UNIVERSITÀ DEGLI STUDI DI NAPOLI
"PARTHENOPE"

Facoltà di Scienze e Tecnologie
Corso di Laurea in Informatica

Relazione di Progetto
Corso di Programmazione III

Gestionale Ristorante in JavaFX

Studente:
Cristian Fabozzo
Mat. 0124003122

Anno Accademico 2025/2026

Indice

1 Scopo e Requisiti del Progetto

Il progetto ha come obiettivo la realizzazione di un'applicazione Desktop per la gestione integrata dei flussi operativi di un ristorante. L'applicazione è progettata per semplificare la comunicazione tra la sala (presa delle comande) e la cucina, nonché per gestire la fase di pagamento e chiusura del conto.

1.1 Funzionalità Principali

Il sistema permette di gestire il ciclo di vita completo di un ordine attraverso un'interfaccia grafica intuitiva:

- **Gestione Ordini** I camerieri possono creare nuove comande e inviare l'ordine alla cucina.
- **Gestione Cucina:** Il personale di cucina visualizza gli ordini da preparare
- **Gestione Pagamenti** Il sistema calcola automaticamente il totale dell'ordine e permette di registrare il pagamento tramite diverse modalità (Contanti, Carta, ecc.).
- **Persistenza Dati:** Tutti i dati relativi a ordini, dettagli e stati vengono salvati in modo permanente su database.

2 Tecnologie Utilizzate

Per garantire portabilità, efficienza e una chiara separazione delle responsabilità, sono state adottate le seguenti tecnologie:

- **Linguaggio:** Java (JDK 21).
- **Interfaccia Grafica (GUI):** JavaFX, con utilizzo di file FXML per la definizione dichiarativa del layout.
- **Gestione Dipendenze:** Apache Maven.
- **Database:** SQLite. È stato scelto per la sua natura "serverless": il database è un semplice file (`db_ristorante.db`) incluso nel progetto, garantendo la massima portabilità senza configurazioni esterne.
- **Librerie di Supporto:** JDBC (driver SQLite) per la connessione al database.

3 Architettura e Struttura del Progetto

L'applicazione segue rigorosamente il pattern architetturale **MVC (Model-View-Controller)**, che separa la logica di business dalla presentazione dei dati.

- **Model** Contiene le classi POJO (es. `RestaurantOrder`, `OrderDetail`) e la logica di accesso ai dati (DAO).
- **View** Costituita dai file `.fxml` che definiscono la struttura grafica delle schermate (Login, Tavoli, Cucina).
- **Controller** Classi Java che gestiscono l'interazione utente. Ricevono gli input dalla View, invocano la logica del Model e aggiornano l'interfaccia (es. `PaymentController`, `KitchenController`).

4 Design Patterns (GoF) Implementati

Per risolvere problemi ricorrenti di progettazione e rendere il codice estensibile, sono stati implementati i seguenti Design Pattern.

4.1 Strategy Pattern (Gestione Pagamenti)

Utilizzato nel modulo dei pagamenti per gestire algoritmi diversi in modo intercambiabile. Invece di usare lunghi blocchi `if-else` per controllare il tipo di pagamento, il comportamento è incapsulato in classi specifiche.

Implementazione:

- Interfaccia `PaymentMethod` con metodo `pay()`.
- Classi concrete: `Cash`, `Card`, `Atm`.
- Il `PaymentController` inietta la strategia scelta nel Service al momento del click.

