController.fxml*

```
1  public class DishController {
2
3      @FXML private TextField searchId;
4
5      @FXML TableView<Dish> dishTable;
6      @FXML private TableColumn<Dish, Integer> colDishId;
7      @FXML private TableColumn<Dish, String> colDishName;
8      @FXML private TableColumn<Dish, String> colDishCourse;
9      @FXML private TableColumn<Dish, Double> colDishPrice;
10     @FXML private TableColumn<Dish, String> colDishDate;
11
12     @FXML private ImageView returnButton;
13
14     @FXML
15     public void initialize() throws Exception{
16         colDishId.setCellValueFactory(cellData -> cellData.
               getValue().getDishId().asObject());
17         colDishName.setCellValueFactory(cellData -> cellData.
               getValue().getDishName());
18         colDishCourse.setCellValueFactory(cellData -> cellData.
               getValue().getDishCourse());
19         colDishPrice.setCellValueFactory(cellData -> cellData.
               getValue().getDishPrice().asObject());
20         colDishDate.setCellValueFactory(cellData -> cellData.
               getValue().getDishDate());
21         ObservableList<Dish> dishList = DishDAO.getAllRecords();
22         populateTable(dishList);
23     }
24
25     private void populateTable(ObservableList<Dish> dishList) {
26         dishTable.setItems(dishList);
27     }
28
29     @FXML
30     public void refreshDish() throws SQLException,
           ClassNotFoundException {
31         ObservableList<Dish> dishList = DishDAO.getAllRecords();
32         populateTable(dishList);
33     }
34
35     @FXML
36     public void addDish() throws Exception{
37         try {
38             Parent root = FXMLLoader.load(Objects.requireNonNull(
                   getClass().getResource("DishAdd.fxml")));
39             Scene admDish = new Scene(root);
40
```

```
41              Stage window = new Stage();
42              window.setScene(admDish);
43              window.show();
44          }
45          catch (Exception e) {
46              System.out.println("Error occurred while opening
                    addDish page");
47              throw e;
48          }
49      }
50
51      public void updateDish() throws Exception{
52          try{
53              Dish dish = dishTable.getSelectionModel().
                    getSelectedItem();
54              if (dish != null) {
55                  DishUpdateController.setCurrent(dish);
56                  Parent root = FXMLLoader.load(Objects.
                        requireNonNull(getClass().getResource("
                        DishUpdate.fxml")));
57
58                  Stage window = new Stage();
59                  window.setScene(new Scene(root));
60                  window.show();
61              }
62          }
63          catch (Exception e) {
64              System.out.println("Error occurred while opening
                    updateDish page");
65              throw e;
66          }
67      }
68
69      @FXML
70      public void deleteDish() throws SQLException,
            ClassNotFoundException {
71          try {
72              Dish dish = dishTable.getSelectionModel().
                    getSelectedItem();
73              if (dish != null) {
74                  Alert alert = new Alert(Alert.AlertType.
                        CONFIRMATION);
75                  alert.setTitle("Confirm Deletion");
76                  alert.setHeaderText("Do you really want to delete
                        this dish?");
77                  alert.setContentText("Press OK to delete this dish
                        ");
78
79                  if (alert.showAndWait().get() == ButtonType.OK) {
80                      DishDAO.deleteDishData(dish.getId());
81                      ObservableList<Dish> dishList = DishDAO.
                            getAllRecords();
82                      populateTable(dishList);
83                  }
84              }
85          }
```

```
86          catch (SQLException e) {
87              System.out.println("Error occurred while deleting dish
                    ");
88              throw e;
89          }
90      }
91
92      @FXML
93      public void searchDish() throws SQLException,
            ClassNotFoundException {
94          if (searchId.getText().isEmpty()){
95              ObservableList<Dish> dishList = DishDAO.getAllRecords
                    ();
96              populateTable(dishList);
97          }
98          else {
99              ObservableList<Dish> dishList = DishDAO.searchDishData
                    (searchId.getText());
100
101             if (!dishList.isEmpty()) {
102                 populateTable(dishList);
103             }
104             else {
105                 System.out.println("Dish not found");
106             }
107         }
108     }
109
110     public void return_back(MouseEvent event) throws Exception {
111         try {
112             //si possono togliere queste due righe?
113             Stage stage = (Stage) returnButton.getScene().
                    getWindow(); // Ottieni il riferimento alla
                    finestra di AdministratorDish
114             stage.close(); // Chiudi la finestra di
                    AdministratorDish
115
116             Parent root = FXMLLoader.load(Objects.requireNonNull(
                    getClass().getResource("AdministratorView.fxml")));
117             Scene admDish = new Scene(root);
118
119             Stage window = new Stage();
120             window.setScene(admDish);
121             window.show();
122         }
123         catch (Exception e) {
124             System.out.println("Error occurred while opening
                    Administrator page");
125             throw e;
126         }
127     }
128 }
```

