



Università degli Studi di Napoli
Parthenope

Progetto di Programmazione III

OrderApp

Relazione

Biancarosa Pasquale (2098)

Militerno Eugenio (2454)

Ruotolo Pasquale (2322)

Indice

1.1 Requisiti.....	3
1.2 Analisi dei requisiti.....	3
1.3 Diagramma del Database.....	4
1.4 Diagramma delle classi.....	5
1.5 Pattern utilizzati.....	7
1.5.1 - Facade.....	7
1.5.2 - Strategy.....	8
1.5.3 - DAO (Data Access Object).....	9
1.5.3.1 Model.....	9
1.5.3.2 DAO.....	10
1.5.3.3 Interfaccia.....	11
1.5.4 - Factory.....	12
1.5.5 - Observer.....	12
1.5.6 - Decorator.....	13
1.6 - Sistema di automazione.....	14

Progetto programmazione III e Laboratorio

1.1 Requisiti

Si vuole sviluppare un software che deve gestire **il sistema di ordini in un ristorante**. L'accesso al software può avvenire in modalità **Admin** o in modalità **Utente**, per la gestione e l'aggiunta di prodotti e categorie ordinabili.

Gli ordini possono essere annullati (eliminati), e si può anche modificare un piatto.

Il cliente seduto al tavolo, **identificato da un numero univoco**, ordina un piatto, che gli verrà consegnato simulando i tempi della cucina. Il cameriere, una volta consegnato l'ordine, **segnala che è stato portato al tavolo**. Il cliente, una volta consumato l'ordine, procede in cassa dall'admin dove potrà pagare, **scegliendo fra carta, contanti e bancomat**. Una volta pagato, **esce dal ristorante liberando il tavolo**.

1.2 Analisi dei requisiti

Il sistema deve prevedere l'accesso sia in modalità **amministratore** che in modalità **utente**.

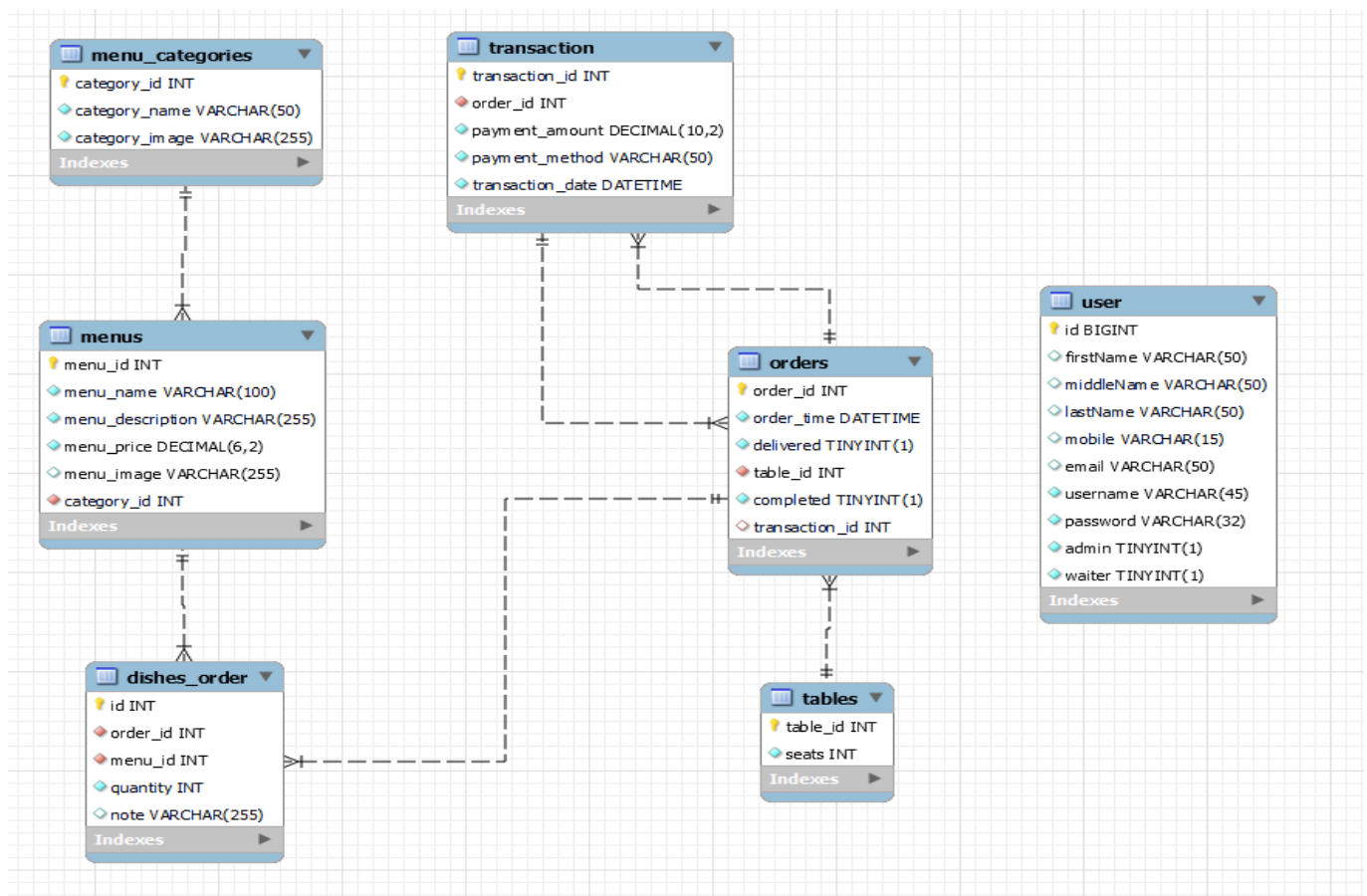
L'amministratore può effettuare le seguenti operazioni:

- **Aggiungere o modificare** piatti
- Visualizzare gli **utenti** registrati
- Visualizzare gli **ordini pagati**
- Permettere il **pagamento** degli ordini

Il Cameriere può effettuare le seguenti operazioni:

- **Aggiungere un ordine** per un tavolo
- **Rimuovere** un ordine o un piatto dall'ordine
- **Consegnare** l'ordine al tavolo

1.3 Diagramma del Database



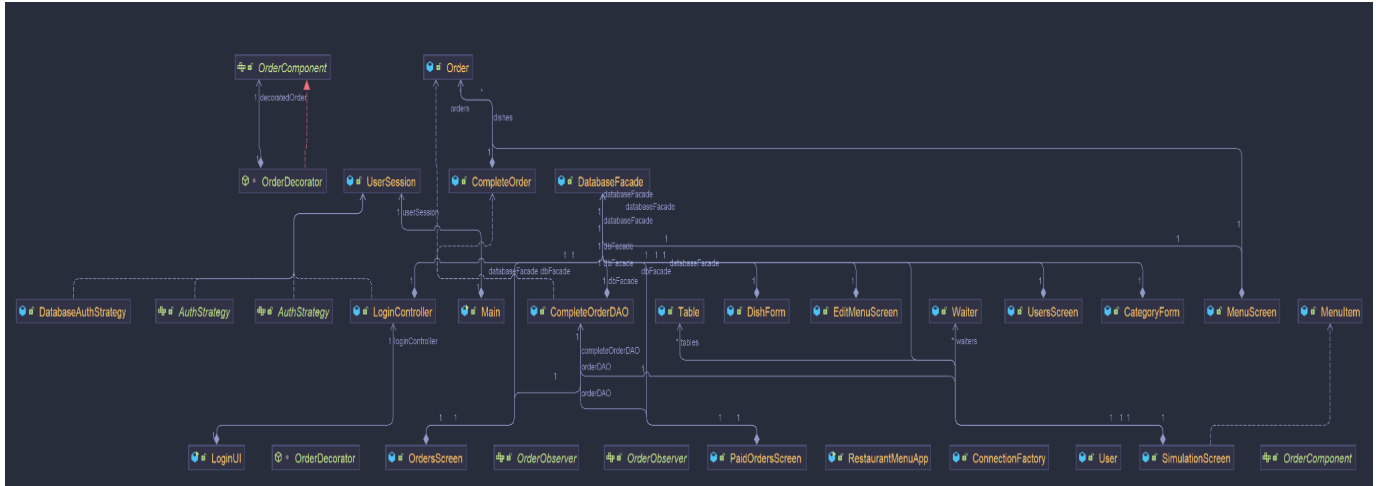
Il database memorizza i dati dell'utente nella tabella **User**, e dopo il log in ne avvia la sessione.

Tramite la flag “**admin**” e “**waiter**” è possibile differenziare gli utenti camerieri dall'admin per dargli accesso a funzioni dedicate.

Il database memorizza i menù, suddivisi in categorie, e i vari piatti, tiene traccia degli ordini effettuati e delle transazioni eseguite.