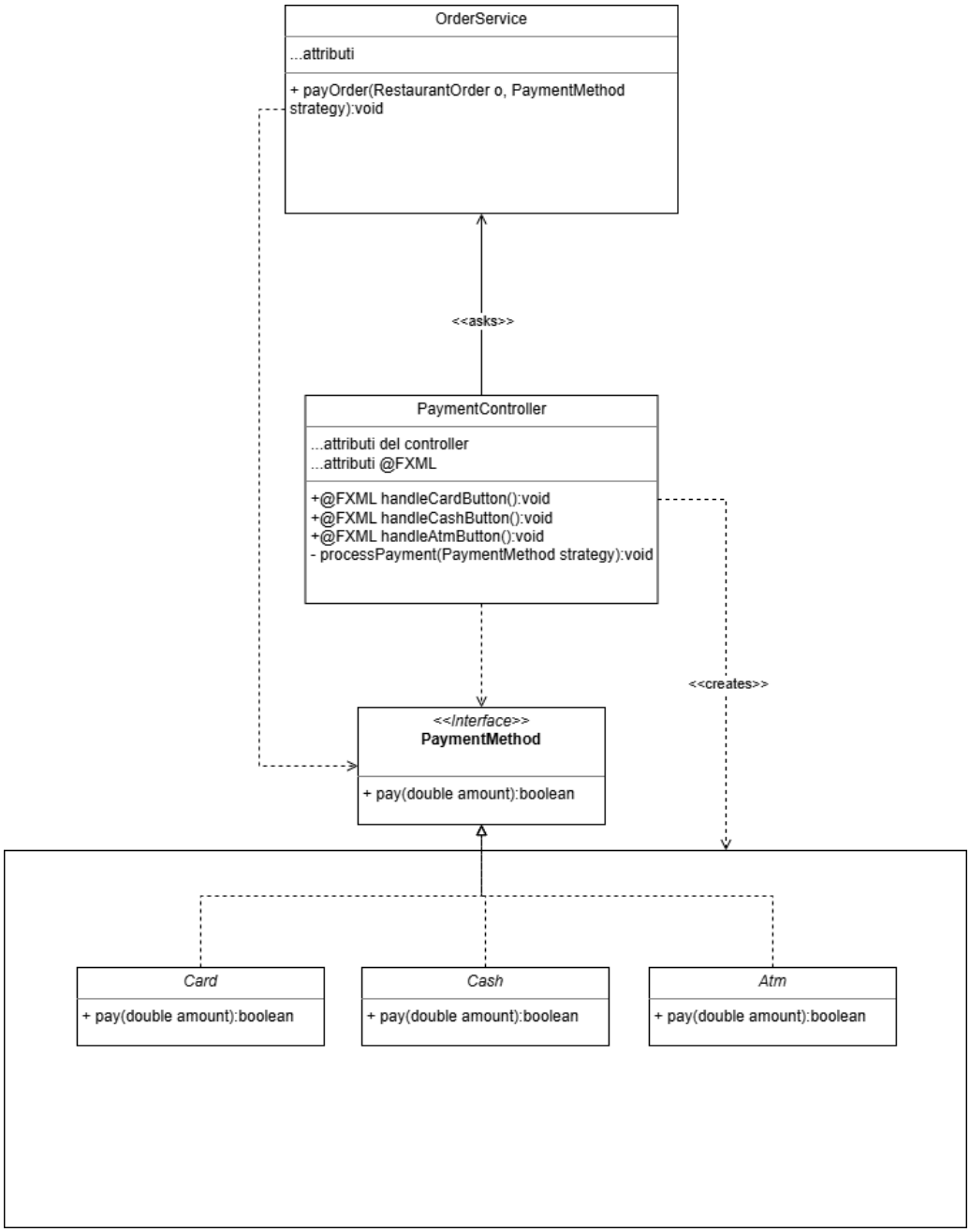

Figura 1: Diagramma UML Strategy Pattern

```
1
2  //la strategy si occupa solo del pagamento
3  //tipo: controllare se la carta e' valida, calcolare il resto...
4
5  public interface PaymentMethod {
6      //ogni strategy concreta potra' implementare il metodo di pagamento con
       la propria logica
7      //per ora visualizzeranno in modo diverso solo la stampa
8      public boolean pay(double amount);
9  }
10
11
12
13 public class Atm implements PaymentMethod{
14     @Override
15     public boolean pay(double amount) {
```

```
16          System.out.println("Hai scelto pagamento con bancomat/atm");
17          if(amount < 0){
18              return false;
19          }
20          return true;
21      }
22      @Override
23      public String toString(){
24          return "ATM";
25      }
26
27  }
28
29
30  public class Cash implements PaymentMethod{
31
32      @Override
33      public boolean pay(double amount) {
34          System.out.println("Hai scelto pagamento con contanti");
35          if(amount < 0){
36              return false;
37          }
38          return true;
39      }
40
41      @Override
42      public String toString(){
43          return "CASH";
44      }
45  }
46
47
48  public class Card implements PaymentMethod{
49
50      @Override
51      public boolean pay(double amount) {
52          System.out.println("Hai scelto pagamento con carta");
53          if(amount < 0){
54              return false;
55          }
56          return true;
57      }
58      @Override
59      public String toString(){
60          return "CARD";
61      }
62  }
63
64
65  public class OrderService{
66  //...attributi OrderService
67
68  //logica per il pagamento dell'ordine
69      //il service si interpone fra strategy e state
70      //ha un ruolo principale
71      //dice: con quale strategia devo pagare
72      //      in quale state cambiare
73      //      e quando devo aggiornare (in caso di success) il metodo di
        pagamento dell'ordine
74
```

```
75  public void payOrder(RestaurantOrder o, PaymentMethod strategy) throws
      PaymentException{
76          //Prendo il totale dell'ordine
77          double amount = o.getTotalPrice();
78          System.out.println("Il cliente ha pagato" + o.getTotalPrice());
79          //Eseguo la strategy, eseternamente viene passato l'oggetto scelto
80          //quindi c'e' un constructor injection
81          boolean success = strategy.pay(amount);
82
83          //se il pagamento ha successo allora eseguo le seguenti operazioni
84          if(success){
85              System.out.println("Pagamento riuscito con successo");
86
87              System.out.println("Cambiando sul db lo stato dell'ordine in
      PAID");
88              o.payOrder();
89              System.out.println("Aggiornando il db");
90              //aggiorno lo stato del sistema
91              o.setPaymentMethod(org.ristorante.model.PaymentMethod.valueOf(
      strategy.toString()));
92              dao.update(o);
93          }else{
94              //lancia un errore per i pagamenti
95              throw new PaymentException("errore pagamenti");
96          }
97
98       }
99
100 //...altri metodi OrderService
101 }
102
103
104 private void processPayment(PaymentMethod strategy) {
105         // Recupero l'ordine selezionato dal componente figlio
106         RestaurantOrder selected = orderListController.getSelectedOrder();
107
108         if (selected == null) {
109             super.showError("Seleziona un ordine da pagare!");
110             return;
111         }
112
113         try {
114             // eseguo il pagamento
115             service.payOrder(selected, strategy);
116
117             //si potrebbe utilizzare un metodo in base controller per il
      feedback positivo
118             //super.showInfo("Pagamento riuscito con " + strategy.
      getMethodName());
119
120             // ricarico la lista per far sparire l'ordine appena pagato
121             loadOrdersToPay();
122
123         } catch (PaymentException e) {
124             // Gestione errore specifica del pagamento
125             super.showError("Errore durante il pagamento: " + e.getMessage()
      );
126         } catch (Exception e) {
127             // Gestione errori generici
128             super.showError("Errore di sistema: " + e.getMessage());
```

```
129             }
130         }
```

4.2 State Pattern (Ciclo di Vita Ordine)

Utilizzato per gestire le transizioni di stato di un ordine. Ogni ordine possiede uno stato interno (`current_state`) che determina quali operazioni sono permesse. Le transizioni seguono il flusso: `ORDERED` → `SERVED` → `PAID`.