7.1.3 Vantaggi dell'Uso del Pattern MVC

L'adozione del pattern MVC nel nostro progetto ha offerto numerosi vantaggi:

- **Manutenibilità:** La separazione delle preoccupazioni ha reso il codice più facile da mantenere e aggiornare. Le modifiche al modello, alla vista o al controller potevano essere effettuate indipendentemente, riducendo il rischio di introdurre errori.
- **Riutilizzabilità:** Le componenti del modello e del controller potevano essere riutilizzate in diverse parti dell'applicazione, riducendo la duplicazione del codice.
- **Collaborazione:** La chiara separazione tra modello, vista e controller ha facilitato la collaborazione, permettendoci di lavorare simultaneamente su diverse parti dell'applicazione senza conflitti.

7.2 Singleton

7.2.1 Informazioni generali su Singleton

Abbiamo integrato il design pattern Singleton per garantire che alcune classi abbiano una singola istanza condivisa in tutto il sistema. Il pattern Singleton è ideale per la gestione di risorse che devono essere uniche e accessibili globalmente, come la connessione al database o la configurazione dell'applicazione. Il design pattern Singleton assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale a tale istanza. Le caratteristiche principali del pattern Singleton sono:

1. **Unica Istanza:** La classe Singleton mantiene un riferimento statico a una singola istanza della classe stessa e previene la creazione di altre istanze.
2. **Accesso Globale:** Fornisce un metodo statico per ottenere l'unica istanza, permettendo un accesso controllato e globale alla risorsa.

7.2.2 Implementazione del Pattern Singleton nel Progetto

Si è scelto di implementare il design pattern Singleton all'interno della classe ***ConfigurationManager***, con l'obiettivo di gestire la configurazione dell'applicazione in maniera centralizzata e sicura.*ConfigurationManager* è progettata per garantire che una singola istanza sia creata e mantenuta durante l'intero ciclo di vita dell'applicazione. Questo è ottenuto attraverso il metodo statico `getInstance()`, che segue la seguente logica:

- Se l'istanza non è ancora stata creata (`instance == null`), viene istanziata una nuova *ConfigurationManager*.
- Altrimenti, se l'istanza esiste già, viene restituita la stessa istanza.

Si occupa di immagazzinare e gestire informazioni essenziali come le credenziali di accesso al database, l'username dell'utente autenticato, l'URL del database e il valore di prezzo rilevante per l'applicazione.
Queste informazioni sono cruciali poiché vengono utilizzate in diversi contesti all'interno

dell'applicazione e devono essere facilmente accessibili ma al contempo protette da accessi concorrenti o non autorizzati. Un esempio banale è l'utilizzo dell'username dell'utente che ha effettuato l'accesso che verrà mostrato all'interno dell'interfaccia utente corrispondente al ruolo di colui che si è connesso.
Ecco come viene implementato questo meccanismo:

Listing 4: Codice Java Implementazione del Singleton

```java
package com.restaurant.model;

public class ConfigurationManager {

    private static ConfigurationManager instance;

    private String username;
    private String dbUrl;
    private String dbUsername;
    private String dbPassword;
    private double price;

    private ConfigurationManager() {
        this.username = ""; //del login
        this.dbUrl = "jdbc:mysql://127.0.0.1:3306/db.restaurant"; //per
            database
        this.dbUsername = "root"; //per database
        this.dbPassword = "root"; //per database
        this.price = 0.0;
    }

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

    public void setPrice(double price) {
        this.price = price;
    }

    public double getPrice() {
        return price;
    }

    public String getDbUsername() {
        return dbUsername;
    }

    public String getDbPassword() {
        return dbPassword;
    }

    public String getDbUrl() {
        return dbUrl;
    }

```

```
48      public String getUsername() {
49          return username;
50      }
51
52      public void setUsername(String username) {
53          this.username = username;
54      }
55  }
```

Il costruttore della classe è dichiarato privato per impedire la creazione di istanze multiple al di fuori della classe stessa. Questo assicura che tutte le configurazioni vitali dell'applicazione siano gestite in modo centralizzato, riducendo il rischio di inconsistenze e facilitando la gestione delle risorse.