Tabelle:

- Tramite la tabella **User** vengono memorizzati i dati degli utenti del Db. Ne si protegge l'accesso tramite email e password, e grazie alle flag admin e waiter è possibile avviare una sessione con i giusti livelli di permessi.
- Tramite le tabelle **Tables, Orders, Transaction, Dishes_order** è possibile memorizzare gli ordini associati ai tavoli con il corrispettivo stato e le transazioni associate
- Tramite le tabelle **Menu, Menu_categories** vengono memorizzate i vari piatti e categorie che poi vengono associate agli ordini effettuati



Le funzioni delle classi sono le seguenti:

- **AuthStrategy** : Interfaccia che gestisce la scelta del Login (Waiter o Admin)
- **CompleteOrder** : Gestisce l'ordine ed il suo completamento (Consegna e Pagamento)
- **MenuItem** : Record che gestisce il piatto (nome,id,prezzo)
- **Order** : Gestisce il singolo ordine
- **OrderComponent** : Interfaccia che permette di aggiungere note e modificarne la quantità
- **OrderDecorator** : Gestisce la modifica dell'ordine effettiva
- **OrderObserver** : Interfaccia che permette di notificare i cambiamenti di stato dell'ordine
- **Table** : Gestisce il tavolo
- **User** : Gestisce l'utente
- **UserSession** : Gestisce la sessione d'accesso al DB
- **Waiter** : Record che gestisce il waiter
- **CategoryForm** : Gestisce il menù dei piatti, è un form a tendina che può essere modificato
- **CompleteOrderDAO** : Connette ordini DB con ordini App
- **ConnectionFactory** : Gestisce la connessione al DB
- **DatabaseAuthStrategy** : Gestisce la scelta dell'autenticazione al DB (autenticazione Waiter/Admin)

- **DatabaseFacade** : Gestisce la connessione al DB per eseguire statement ed ottenere risultati
- **DishForm** : Gestisce il form del singolo piatto (il piatto dentro la categoria)
- **EditMenuScreen** : Gestisce le modifiche che vengono fatte all'UI del menu, è UI
- **LoginController** : Gestisce l'implementazione e la logica del Login
- **LoginUI** : Gestisce l'UI del Login
- **Main** : Classe principale dell'app
- **MenuScreen** : Implementa la UI del menù, prezzi di piatti ecc
- **OrdersScreen** : Gestisce la schermata degli ordini, permette notifiche su cambiamenti di stato degli ordini
- **PaidOrdersScreen** : Gestisce gli ordini PAGATI (completati)
- **RestaurantMenuApp** : La classe centrale dell'app
- **SimulationScreen** : Gestisce la UI della schermata di simulazione
- **UserScreen** : Gestisce la UI della schermata dell'utente dell'app

1.5 Pattern utilizzati

1.5.1 - Facade

Questo pattern fornisce un'interfaccia semplificata per un insieme di interfacce di un sottosistema complesso, rendendo più facile l'utilizzo di quel sottosistema. L'idea principale è di nascondere la complessità del sistema e fornire una singola interfaccia per eseguire operazioni diverse.

```
package com.example.orderapp;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class DatabaseFacade {  Pasquale Ruotolo
    private Connection connection;  8 usages

    public Connection openConnection() throws SQLException {  Pasquale Ruotolo
        connection = ConnectionFactory.createConnection();
        return connection;
    }

    public void closeConnection() {  19 usages  Pasquale Ruotolo
        if (connection != null) {
            try {
                connection.close();
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }

    public Connection getConnection() {  Pasquale Ruotolo
        return connection;
    }

    // Additional methods for executing queries
    public ResultSet executeQuery(String query) throws SQLException {  no usages  Pasquale Ruotolo
        PreparedStatement stmt = connection.prepareStatement(query);
        return stmt.executeQuery();
    }

    public ResultSet executeQuery(String query, int param) throws SQLException {  no usages  Pasquale Ruotolo
        PreparedStatement stmt = connection.prepareStatement(query);
        stmt.setInt( parameterIndex: 1, param);
        return stmt.executeQuery();
    }

    public void executeUpdate(String query, Object... params) throws SQLException {  no usages  Pasquale Ruotolo
        PreparedStatement stmt = connection.prepareStatement(query);
        for (int i = 0; i < params.length; i++) {
            stmt.setObject( parameterIndex: i + 1, params[i]);
        }
        stmt.executeUpdate();
    }
}
```

1.5.2 - Strategy

Questo pattern consente di **scegliere l'algoritmo da utilizzare in fase di esecuzione**, senza modificare il contesto in cui l'algoritmo è utilizzato. È utile **quando ci sono più varianti di un algoritmo** e si vuole evitare un codice condizionale complesso.

```
package com.example.orderapp;

public interface AuthStrategy { 1 implementation  🧑 Pasquale Ruotolo
    UserSession authenticate(String userId, String password); 1 usage 1 implementation  🧑 Pasquale Ruotolo
}
```