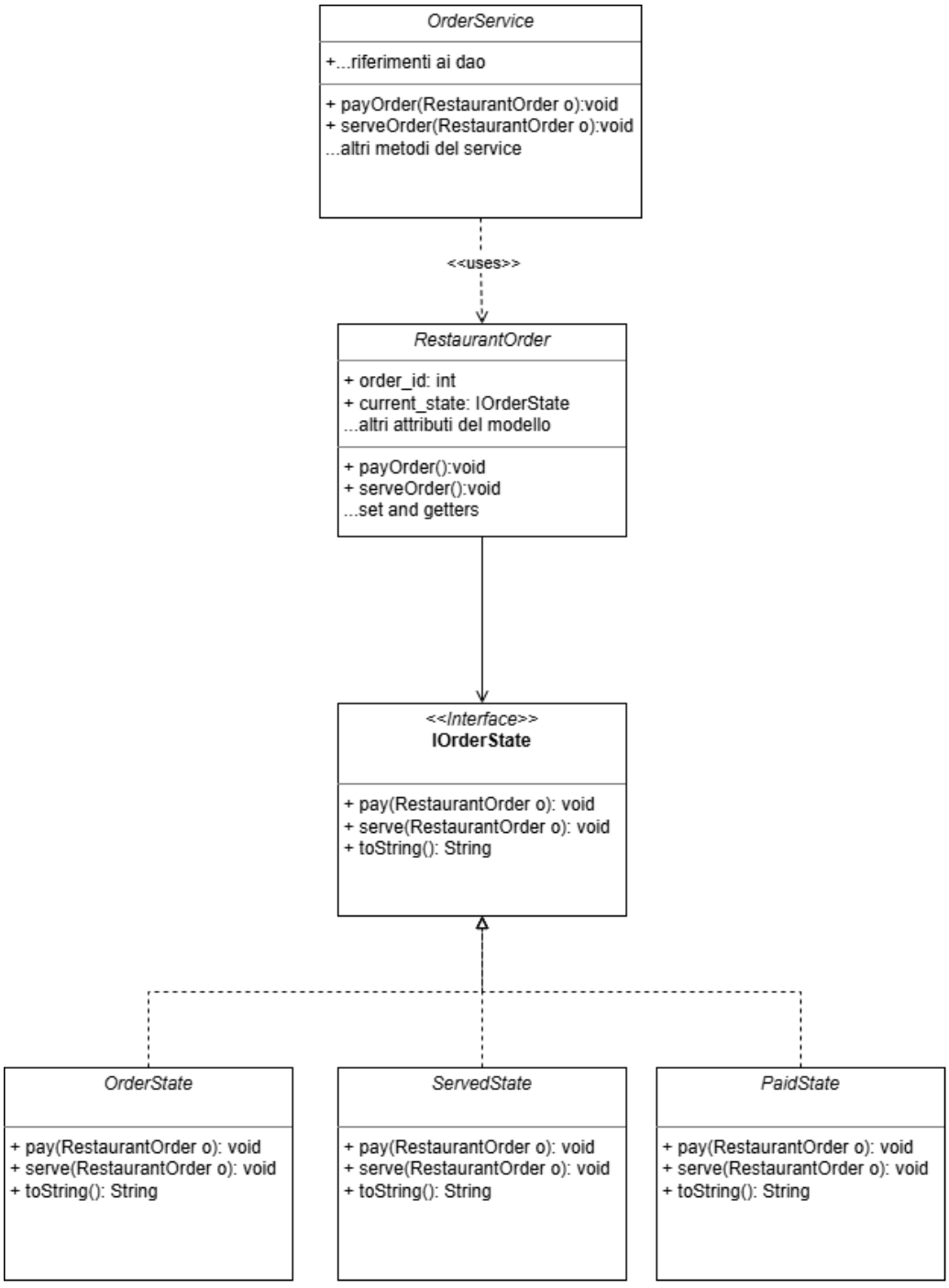

Figura 2: Diagramma UML State Pattern

```
1
2 import...
```

```
3
4  public class OrderService{
5
6    private final IRestaurantOrderDao dao;
7      private final IOrderDetailDao detaildao;
8
9      public OrderService(IRestaurantOrderDao dao, IOrderDetailDao detaildao){
10         this.dao = dao;
11         this.detaildao = detaildao;
12     }
13
14 //logica per il pagamento dell'ordine
15     //il service si interpone fra strategy e state
16     //ha un ruolo principale
17     //dice: con quale strategia devo pagare
18     //      in quale state cambiare
19     //      e quando devo aggiornare (in caso di success) il //metodo di
       pagamento dell'ordine
20
21   public void payOrder(RestaurantOrder o, PaymentMethod strategy) throws
       PaymentException{
22         //Prendo il totale dell'ordine
23         double amount = o.getTotalPrice();
24         System.out.println("Il cliente ha pagato" + o.getTotalPrice());
25         //Eseguo la strategy, eseternamente viene passato //l'oggetto scelto
26         //quindi c'e' un injection
27         boolean success = strategy.pay(amount);
28
29         //se il pagamento ha successo allora eseguo le seguenti operazioni
30         if(success){
31             System.out.println("Pagamento riuscito con successo");
32
33             System.out.println("Cambiando sul db lo stato dell'ordine in
       PAID");
34             o.payOrder();
35             System.out.println("Aggiornando il db");
36             //aggiorno lo stato del sistema
37             o.setPaymentMethod(org.ristorante.model.PaymentMethod.valueOf(
       strategy.toString()));
38             dao.update(o);
39         }else{
40             //lancia un errore per i pagamenti
41             throw new PaymentException("errore pagamenti");
42         }
43
44     }
45
46 //segna come SERVED l'ordine
47     public void serveOrder(RestaurantOrder o){
48         o.serveOrder();
49         System.out.println("Cambiando sul db lo stato dell'ordine in SERVED
       ");
50         dao.update(o);
51     }
52
53 //altri metodi del service
54
55 }//fine OrderService
56
57
```

```
58 //il contratto che deve rispettare ogni State
59 public interface IOrderState {
60     public String getStatusName();
61     public void pay(RestaurantOrder o);
62     public void serve(RestaurantOrder o);
63     public String toString();
64 
65     //altri metodi della classe
66 
67 }
68 
69 
70 
71 public class OrderedState implements IOrderState {
72 
73     @Override
74     public String getStatusName() {
75         return "ORDERED";
76     }
77 
78     @Override
79     public void pay(RestaurantOrder o) {
80         throw new ValidationError("Non puoi pagare un ordine se non e' stato
       servito");
81     }
82 
83     @Override
84     public void serve(RestaurantOrder o) {
85         System.out.println("Sto segnando l'ordine come SERVED");
86         o.setCurrentState(new ServedState());
87     }
88 
89     @Override
90     public String toString() {
91         return "ORDERED";
92     }
93 
94 //altri metodi della classe
95 
96 }//fine classe OrderedState
97 
98 
99 public class PaidState implements IOrderState {
100 
101     @Override
102     public String getStatusName() {
103         return "PAID";
104     }
105 
106     @Override
107     public void pay(RestaurantOrder o) {
108         throw new ValidationError("Ordine gia' pagato");
109     }
110 
111     @Override
112     public void serve(RestaurantOrder o) {
113         throw  new ValidationError("Ordine gia' serivto");
114     }
115 
116     @Override
```

```
117     public String toString() {
118         return "PAID";
119     }
120 
121 //altri metodi della classe
122 
123 }//fine PaidState
124 
125 
126 public class ServedState implements IOrderState {
127     /*@Override
128     public void next(RestaurantOrder o) {
129         o.setCurrentState(new PaidState());
130     }*/
131 
132     @Override
133     public void cancel(RestaurantOrder o) {
134         throw new ValidationError("Per sicurezza non puoi cancellare un
    ordine gia' servito");
135     }
136 
137     @Override
138     public void edit(RestaurantOrder o) {
139         throw new ValidationError("Non puoi aggiornare un ordine gia'
    servito - creane un altro");
140     }
141 
142     @Override
143     public String getStatusName() {
144         return "SERVED";
145     }
146 
147     @Override
148     public void pay(RestaurantOrder o) {
149         System.out.println("Pagamento ordine OK");
150         o.setCurrentState(new PaidState());
151     }
152 
153     @Override
154     public void serve(RestaurantOrder o) {
155         throw new ValidationError("Ordine gia' servito");
156     }
157 
158     @Override
159     public String toString() {
160         return "SERVED";
161     }
162 
163 
164 //altri metodi della classe
165 
166 }//fine classe ServedState
167 
168 
169 
170 package org.ristorante.model;
171 
172 import java.util.Date;
173 
174 public class RestaurantOrder {
```

```
175
176     //attributi...
177
178     //usiamo il pattern State per la gestione dello stato
179     private IOrderState currentState;
180
181     //passiamo this, cioe' l'oggetto su cui poi faremo setCurrentState
        quindi o.setCurrentState(new ...)
182
183
184     public void payOrder(){
185         currentState.pay(this);
186     }
187
188     public void serveOrder(){
189         currentState.serve(this);
190     }
191
192     public IOrderState getCurrentState() {
193         return currentState;
194     }
195
196     public void setCurrentState(IOrderState currentState) {
197         this.currentState = currentState;
198     }
199
200 //altri metodi della classe
201
202 }//fine classe RestaurantOrder
```