7.2.3 Vantaggi dell'Uso del Pattern Singleton

L'adozione del pattern Singleton ha offerto numerosi vantaggi nel nostro progetto:

- **Gestione delle Risorse:** Ha garantito una gestione efficiente e sicura delle risorse condivise, come la connessione al database, prevenendo la creazione di più connessioni e riducendo il carico sul database.
- **Accesso Globale Controllato:** Ha fornito un punto di accesso globale controllato alla connessione al database, semplificando il codice e migliorando la manutenibilità.
- **Inizializzazione Pigra:** Ha permesso l'inizializzazione pigra dell'istanza Singleton, assicurando che la connessione al database venga creata solo quando effettivamente necessaria, migliorando l'efficienza.

7.3 Command

7.3.1 Informazioni generali su Command

Abbiamo utilizzato il design pattern Command per incapsulare tutte le richieste come oggetti, consentendo così di parametrizzare i metodi con richieste diverse, ritardare o mettere in coda l'esecuzione di un comando e supportare le operazioni di annullamento. Il pattern Command è particolarmente utile per la gestione delle operazioni che devono essere eseguite, annullate o ripetute, fornendo una soluzione flessibile e estensibile per il controllo delle azioni. Il design pattern Command è composto da quattro componenti principali:

1. **Command (Comando):** Un'interfaccia che dichiara un metodo execute per eseguire l'azione.
2. **ConcreteCommand (Comando Concreto):** Una classe che implementa l'interfaccia Command, definendo l'associazione tra un'azione e il suo ricevitore. Implementa il metodo execute invocando le operazioni appropriate del ricevitore.
3. **Invoker (Invocatore):** Un oggetto che richiama il metodo execute sul comando.
4. **Receiver (Ricevitore):** La classe che conosce come eseguire le operazioni associate a una richiesta. Qualsiasi classe può fungere da ricevitore.

7.3.2 Implementazione del Pattern Command

Nel nostro specifico scenario, il pattern Command è stato implementato per gestire l'apertura di tre finestre distinte a partire dalla finestra dell'amministratore. L'amministratore, agendo come punto di controllo centrale, può invocare l'apertura di queste finestre mediante l'interazione con pulsanti o immagini specifici. Ogni interazione richiama il comando corrispondente, che viene poi eseguito per aprire la finestra desiderata.

La struttura del sistema prevede l'esistenza di un'interfaccia Command, che viene implementata da tre diverse classi. Queste classi concretizzano il comportamento specifico per l'apertura delle rispettive finestre. Sebbene le implementazioni di tali classi possano sembrare analoghe, ognuna incapsula la logica necessaria per avviare la propria finestra corrispondente. A titolo esemplificativo, considereremo l'implementazione della classe Staff, rappresentativa delle altre due, in quanto il comportamento è sostanzialmente omogeneo:

Listing 5: Interfaccia Command

```
1 package com.restaurant.command;
2 import javafx.event.Event;
3
4 public interface Command {
5     void execute(Event event) throws Exception;
6 }
```

Listing 6: AdmToStaffCommand classe che implementa Command

```
1  package com.restaurant.command;
2
3  import javafx.event.Event;
4  import javafx.fxml.FXMLLoader;
5  import javafx.scene.Node;
6  import javafx.scene.Parent;
7  import javafx.scene.Scene;
8  import javafx.stage.Stage;
9
10 import java.util.Objects;
11
12 public class AdmToStaffCommand implements Command{
13
14     @Override
15     public void execute(Event event) throws Exception{
16         try {
17             Parent root = FXMLLoader.load(Objects.requireNonNull(
                   getClass().getResource("AdministratorStaff.fxml")));
18             Scene admStaff = new Scene(root);
19             //get stage information
20             Stage window = (Stage)((Node)event.getSource()).getScene().
                   getWindow();
21             window.setScene(admStaff);
22             window.show();
23         }
```

```
24          catch (Exception e) {
25              System.out.println("Error occurred while opening Staff page
                    ");
26              throw e;
27          }
28      }
29  }
```

Listing 7: CommandInvoker

```
1  package com.restaurant.command;
2  import javafx.event.Event;
3
4  public class CommandInvoker {
5      private Command command;
6
7      public void setCommand(Command command) {
8          this.command = command;
9      }
10
11     public void executeCommand(Event event) throws Exception {
12         if (command != null) {
13             command.execute(event);
14         }
15     }
16 }
```

Listing 8: Frammento di AdministratorController dove viene utilizzato Command

```
1  public class AdministratorController {
2      private final CommandInvoker commandInvoker = new CommandInvoker();
3
4      public void admStaff(MouseEvent event) throws Exception {
5          commandInvoker.setCommand(new AdmToStaffCommand());
6          commandInvoker.executeCommand(event);
7
8      }
9      public void admStaffButton(ActionEvent event) throws Exception {
10         commandInvoker.setCommand(new AdmToStaffCommand());
11         commandInvoker.executeCommand(event);
12     }
13 }
```

7.3.3 Vantaggi dell'Uso del Pattern Command

L'adozione del pattern Command ha apportato numerosi benefici al nostro progetto:

- **Modularità:** Ha facilitato la modularità del codice, permettendo di aggiungere nuovi comandi senza modificare il codice esistente.
- **Flessibilità:** Ha offerto un meccanismo flessibile per eseguire, annullare e ripetere operazioni, migliorando la manutenibilità e l'estensibilità del sistema.