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class DatabaseAuthStrategy implements AuthStrategy {  🧑 Pasquale Ruotolo
    @Override 1 usage  🧑 Pasquale Ruotolo
    public UserSession authenticate(String userId, String password) {
        UserSession userSession = null;
        DatabaseFacade dbFacade = new DatabaseFacade();
        try {
            dbFacade.openConnection();
            Connection conn = dbFacade.getConnection();
            String query = "SELECT admin FROM user WHERE username = ? AND password = ?";
            try (PreparedStatement preparedStatement = conn.prepareStatement(query)) {
                preparedStatement.setString( parameterIndex: 1, userId);
                preparedStatement.setString( parameterIndex: 2, password);

                ResultSet resultSet = preparedStatement.executeQuery();
                if (resultSet.next()) {
                    boolean isAdmin = resultSet.getBoolean( columnLabel: "admin");
                    userSession = UserSession.getInstance(userId, isAdmin);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            dbFacade.closeConnection();
        }
        return userSession;
    }
}
```

Tramite la classe **DatabaseAuthStrategy** una volta inserite le credenziali di accesso **la sessione è gestita in tempo reale** per garantire l'accesso alle funzioni admin.

1.5.3 - DAO (Data Access Object)

Il pattern DAO è usato per **separare la logica di business dalla logica di accesso ai dati**. Infatti, i componenti della logica di business non dovrebbero mai accedere direttamente al database: questo comporterebbe scarsa manutenibilità. **Solo gli oggetti previsti dal pattern Dao possono accedervi**. Inoltre, se dovessimo modificare il tipo di memoria persistente utilizzata, o anche passare da MySql ad un altro database per esempio, non sarà necessario stravolgere il codice della nostra applicazione, ma basterà modificare i DAO utilizzati.

I concetti principali sono:

- la classe **CompleteOrder** ovvero il modello per gli ordini completati
- un'interfaccia (detta DAO) per ogni tabella contenente tutti i metodi **CRUD** relativi a quella tabella.
- l'interfaccia

1.5.3.1 Model

```
package com.example.orderapp;

import java.sql.Timestamp;
import java.util.ArrayList;
import java.util.List;

public class CompleteOrder {
    private final int orderId;
    private final int tableId;
    private boolean delivered;
    private boolean completed;
    private List<Order> dishes;
    private String paymentMethod;
    private Timestamp transactionDate;

    public CompleteOrder(int orderId, int tableId) {...}

    public int getOrderId() { return orderId; }

    public int getTableId() { return tableId; }

    public boolean isDelivered() { return delivered; }

    public void setDelivered(boolean delivered) { this.delivered = delivered; }

    public boolean isCompleted() { return completed; }

    public void setCompleted(boolean completed) { this.completed = completed; }

    public List<Order> getDishes() { return dishes; }

    public void addDish(Order order) { this.dishes.add(order); }

    public double getTotalPrice() { return dishes.stream().mapToDouble(Order::getTotalPrice).sum(); }

    public String getPaymentMethod() { return paymentMethod; }

    public void setPaymentMethod(String paymentMethod) { this.paymentMethod = paymentMethod; }


    public Timestamp getTransactionDate() { return transactionDate; }


    public void setTransactionDate(Timestamp transactionDate) { this.transactionDate = transactionDate; }
}
```


1.5.3.2 DAO


```
package com.example.orderapp;



import java.sql.*;
import java.util.ArrayList;
import java.util.List;

public class CompleteOrderDAO {  Pasquale Ruotolo

    private DatabaseFacade dbFacade;  24 usages

    public CompleteOrderDAO() {  Pasquale Ruotolo
        |     this.dbFacade = new DatabaseFacade();
        |
    }

    private List<OrderObserver> observers = new ArrayList<>();  1 usage

    public void addObserver(OrderObserver observer) {  no usages  Pasquale Ruotolo
        |     observers.add(observer);
        |
    }

    public List<CompleteOrder> getAllNonCompletedOrders() {...}

    public List<CompleteOrder> getAllCompletedOrders() {...}

    public void updateOrderStatus(int orderId, boolean delivered) {...}

    public void deleteOrder(int orderId) {...}

    public void deleteDish(int orderId, int menuId) {...}

    public int processPaymentTransaction(int orderId, String paymentMethod, int tableId) {...}

    public List<Order> getOrderDetails(int orderId) {...}
}
```

1.5.3.3 Interfaccia

```
public class OrdersScreen extends Stage implements OrderObserver {  Pasquale Ruotolo