4.3 Singleton Pattern

Utilizzato per la classe `DBConnection`. Questo pattern garantisce che esista **una sola istanza** della connessione al database in tutta l'applicazione, ottimizzando le risorse e prevenendo conflitti di accesso al file SQLite.

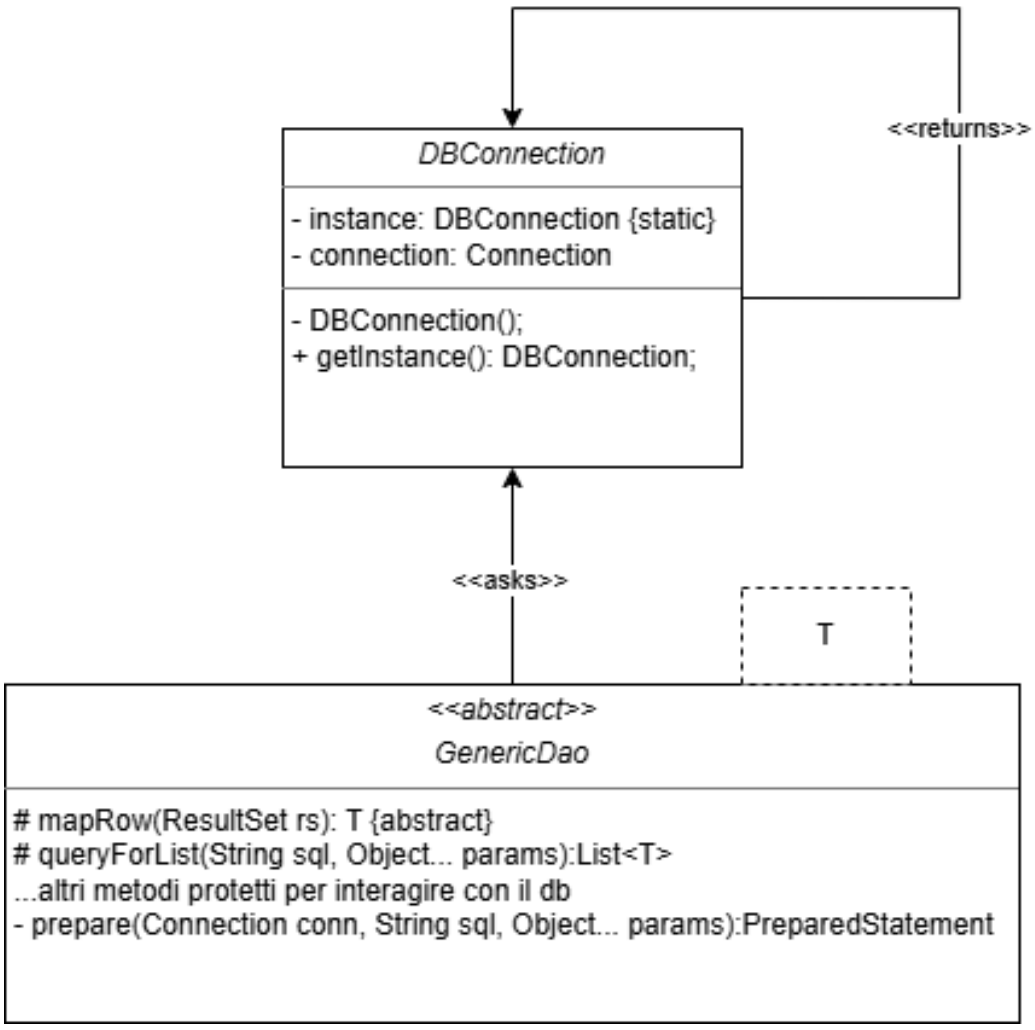

Figura 3: Diagramma UML Singleton Pattern

```
1
2  //questi moduli sono importanti per stabilire la connessione
3  //al db e poter gestire gli errori
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.SQLException;
7
8  public class DBConnection {
9      //crea un oggetto di DBConnection
10     private static DBConnection instance;
11
12     private final Connection connection;
13
14     //essendo un singleton il costruttore deve essere privato
15     //cosi non puo' essere istanziato esternamente
16     private DBConnection() throws SQLException {
17         //tenta la connessione al db
18         String url = "jdbc:sqlite:db_ristorante.sqlite";
19
20         //DriverManager e' fondamentale per creare il collegamento
21         this.connection = DriverManager.getConnection(url);
22         //connection.createStatement().execute("PRAGMA foreign_keys = ON");
23
24         System.out.println("Ok DBConnection");
25     };
```

```
26
27      //prendi l'unico oggetto disponibile
28      public static DBConnection getInstance() throws SQLException {
29          //isClosed e' un metodo ereditato
30          if(instance == null || instance.getConnection().isClosed()){
31              instance = new DBConnection();
32              System.err.println("Ok getInstance DBConnection");
33          }
34          return instance;
35      }
36
37      public Connection getConnection() {
38          return connection;
39      }
40  }
41
42      //<T>    un segnaposto: verr  sostituito da User, RestaurantTable,
    Order, ecc.
43      public abstract class GenericDao<T> {
44
45
46          //GenericDao non sa come mappare l'oggetto quindi rimane un abstract
     method
47          //a conoscenza ovviamente solo di chi eredita
48          //questo metodo serve a mappare l'oggetto dal db ad un oggetto di
    java
49          protected abstract T mapRow(ResultSet rs) throws SQLException;
50
51
52          //Serve per le Select che restituiscono molte righe (getall...)
53
54          protected List<T> queryForList(String sql, Object... params) {
55              List<T> list = new ArrayList<>();
56
57              // Il try-with-resources    fondamentale: chiude Connection e
    Statement anche se esplode tutto.
58              try (Connection conn = DBConnection.getInstance().getConnection
    ();
59                   PreparedStatement ps = prepare(conn, sql, params);
60                   ResultSet rs = ps.executeQuery()) {
61
62
63                  //chiediamo ai figli di usare il loro metodo mapRow
64                  while (rs.next()) {
65
66                      list.add(mapRow(rs));
67                  }
68
69              } catch (SQLException e) {
70                  //lanciamo l'errore dal generic
71                  throw new DaoError("Errore dal generic Dao queryForList " +
    sql, e);
72              }
73              return list;
74          }
75
76          //query che restituiscono un solo risultato
77          protected T queryForSingle(String sql, Object... params) {
78              try (Connection conn = DBConnection.getInstance().getConnection
    ();
```

```
                PreparedStatement ps = prepare(conn, sql, params);
                ResultSet rs = ps.executeQuery()) {

            // se c'e' almeno un risultato, lo mappiamo e lo restituiamo.
            if (rs.next()) {

                return mapRow(rs);
            }

        } catch (SQLException e) {
            throw new DaoError("Errore del generico Dao queryForSingle " + sql, e);
        }
        return null;
    }

    //per delete, update, insert
    //si chiama update perche' l'azione principale e' quella di usare un executeUpdate()

    protected boolean update(String sql, Object... params) {
        try (Connection conn = DBConnection.getInstance().getConnection();
             PreparedStatement ps = prepare(conn, sql, params)) {

            // executeUpdate ci dice quante righe sono state toccate
            return ps.executeUpdate() > 0;

        } catch (SQLException e) {
            throw new DaoError("GenericDao - Errore nell'update/delete: " + sql, e);
        }
    }



    // prende i parametri variabili (...) e li mette al posto giusto nei ?
    private PreparedStatement prepare(Connection conn, String sql, Object... params) throws SQLException {
        PreparedStatement ps = conn.prepareStatement(sql);
        for (int i = 0; i < params.length; i++) {
            // setObject capisce da solo se gli stai passando un int, una stringa o un double.
            ps.setObject(i + 1, params[i]);
        }
        return ps;
    }
}
```