- **Disaccoppiamento:** Ha promosso il disaccoppiamento tra gli invocatori e i ricevitori delle operazioni, consentendo una maggiore indipendenza tra le componenti.
- **Registrazione e Coda delle Operazioni:** Ha permesso di registrare e mettere in coda le operazioni, fornendo un maggiore controllo sul flusso delle operazioni.

7.4 Strategy

7.4.1 Informazioni Generali su Strategy

Il design pattern Strategy è stato adottato per definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Questo pattern consente di selezionare l'algoritmo da utilizzare in modo dinamico, migliorando la flessibilità e la manutenibilità del codice. L'uso del pattern Strategy è particolarmente utile quando si ha la necessità di variare il comportamento di un algoritmo senza modificare il contesto che lo utilizza. I componenti principali del pattern sono:

1. **Strategy (Strategia):** Un'interfaccia comune a tutti gli algoritmi, che definisce un metodo da implementare.
2. **ConcreteStrategy (Strategia Concreta):** Una classe che implementa l'interfaccia Strategy, fornendo una specifica implementazione dell'algoritmo.
3. **Context (Contesto):** La classe che utilizza l'oggetto Strategy per eseguire l'algoritmo. Il Context mantiene un riferimento a un oggetto Strategy e delega l'esecuzione dell'algoritmo a questo oggetto.

7.4.2 Implementazione del Pattern Strategy nel Progetto

L'interfaccia **PaymentStrategy** funge da contratto per le diverse modalità di pagamento, definendo il metodo calculateTotalPrice, che verrà implementato da classi concrete (queste classi concrete sono organizzate e mantenute all'interno di una classe contenitore chiamata ConcreteStrategy, garantendo un'organizzazione coerente e facilitando la gestione centralizzata dei vari metodi di pagamento.) corrispondenti ai tre metodi di pagamento accettati:

1. Carta
2. Buono Pasto
3. Contanti

Ogni classe concreta implementa l'interfaccia PaymentStrategy, sovrascrivendo il metodo *calculateTotalPrice* e applicando uno sconto specifico in base al metodo di pagamento scelto.

Una volta completato l'ordine, il cliente viene indirizzato alla schermata di pagamento. Qui, il cliente ha la possibilità di scegliere il metodo di pagamento desiderato tramite la selezione del rispettivo pulsante. Questo evento attiva un controller che coordina il

processo di calcolo del prezzo totale da pagare. Il controller istanzia un oggetto *PaymentContext*, associando ad esso la strategia di pagamento selezionata dal cliente.

Listing 9: Interfaccia PaymentStrategy

```
interface PaymentStrategy {
    double calculatePayment(double amount);
}
```

Listing 10: Classe PaymentContext

```
class PaymentContext {
    private final PaymentStrategy paymentStrategy;

    public PaymentContext(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public double calculatePaymentAmount(double paymentAmount) {
        return paymentStrategy.calculatePayment(paymentAmount);
    }
}
```

Listing 11: Classe ConcreteStrategy

```
class CashPaymentStrategy implements PaymentStrategy {
    @Override
    public double calculatePayment(double amount) {
        return amount;
    }
}

class CreditCardPaymentStrategy implements PaymentStrategy {
    @Override
    public double calculatePayment(double amount) {
        return amount * 0.95;
    }
}

class MealVoucherStrategy implements PaymentStrategy {
    @Override
    public double calculatePayment(double amount) {
        return amount * 0.80;
    }
}
```