4.4 Singleton Pattern (User Session)

Utilizzato per la classe `UserSession`. Questo pattern permette di mantenere lo stato dell'utente autenticato (ID, ruolo e permessi) in un'**unica istanza globale**, accessibile da qualsiasi controller senza la necessità di passare l'oggetto `User` come parametro.

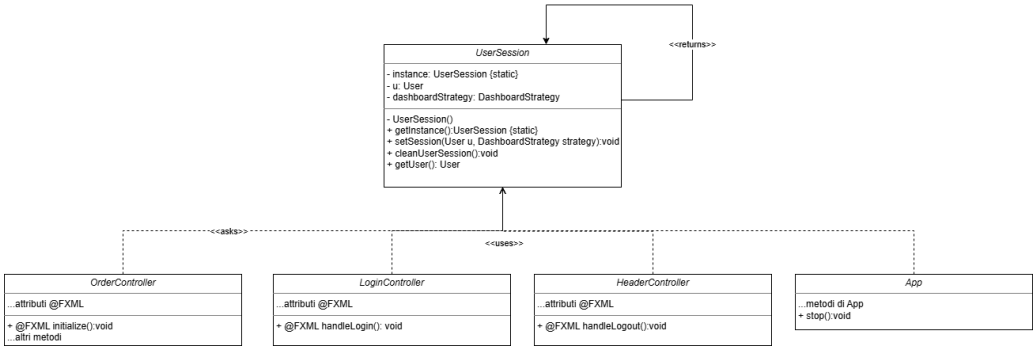

Figura 4: Diagramma UML Singleton Pattern UserSession

```

public class UserSession {
    //   un sigleton per gestire la sessione utente
    //siccome    un singleton ci assicuriamo che l'utente loggato sia sempre
    al massimo 1

    private static UserSession instance;
    private User u;
    DashboardStrategy dashboardStrategy;
    private UserSession(){};

    public static UserSession getInstance() {
        if(instance == null) {
            instance = new UserSession();
            System.out.println("Ok getInstance UserSession");
        }
        return instance;
    }

    //al login riceveremo un user passato dal AuthService
    //l'user passato sara' il nuovo user della sessione
    public void setSession(User u, DashboardStrategy strategy){
        System.out.println("Sto settando la sessione, con user_id: " + u.
   getUser_id());
        this.u = u;
    }

    //al logout puliremo la sessione
    public void cleanUserSession(){
        System.out.println("Sto pulendo la sessione...");
        this.u = null;
        this.dashboardStrategy = null;
    }

    //Questo metodo serve solo a capire chi    loggato
    public User getUser(){
        System.out.println("Sto restituendo l'utente corrente..." + u);
        return u;
    }


}

public class LoginController extends BaseController{

    //...metodi di LoginController
```

```
45
46      @FXML
47      public void handleLogin(){
48          try {
49              //pulizia degli errori precedenti
50              errorLabel.setVisible(false);
51
52              User loggedUser = authService.login(usernameField.getText(),
    passwordField.getText());
53
54              System.out.println("Settando la strategia");
55              //creo la factory e decido dove andare
56              StrategyFactory factory = new StrategyFactory();
57              DashboardStrategy strategy = factory.createStrategy(loggedUser.
    getRole());
58
59              System.out.println("Settando l'user corrente nella sessione");
60              //!!! QUI L'UTILIZZO DI UserSession !!!
61              //settiamo l'utente loggato
62              UserSession.getInstance().setSession(loggedUser, strategy);
63
64              //cambio scena con i dati forniti dallo strategy
65              System.out.println("Cambiando scena in base al ruolo dell'utente
    ");
66              super.changeScene(strategy.getFxmlPath(), strategy.getTitle());
67
68
69          }catch (DaoError e){
70              errorLabel.setText("Sistema momentaneamente non disponibile.");
71              errorLabel.setStyle("-fx-text-fill: red;");
72              errorLabel.setVisible(true);
73
74          }catch (ValidationError e){
75              errorLabel.setText(e.getMessage());
76              errorLabel.setStyle("-fx-text-fill: orange;");
77              errorLabel.setVisible(true);
78
79          }catch (Exception e){
80              errorLabel.setText("Errore sconosciuto.");
81              e.printStackTrace();
82          }
83      }
84
85  }
86
87
88  public class OrderController extends BaseController{
89
90  //...attributi di OrderController
91
92
93  @FXML public void initialize() {
94          this.invoker = new CommandInvoker();
95          this.orderService = new OrderService(new RestaurantOrderDao(), new
    OrderDetailDao());
96          this.menuItemDao = new MenuItemDao();
97          //!!! QUI L'UTILIZZO DI UserSession !!!
98          //viene usato per stabilire chi sta prendendo l'ordine
99          //dato che serve al db quando registriamo un ordine
100         this.activeUserID =
```

```
        UserSession.getInstance().getUser().getUser_id();
        // Inizializzo la lista temporanea vuota
        this.tempOrderList = FXCollections.observableArrayList();
        this.orderTableView.setItems(tempOrderList);

        loadMenu();

        columnsConfig();
    }

//...altri metodi di OrderController

 }


//questo componente riutilizzabile e' semplicissimo
//in ogni dashobard e' presente per permettere la disconnessione
//sfrutta quella che e' la userSession
public class HeaderController extends BaseController {
    //...attributi di HeaderController

    @FXML public void handleLogout(){
        //al logout puliamo la sessione
        System.out.println("Utente attuale" + UserSession.getInstance().
   getUser());
        UserSession.getInstance().cleanUserSession();
        System.out.println("disconnessione dell' utente");
        //riporta al login
        System.out.println("Riportando al login");
        SceneManager.changeScene("/org/ristorante/view/login.fxml", "Login")
   ;
    }

}


public class App extends Application {
    //...metodi di app

    @Override
    public void stop(){
        System.out.println("Sto chiudendo l'app...");
        //quando ci si scollega puliamo la sessione
        UserSession.getInstance().cleanUserSession();
    }

}
```

4.5 Command Pattern

Utilizzato per incapsulare l'invio di una comanda. Questo pattern **disaccoppia** l'oggetto che invoca l'operazione (l'interfaccia grafica) da quello che la esegue (il Service), facilitando l'estensione del sistema e la gestione di operazioni complesse.

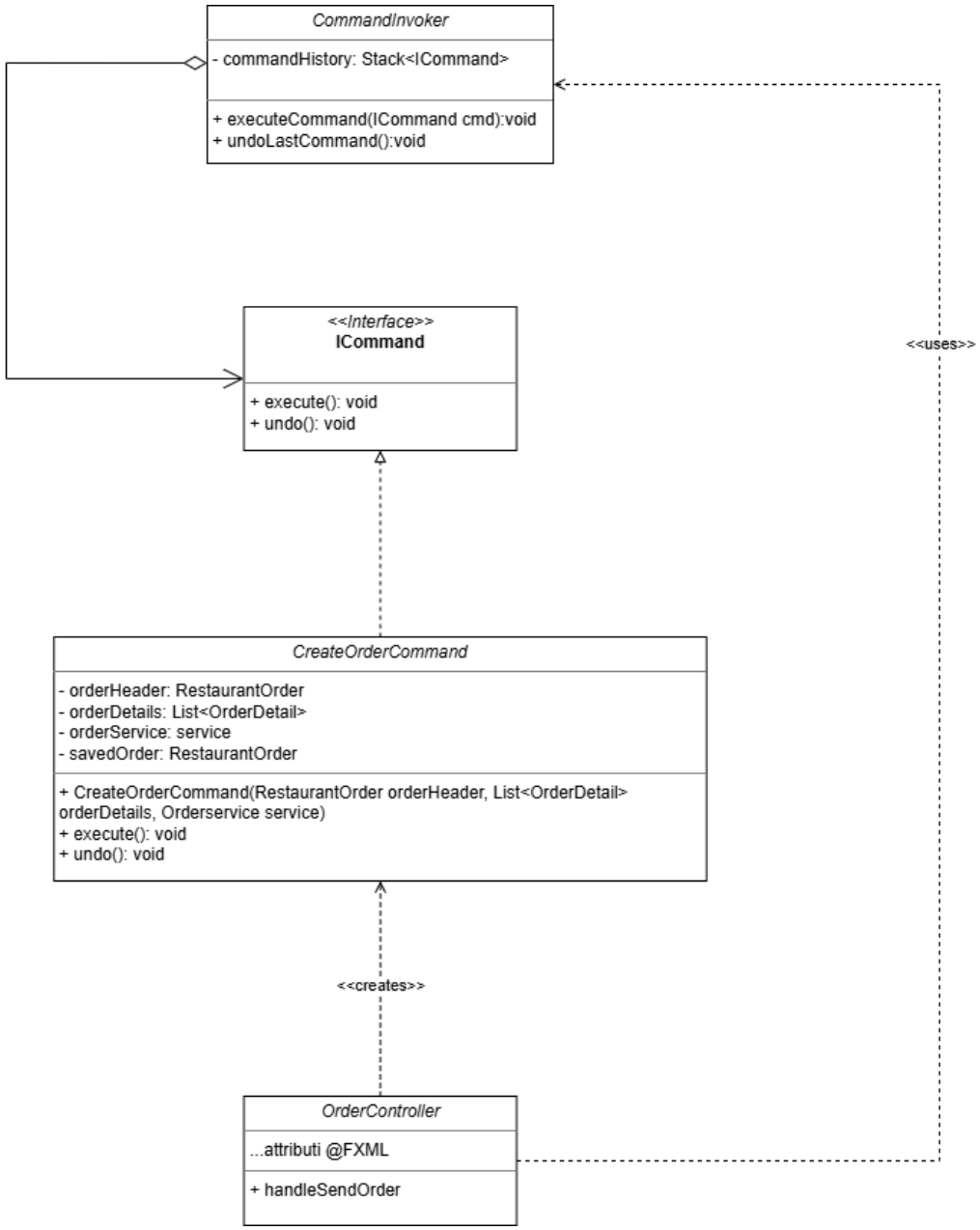

Figura 5: Diagramma UML Command Pattern

```
1
2 package org.ristorante.command;
3
4 //pattern comportamentale che in questo contesto ci permette di annullare le
      ultime azioni
5 //per ogni azione diversa = command diverso
6 //gli oggetti dello stesso dominio entrano a far parte di una coda
7 //la traccia chiede esplicitamente di annullare l'ultimo ordine effettuato
8 //poniamo il caso, piu avanti nel progetto si vuole anche annullare l'ultima
      modifica, o comunque
9 //un ultima azione sull'ordine (cancellazione, aggiornamento ...) basterebbe
      cliccare semplicemente
10 //tasto annulla (o il classico ctrl+z), quel tasto annulla non conosce i
    dettagli, lui chiama undo() e
11 //pesca l'ultimo oggetto creato che altro non e'
12 // che una richiesta. Grazie al polimorfismo verra' tradotto nell'undo
    giusto
```

```
13
14 //il contratto che deve rispettare un Comando. Qui ne abbiamo un solo tipo quindi
15 //generalizziamo il nome, altrimenti potremmo chiamarlo OrderCommand o qualcosa
16 //del genere
17
18 public interface ICommand {
19     public void execute();
20     public void undo();
21
22 }
23
24
25
26
27 package org.ristorante.command;
28
29 //l'invoker prende il ruolo di chi dice come e quando eseguire i comandi
30 //serve a snellire il controller. Quest'ultimo deve solo ricevere un evento UI(Fxml)
31 //raccogliere i dati dalla view/file.xml e costruire il command
32
33 import java.util.Stack;
34
35 public class CommandInvoker {
36     //memorizziamo nello stack (che e' perfetto per rappresentare concetto di ultima azione)
37     //oggetti che rispettano il contratto ICommand (se in futuro volessimo aggiungere altre eventi e' comodo)
38     //potremmo anche dire CreateOrderCommand, ma e' antipattern
39
40     //sfruttiamo quelle che sono le tipiche proprieta' di uno stack
41     //push, pop, isempty...
42     private Stack<ICommand> commandHistory = new Stack<>();
43
44     public void executeCommand(ICommand cmd){
45         //quindi eseguiamo e registriamo questa richiesta nello stack
46         cmd.execute();
47         commandHistory.push(cmd);
48
49         System.out.println("L'invoker ha eseguito e memorizzato la richiesta");
50         System.out.println("Dimensione stack " + commandHistory.size());
51     }
52
53     public void undoLastCommand(){
54         if(commandHistory.isEmpty()){
55             System.out.println("Niente da annullare");
56             return;
57         }
58
59         //cacciamo fuori dallo stack l'ultima richiesta
60         //tenendola ovviamente in memoria
61         //potremmo anche prima visualizzare l'ultimo comando, annullare e poi cacciare dallo stack
62         ICommand lastCmd = commandHistory.pop();
63         lastCmd.undo();
64
65         System.out.println("L'invoker ha annullato la richiesta");
```

```
66          System.out.println("Dimensione stack " + commandHistory.size());
67      }
68  }
69
70
71  //questo rappresenta il command, cioe' la richiesta
72  //che rappresenta la creazione di un oggetto
73
74  public class CreateOrderCommand implements ICommand {
75
76      // i pacchetti gi  pronti (POJO) creati dal Controller
77      final private RestaurantOrder orderHeader;   //l'ordine (senza id)
78      final private List<OrderDetail> orderDetails; // le voci (a cui dobbiamo
        assegnare ancora un id ordine)
79      final private OrderService service;
80
81      // ci serve per sapere quale ordine abbiamo creato (per l'undo)
82      private RestaurantOrder savedOrder;
83
84      // riceve i POJO pronti
85      public CreateOrderCommand(RestaurantOrder orderHeader, List<OrderDetail>
        orderDetails, OrderService service) {
86          this.orderHeader = orderHeader;
87          this.orderDetails = orderDetails;
88          this.service = service;
89      }
90
91      @Override