Listing 12: Classe ConcreteStrategy

```
public class PaymentController {

    @FXML
    private Label PayAmount;
    private double totalAmount;

    @FXML
    public void initialize() {
```

```
9           ConfigurationManager configurationManager =
                ConfigurationManager.getInstance();
10          totalAmount = configurationManager.getPrice();
11          PayAmount.setText(Double.toString(totalAmount));
12      }
13
14      @FXML
15      public void CashPay(){
16          PaymentContext cashPayment = new PaymentContext(new
                CashPaymentStrategy());
17          double totalAmountTmp = cashPayment.calculatePaymentAmount(
                totalAmount);
18          Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
19          alert.setTitle("Confirm Payment");
20          alert.setHeaderText("You Payed with Cash and have " +
                totalAmountTmp + " to pay!");
21          alert.setContentText("Press OK to pay");
22
23          Optional<ButtonType> result = alert.showAndWait();
24          if (result.isPresent() && result.get() == ButtonType.OK) {
25              Stage stage = (Stage) PayAmount.getScene().getWindow();
26              stage.close();
27          }
28      }
29
30      @FXML
31      public void CardPay(){
32          PaymentContext creditCardPayment  = new PaymentContext(new
                CreditCardPaymentStrategy());
33          double totalAmountTmp = creditCardPayment .
                calculatePaymentAmount(totalAmount);
34          Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
35          alert.setTitle("Confirm Payment");
36          alert.setHeaderText("You Payed with Card with 5% discount and
                have " + totalAmountTmp + " to pay!");
37          alert.setContentText("Press OK to pay");
38          Optional<ButtonType> result = alert.showAndWait();
39          if (result.isPresent() && result.get() == ButtonType.OK) {
40              Stage stage = (Stage) PayAmount.getScene().getWindow();
41              stage.close();
42          }
43      }
44
45      @FXML
46      public void MealVoucherPay(){
47          PaymentContext mealVoucherPayment  = new PaymentContext(new
                MealVoucherStrategy());
48          double totalAmountTmp = mealVoucherPayment .
                calculatePaymentAmount(totalAmount);
49          Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
50          alert.setTitle("Confirm Payment");
51          alert.setHeaderText("You Payed with MealVoucher with 20%
                discount and have " + totalAmountTmp + " to pay!");
52          alert.setContentText("Press OK to pay");
53
54          Optional<ButtonType> result = alert.showAndWait();
```

```
55          if (result.isPresent() && result.get() == ButtonType.OK) {
56              Stage stage = (Stage) PayAmount.getScene().getWindow();
57              stage.close();
58          }
59      }
60  }
```

7.4.3 Vantaggi dell'Uso del Pattern Strategy

L'adozione del pattern Strategy ha offerto numerosi benefici al nostro progetto:

- **Flessibilità:** Ha permesso di aggiungere facilmente nuovi metodi di pagamento senza modificare il codice esistente, migliorando la flessibilità e la manutenibilità del sistema.
- **Chiarezza del Codice:** Ha migliorato la chiarezza del codice, separando la logica di pagamento dal contesto dell'ordine.
- **Riutilizzabilità:** Ha promosso la riutilizzabilità del codice, consentendo di riutilizzare le implementazioni delle strategie in diversi contesti.
- **Semplicità nell'Estensione:** Ha semplificato l'estensione del sistema, permettendo di aggiungere nuove strategie di pagamento implementando semplicemente l'interfaccia PaymentStrategy.

7.5 DAO

7.5.1 Informazioni generali su DAO

Il design pattern Data Access Object (DAO) è stato implementato per separare la logica di accesso ai dati dal resto dell'applicazione, promuovendo una gestione ordinata e modulare dell'interazione con il database. Questo pattern è cruciale per mantenere un codice pulito, organizzato e facilmente manutenibile, permettendo di isolare la logica di persistenza dei dati dalle logiche di business. Il design pattern DAO fornisce un'interfaccia astratta per eseguire operazioni CRUD (Create, Read, Update, Delete) su un data source, che può essere un database, un file, un servizio web, ecc. I componenti principali del pattern DAO sono:

1. **DAO Interface (Interfaccia DAO):** Definisce i metodi standard per operazioni CRUD su un'entità specifica.
2. **DAO Implementation (Implementazione DAO):** Implementa i metodi definiti nell'interfaccia DAO, eseguendo le operazioni specifiche sul data source.
3. **Entity (Entità):** Rappresenta l'oggetto di dominio per il quale vengono eseguite le operazioni CRUD.
4. **Data Source (Fonte dei Dati):** La risorsa effettiva da cui i dati vengono recuperati e a cui vengono inviati (ad esempio, un database MySQL).

7.5.2 Implementazione del Pattern DAO nel Progetto

Il pacchetto *com.restaurant.DBUti*l include una serie di classi DAO, ciascuna responsabile della gestione di specifiche entità del dominio. Di seguito, vengono delineate alcune delle principali classi e i loro ruoli:

1. DBUtil

   - **Descrizione:** Questa classe fornisce metodi statici per la connessione e la disconnessione dal database, nonché per l'esecuzione di query SQL.
   - **Metodi Principali**
     - **dbConnect():** Stabilisce la connessione con il database utilizzando le credenziali fornite dalla classe ConfigurationManager.
     - **dbDisconnect**(): Chiude la connessione al database se aperta.
     - **dbExecuteQuery(String sqlStm):** Esegue operazioni di inserimento, aggiornamento e cancellazione.
     - **dbExecute(String sqlQuery):** Esegue query di selezione e restituisce un ResultSet.