92      public void execute() {
93          System.out.println("Command: invio al service la richiesta di
    creazione...");
94
95          // Passiamo i dati al Service.
96          // Il Service salver  l'header, prender  l'ID, lo metter  nei
    dettagli e salver  i dettagli.
97          // Ci restituisce l'oggetto "Pieno" (con ID e tutto).
98          this.savedOrder = service.createOrder(this.orderDetails, this.
    orderHeader);
99
100         System.out.println("Command: Ordine creato con ID " + savedOrder.
    getOrder_id());
101     }
102
103     @Override
104     public void undo() {
105         // Se non ho mai salvato nulla (o    fallito), non faccio nulla
106         if (savedOrder == null || savedOrder.getOrder_id() == 0) {
107             return;
108         }
109
110         System.out.println("Command: ANNULLO ordine ID " + savedOrder.
    getOrder_id());
111
112         // Chiamo il service per cancellare
113         service.cancelOrder(savedOrder.getOrder_id());
114
115         // Reset per evitare doppi undo
116         this.savedOrder = null;
117     }
118 }
```

```
119
120
121 public class OrderController extends BaseController(){
122
123     private CommandInvoker invoker;
124
125 //...attributi
126
127
128
129
130
131
132     // Metodo chiamato quando apri la finestra
133     @FXML public void initialize() {
134         this.invoker = new CommandInvoker();
135         //...altre configurazioni, inizializzazioni di variabili etc...
136     }
137
138     //logica del bottone per inviare gli ordini, colui che usera' il command
139     @FXML
140     public void handleSendOrder() {
141         if (tempOrderList.isEmpty()) {
142             System.out.println("La comanda    vuota");
143             return;
144         }
145
146         // preparo i pojo (Pacchetti dati)
147
148         // Header dell'ordine
149         RestaurantOrder orderHeader = new RestaurantOrder();
150         orderHeader.setTable_id_fk(currentTable.getTable_id());
151         orderHeader.setUser_id_fk(activeUserID);
152         //impostiamo lo stato a "ORDERED" passando quindi un oggetto
    IOrderState
153         orderHeader.setCurrentState(new OrderedState());
154         //passiamo il totale, ci assicuriamo che e' sempre aggiornato poiche
    ' ogni volta
155         //che aggiungiamo una voce questa chiama la funzione del calcolo
    totale
156         //parsa tutti gli oggetti e ne restituisce il totale globalmente
157         orderHeader.setTotalPrice(this.total);
158         // Lista dei dettagli (Converto ObservableList in ArrayList normale)
159         List<OrderDetail> detailsToSend = new ArrayList<>(tempOrderList);
160
161         //!!! QUI UTILIZZIAMO IL COMMAND !!!
162         // creo il command
163         // passo i pacchetti pronti. Il Command non sa nulla di UI.
164         CreateOrderCommand cmd = new CreateOrderCommand(orderHeader,
    detailsToSend, orderService);
165
166         // eseguiamo tramite l'invoker
167         invoker.executeCommand(cmd);
168
169         // resetUi
170         tempOrderList.clear();
171         //puliamo il totale e la label del totale
172         //in questo modo non e' la miglior pratica pero'
173         //cerchiamo di velocizzare tutti i processi grafici
174         //per concentrarci sulla logica dei pattern
```

```
175         total = 0;
176         totalLabel.setText("Totale: 0$");
177         System.out.println("Ordine inviato");
178 
179         //vogliamo mostrare un alert con i seguenti dati
180         //abbiamo bisogno di un tipo Optional perche' l'alert puo'
    restituire anche null
181         Optional<ButtonType> result = showAlert(
182                 Alert.AlertType.CONFIRMATION,
183                 "Ordine Inviato",
184                 "L'ordine    stato trasmesso.",
185                 "Premi OK per confermare, oppure Cancel per annullarlo
    subito."
186         );
187 
188 
189 }
```

4.6 Factory Method Pattern Ottimizzato (Dashboard Strategy)

Per la gestione della navigazione e delle schermate basate sul ruolo dell'utente (`Admin`, `Waiter`, ecc.), è stato utilizzato il **Factory Pattern** abbinato a una `EnumMap`.

Invece di adottare un approccio convenzionale (e meno efficiente) basato su istruzioni `switch-case` per istanziare le strategie al volo, la `StrategyFactory` funge da *Registry*. Al momento della sua inizializzazione, pre-carica le istanze concrete delle strategie all'interno di una `EnumMap`, associando ogni enumerativo `Role` alla corrispondente implementazione di `DashboardStrategy`.

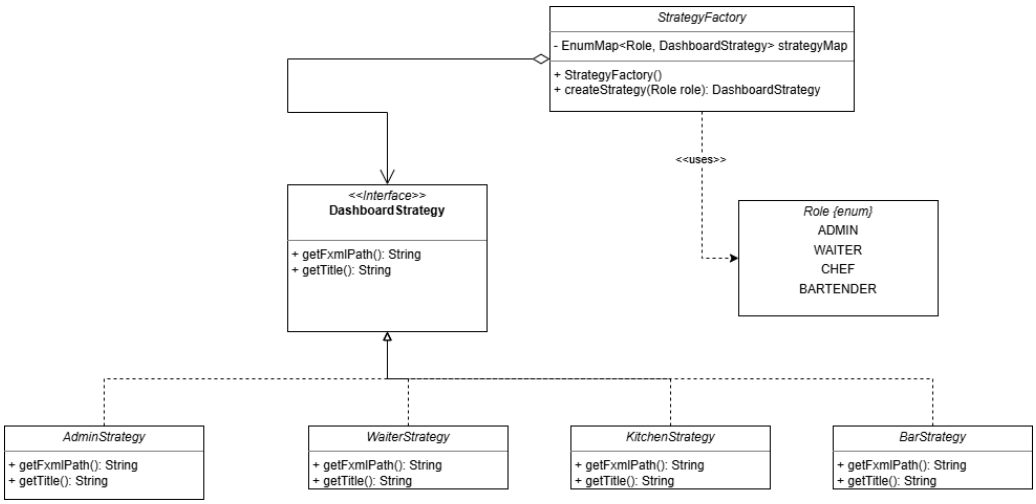

Figura 6: Diagramma UML Factory Pattern

```
1 
2 
3 
4 /*facotry + strategy
5 questo file rappresenta l'unico punto di accesso per la creazione delle
    logiche.
6 Il factory e' spesso il compagno dello strategy, offre varie potenzialita':
7 1. La factory si assume la responsabilita' di decidere quale strategy dare
    al programma in base al ruolo
8 2. niente dipendency injection, esternamente passiamo una stringa sara' la
    factory poi a decidere
```

```
    si nota la presenza di questa EnumMap (che tale potrebbe essere un hashmap o altre strutture chiave-valore)
    noi diamo la stringa e a quella stringa corrisponde una strategy. Si usa EnumMap per un fatto tecnico
    e' piu veloce, non usa algoritmi complessi di hashing ma usa l'indice.
    passando Role.class si accede proprio a tutti questi vantaggi, in pratica e' come se dessimo un cassetto
    specifico dell'armadio.
3. Se domani dovesse nascere un nuovo ruolo, basta aggiungere una riga qui e un altra costante nell'enum Role

 */

public class StrategyFactory {
    private final EnumMap<Role, DashboardStrategy> strategyMap;
    public StrategyFactory(){
        //inizializza la mappa per gli Enum
        strategyMap = new EnumMap<>(Role.class);
        strategyMap.put(Role.ADMIN, new AdminStrategy());
        strategyMap.put(Role.WAITER, new WaiterStrategy());
        strategyMap.put(Role.CHEF, new KitchenStrategy());
        strategyMap.put(Role.BARTENDER, new BarStrategy());

        System.out.println("Ho caricato correttamente le strategie");

    }

    //questo modo potrebbe anche decidere di lanciare un errore
    //per esempio "strategy non valida"
    public DashboardStrategy createStrategy(Role role){
        System.out.println("Sto ottenendo la strategy corretta...");
        return strategyMap.get(role);
    }

}



public interface DashboardStrategy {
    //e' il contratto che devono seguire le strategy

    //ottiene il Path per la pagina fxml giusta
    public String getFxmlPath();
    //ottiene informazioni riguardo il titolo
    public String getTitle();


}


public class AdminStrategy implements DashboardStrategy {
    @Override
    public String getFxmlPath() {
        System.out.println("Sto restituendo il path grazie alla strategy ...");
        return "/org/ristorante/view/admin_dashboard.fxml";
    }

    @Override
    public String getTitle() {
```

```
64          System.out.println("Sto restituendo il title grazie alla strategy
    ...");
65          return "Pannello amministratore";
66      }
67  }
68
69
70  public class WaiterStrategy implements DashboardStrategy {
71      @Override
72      public String getFxmlPath() {
73          //ricordati che devi omettere /resources
74          //poiche' e' considerata come una root
75          System.out.println("Sto restituendo il path grazie alla strategy
    ...");
76          return "/org/ristorante/view/waiter_dashboard.fxml";
77      }
78
79      @Override
80      public String getTitle() {
81          System.out.println("Sto restituendo il title grazie alla strategy
    ...");
82          return "Pannello cameriere";
83      }
84  }
85
86
87
88  public class BarStrategy implements DashboardStrategy {
89
90      @Override
91      public String getFxmlPath() {
92          System.out.println("Sto restituendo il path grazie alla strategy
    ...");
93          return "/org/ristorante/view/bar_dashboard.fxml";
94      }
95
96      @Override
97      public String getTitle() {
98          System.out.println("Sto restituendo il title grazie alla strategy
    ...");
99          return "Bar";
100     }
101 }
102
103
104 public class KitchenStrategy implements DashboardStrategy {
105
106
107     @Override
108     public String getFxmlPath() {
109         System.out.println("Sto restituendo il path grazie alla strategy
    ...");
110         return "/org/ristorante/view/kitchen_dashboard.fxml";
111     }
112
113     @Override
114     public String getTitle() {
115         System.out.println("Sto restituendo il title grazie alla strategy
    ...");
116         return "Cucina";
```

```
117         }
118 }
```

5 Conclusioni

Il progetto realizzato soddisfa i requisiti di gestione di un ristorante, offrendo un'interfaccia reattiva e una solida gestione dei dati. L'adozione dell'architettura MVC e dei Design Pattern (in particolare Strategy e DAO) ha reso il codice modulare, facile da testare e pronto per future estensioni, come l'aggiunta di nuovi metodi di pagamento o nuove funzionalità per la cucina.