Listing 13: Classe DBUtil

```
1  public class DBUtil {
2
3      private static Stage stage;
4      private static Connection con;
5
6      //database connection method
7      public static void dbConnect() throws SQLException,
           ClassNotFoundException  {
8
9          ConfigurationManager configurationManager =
               ConfigurationManager.getInstance();
10         try {
11             Class.forName("com.mysql.cj.jdbc.Driver");
12         }
13         catch (ClassNotFoundException e) {
14             throw new ClassNotFoundException(e.getMessage());
15         }
16         try{
17             con = DriverManager.getConnection(configurationManager
                   .getDbUrl(), configurationManager.getDbUsername(),
                   configurationManager.getDbPassword());
18         }
19         catch (SQLException e) {
20             System.out.println("Connection Failed");
21             throw e;
22         }
23
24     }
25
26     //disconnecting database method
27     public static void dbDisconnect() throws SQLException {
28         try{
```

```
29              if(con != null && !con.isClosed()){
30                  con.close();
31              }
32          }
33          catch (SQLException e) {
34              throw new SQLException(e.getMessage());
35          }
36      }
37      //for insert update delete from db
38      public static void dbExecuteQuery(String sqlStm) throws
            SQLException {
39          Statement stm = null;
40          try{
41              dbConnect();
42              stm = con.createStatement();
43              stm.executeUpdate(sqlStm);
44          }
45          catch (SQLException | ClassNotFoundException e){
46              System.out.println("Problem occurred at dbExecuteQuery
                    ");
47              throw new SQLException(e.getMessage());
48          }
49          finally {
50              if(stm != null){
51                  stm.close();
52              }
53              System.out.println("Connect Disconnected");
54              dbDisconnect();
55          }
56      }
57      //for getting data from database(selects)
58      public static ResultSet dbExecute(String sqlQuery) throws
            ClassNotFoundException, SQLException {
59          Statement stmt;
60          ResultSet rs;
61          try {
62
63              dbConnect();
64              stmt = con.createStatement();
65              rs = stmt.executeQuery(sqlQuery);
66
67          } catch (SQLException e) {
68              System.out.println("Problem occurred in dbExecute " +
                    e);
69              throw e;
70          }
71          return rs;
72      }
73
74  }
```

2. LoginDAO

   - **Descrizione:** Questa classe gestisce le operazioni di autenticazione degli utenti.

- **Metodi Principali**
  - **isLogin(String username, String password):** Verifica se le credenziali dell'utente sono valide.

Listing 14: Classe LoginDAO

```
public class LoginDAO {

    public static boolean isLogin(String username, String password
        ) throws SQLException {
        String sql = "select * from utente where username='"+
            username+"' and password='"+password+"'";
        try{
            ResultSet rs = DBUtil.dbExecute(sql);

            return rs.next();
        }
        catch (Exception e){
            return false;
        }
    }
}
```

3. StaffDAO

- **Descrizione:** Questa classe gestisce le operazioni relative allo staff.
- **Metodi Principali**
  - **insertStaffData(int id, String name, String gender, int age, String role, String salary, String date):** Inserisce un nuovo membro dello staff.
  - **updateStaffData(int id, String name, String gender, int age, String role, String salary, String date):** Aggiorna i dati di un membro dello staff esistente.
  - **deleteStaffByID(int id):** Elimina un membro dello staff tramite ID.
  - **getAllRecords():** Recupera tutti i membri dello staff.
  - **searchStaffData(String id):** Cerca un membro dello staff specifico tramite ID.
  - **sumSalaryData():** Calcola il totale degli stipendi dello staff.

Listing 15: Classe StaffDAO

```
public class StaffDAO {

    public static void insertStaffData(int id, String name, String
        gender, int age, String role, String salary, String date)
        throws SQLException {
        String sql = "insert into staff values("+id+",'"+name+"','
            "+gender+"',"+age+",'"+role+"','"+salary+"','"+date+"')
            ";
        try {
            DBUtil.dbExecuteQuery(sql);
        }
```

```
8           catch (SQLException e) {
9               System.out.println("Exception occurred while inserting
                    the staff");
10              throw e;
11          }
12      }
13
14      public static void updateStaffData(int id, String name, String
            gender, int age, String role, String salary, String date)
            throws SQLException {
15          String sql = "UPDATE staff SET name='" + name + "', gender
                ='" + gender + "', age=" + age + ", role='" + role + "
                ', salary='" + salary + "', date='" + date + "' WHERE
                id=" + id;
16          try{
17              DBUtil.dbExecuteQuery(sql);
18          }
19          catch (SQLException e) {
20              System.out.println("Exception occurred while updating
                    the staff");
21              throw e;
22          }
23      }
24
25      public static void deleteStaffByID(int id) throws SQLException
            {
26          String sql = "delete from staff where id="+id;
27          try{
28              DBUtil.dbExecuteQuery(sql);
29          }
30          catch (SQLException e) {
31              System.out.println("Exception occurred while deleting
                    the Staff Record");
32              throw e;
33          }
34      }
35      public static ObservableList<Staff> getAllRecords() throws
            SQLException, ClassNotFoundException {
36          String sql = "select * from staff";
37
38          try {
39              ResultSet rs = DBUtil.dbExecute(sql);
40              ObservableList<Staff> staffList;
41              staffList = getStaffObjects(rs);
42              return staffList;
43          }
44          catch (SQLException e) {
45              System.out.println("Error occurred while fetching
                    staff records");
46              throw e;
47          }
48      }
49
50      public static ObservableList<Staff> getStaffObjects(ResultSet
            rs) throws SQLException, ClassNotFoundException {
51          try
```

```
52          {
53              ObservableList<Staff> staffList = FXCollections.
                    observableArrayList();
54              while (rs.next())
55              {
56                  Staff staff = new Staff();
57                  staff.setId(rs.getInt("id"));
58                  staff.setName(rs.getString("name"));
59                  staff.setGender(rs.getString("gender"));
60                  staff.setAge(rs.getInt("age"));
61                  staff.setRole(rs.getString("role"));
62                  staff.setSalary(rs.getString("salary"));
63                  staff.setDate(rs.getString("date"));
64                  staffList.add(staff);
65              }
66              return staffList;
67          }
68          catch (SQLException e){
69              System.out.println("Error occurred while fetching
                    staff records");
70              throw e;
71          }
72      }
73
74      public static ObservableList<Staff> searchStaffData(String id)
            throws SQLException, ClassNotFoundException {
75          String sql = "select * from staff where id="+id;
76          try{
77              ResultSet rs = DBUtil.dbExecute(sql);
78              ObservableList<Staff> staffList;
79              staffList = getStaffObjects(rs);
80              return staffList;
81          }
82          catch (SQLException e){
83              System.out.println("Error occurred while searching
                    staff records");
84              throw e;
85          }
86      }
87
88      public static double sumSalaryData() throws SQLException,
            ClassNotFoundException {
89          String sql = "select sum(salary) as totalSalary from staff
                ";
90          double totalSalary = 0;
91          try{
92              ResultSet rs = DBUtil.dbExecute(sql);
93              if (rs.next()){
94                  totalSalary = rs.getDouble("totalSalary");
95              }
96              return totalSalary;
97          }
98          catch (SQLException e) {
99              System.out.println("Error occurred while calculating
                    total salary");
100             throw e;
```

```
101                 }
102             }
103 }
```

Si omettono per compattezza DishDAO e OrdersDAO in quanto presentano la stessa struttura di StaffDAO, con l'unica differenza di non presentare alcuni metodi.

7.5.3 Vantaggi dell'Uso del Pattern DAO

L'adozione del pattern DAO ha offerto numerosi benefici al nostro progetto:

- **Uso di Interfacce:** Abbiamo definito interfacce DAO per ogni entità, garantendo flessibilità e facilitando il testing.
- **Gestione delle Connessioni:** Abbiamo implementato un Singleton per gestire la connessione al database, assicurando un utilizzo efficiente delle risorse.
- **Gestione delle Eccezioni:** Abbiamo gestito adeguatamente le eccezioni SQL, assicurando che eventuali errori vengano catturati e gestiti in modo appropriato.

Capitolo IV

Unit Test

8 Unit Testing

Per il nostro progetto, abbiamo utilizzato JUnit 5 per implementare i test unitari delle funzionalità principali. Per i test, abbiamo importato le seguenti librerie:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;
```

Le asserzioni principali utilizzate per gestire i test sono state:
`assertEquals`, `assertFalse`, `assertTrue`, `assertNotNull` e `assertNull`. È possibile trovare tutta la parte di codice riguardante l'unit testing nella cartella
`src/test`.

8.1 Test delle Operazioni CRUD

I test delle operazioni CRUD sono stati condotti per garantire che ogni funzione operasse correttamente con il database. In particolare, sono stati implementati i seguenti test:

- **Create (C)**: Verifica che i nuovi record possano essere creati correttamente nel database.
- **Read (R)**: Assicura che i dati possano essere letti correttamente dal database.
- **Update (U)**: Controlla che i dati esistenti possano essere aggiornati senza errori.
- **Delete (D)**: Conferma che i record possano essere eliminati correttamente dal database.

Questi test sono stati applicati ai moduli `orders`, `staff`, `dish` e `login`, garantendo che tutte le operazioni di gestione dei dati funzionassero come previsto.

8.2 Test della Connessione al Database

Oltre ai test CRUD, abbiamo implementato test per verificare la connessione e la disconnessione al database, nonché l'esecuzione delle query. In particolare:

- **Connessione al Database**: Verifica che il sistema sia in grado di stabilire correttamente una connessione al database.

- **Disconnessione dal Database**: Assicura che la connessione al database possa essere terminata correttamente.
- **Esecuzione di Query**: Controlla che le query SQL possano essere eseguite correttamente e che i risultati siano quelli attesi.

Questi test hanno garantito che le operazioni di interazione con il database fossero affidabili e robuste.

8.3 Test delle Funzionalità di Pagamento

Abbiamo implementato test specifici per la funzionalità di pagamento. Questi test avevano l'obiettivo di verificare che le transazioni di pagamento fossero eseguite correttamente e senza errori. Sono stati testati vari scenari di pagamento, inclusi:

- Pagamenti con successo.
- Pagamenti falliti a causa di fondi insufficienti.
- Gestione delle eccezioni durante il processo di pagamento.

I test delle funzionalità di pagamento hanno assicurato che il sistema fosse robusto e in grado di gestire correttamente tutte le situazioni previste durante una transazione.

8.4 Conclusione

L'uso di JUnit 5 per i test unitari ha fornito una solida base per la verifica delle funzionalità del progetto. Grazie a questi test, è stato possibile individuare e correggere tempestivamente eventuali bug, garantendo la qualità e l'affidabilità del software sviluppato.

Capitolo V

DataBase

9 Introduzione

Per la strutturazione e l'implementazione del DataBase destinato ad essere integrato all'interno della nostra applicazione software, abbiamo optato per l'adozione di **MySQL**. L'impiego di un sistema di gestione di database relazionale si è rivelato estremamente vantaggioso, non soltanto per quanto concerne l'archiviazione sicura e ordinata dei dati, ma anche per la possibilità di interagire dinamicamente con essi. Questo approccio ha permesso un'efficiente trasmissione bidirezionale delle informazioni tra il cliente e l'amministratore del sistema, facilitando la gestione operativa e amministrativa dell'intera applicazione.

Inoltre, ci ha consentito di mettere in pratica in modo concreto e pragmatico le nozioni apprese durante il corso accademico di Basi di Dati. Ciò ha rappresentato un'opportunità preziosa per confrontarci con scenari reali di sviluppo software, rafforzando così la nostra comprensione teorica attraverso l'applicazione pratica. Questo esercizio non solo ha consolidato le nostre competenze tecniche nel campo della gestione dei database, ma ha anche migliorato la nostra capacità di progettare e sviluppare soluzioni software robuste ed efficienti, in linea con le migliori pratiche del settore.

10 Tabelle

	username	password
1	admin	admin
2	marco	123

Figura 16: Tabella degli Utenti

	id	name	gender	age	role	salary	date
1	1	Carlo Magno	Male	23	Chef	1800	2024-05-07
2	2	Vinicio Capossela	Male	30	Waiter	1400	2024-05-07
3	3	Zooey Deschanel	Female	29	Waitress	1400	2024-05-08
4	4	Ryan Gosling	Male	40	Chef	1800	2024-05-09

Figura 17: Tabella dello Staff

	Id	Dish	Course	Price	Date
1	1	Antipasto Toscano	Appetizer	14	2024-04-30
2	2	Carbonara	First Course	121	2024-05-08
3	3	Bistecca Fiorentina	Second Course	21	2024-05-01
4	4	Tiramisù	Dessert	123	2024-04-30
5	5	Naturale Minerale	Water	12	2024-05-02
6	6	Chianti Classico	Wine	1	2024-05-01
7	7	Lasagna	First Course	13	2024-05-08

Figura 18: Tabella del Menu

	id	name	price	date
1	2	marco	14	11/11/11
2	3	marco	154	11/11/11
3	4	marco	145	11/11/11
4	5	marco	26	17/05/24
5	6	marco	26	26/07/24
6	7	marco	28	26/07/24
7	8	marco	26	05/08/24

Figura 19: Tabella degli Ordini

Elenco delle figure