    private VBox mainLayout; 5 usages
    private final CompleteOrderDAO orderDAO; 6 usages
    private final DatabaseFacade databaseFacade = new DatabaseFacade(); 2 usages

    public OrdersScreen(Stage primaryStage) {...}

    private void loadOrders() {....}

    private void updateOrderStatus(int orderId, boolean isDelivered) {...}

    private void deleteOrder(int orderId) {...}

    private void deleteDish(int orderId, int menuId) {...}

    private void showPaymentDialog(int orderId, double totalPrice, int tableId) {...}

    private void processPayment(int orderId, String paymentMethod, double amountReceived, double totalPrice, int tableId) {...}

    private void generateReceipt(int orderId, String paymentMethod, double amountReceived, double total) {....}

    private boolean isNumeric(String str) {...}

    private void showErrorAlert(String title, String message) {...}

    private void showPaidOrdersScreen(Stage primaryStage) {...}

    @Override 1 usage  Pasquale Ruotolo
    public void onOrderStatusChanged(int orderId, boolean isDelivered) {...}

    @Override 1 usage  Pasquale Ruotolo
    public void onOrderDeleted(int orderId) {...}

    @Override 1 usage  Pasquale Ruotolo
    public void onDishDeleted(int orderId, int menuId) {...}

    @Override 1 usage  Pasquale Ruotolo
    public void onPaymentProcessed(int orderId, int transactionId, String paymentMethod, double amountReceived) {...}
}
```

1.5.4 - Factory

Il factory method pattern descrive un approccio di programmazione con il quale **poter creare oggetti senza bisogno di dover specificare la loro classe**. Questo permette di **cambiare comodamente e in maniera flessibile** l'oggetto creato. Lo sviluppatore sceglie se specificare il factory method in un'interfaccia e quindi implementarlo come classe figlio o come classe base ed eventualmente sovrascrivere dalle classi derivate. Questo metodo **opera a livello della classe costruttore standard** per separare la costruzione degli oggetti dagli oggetti stessi e permettere così l'utilizzo dei principi **SOLID**.

```
package com.example.orderapp;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {  🐞 Pasquale Ruotolo
    private static final String DB_URL = "jdbc:mysql://localhost:3306/restaurant";  1 usage
    private static final String DB_USER = "root";  1 usage
    private static final String DB_PASSWORD = "562656";  1 usage

    public static Connection createConnection() throws SQLException {  1 usage  🐞 Pasquale Ruotolo
        return DriverManager.getConnection(DB_URL, DB_USER, DB_PASSWORD);
    }
}
```

1.5.5 - Observer

Questo pattern è utilizzato per implementare **un meccanismo di notifica** che consente agli oggetti di osservare e reagire ai cambiamenti di stato di un altro oggetto.

```
package com.example.orderapp;

public interface OrderObserver {  1 implementation  🐞 Pasquale Ruotolo
    void onOrderStatusChanged(int orderId, boolean delivered);  1 usage  1 implementation  🐞 Pasquale Ruotolo
    void onOrderDeleted(int orderId);  1 usage  1 implementation  🐞 Pasquale Ruotolo
    void onDishDeleted(int orderId, int menuId);  1 usage  1 implementation  🐞 Pasquale Ruotolo
    void onPaymentProcessed(int orderId, int transactionId, String paymentMethod, double amountReceived);
}
```

1.5.6 - Decorator

Questo pattern è utile **per aderire al principio di responsabilità singola**, evitando la proliferazione di sottoclassi attraverso l'uso della composizione anziché dell'ereditarietà. (serve per inserire note negli ordini)

```
package com.example.orderapp;

abstract class OrderDecorator implements OrderComponent { no usages  🧑 Pasquale Ruotolo
    protected final OrderComponent decoratedOrder; 10 usages

    public OrderDecorator(OrderComponent decoratedOrder) { no usages  🧑 Pasquale Ruotolo
        this.decoratedOrder = decoratedOrder;
    }

    @Override 4 usages  🧑 Pasquale Ruotolo
    public int getMenuId() { return decoratedOrder.getMenuId(); }

    @Override 3 usages  🧑 Pasquale Ruotolo
    public String getDishName() { return decoratedOrder.getDishName(); }

    @Override 4 usages  🧑 Pasquale Ruotolo
    public double getDishPrice() { return decoratedOrder.getDishPrice(); }

    @Override 5 usages  🧑 Pasquale Ruotolo
    public int getQuantity() { return decoratedOrder.getQuantity(); }

    @Override 2 usages  🧑 Pasquale Ruotolo
    public void incrementQuantity() { decoratedOrder.incrementQuantity(); }

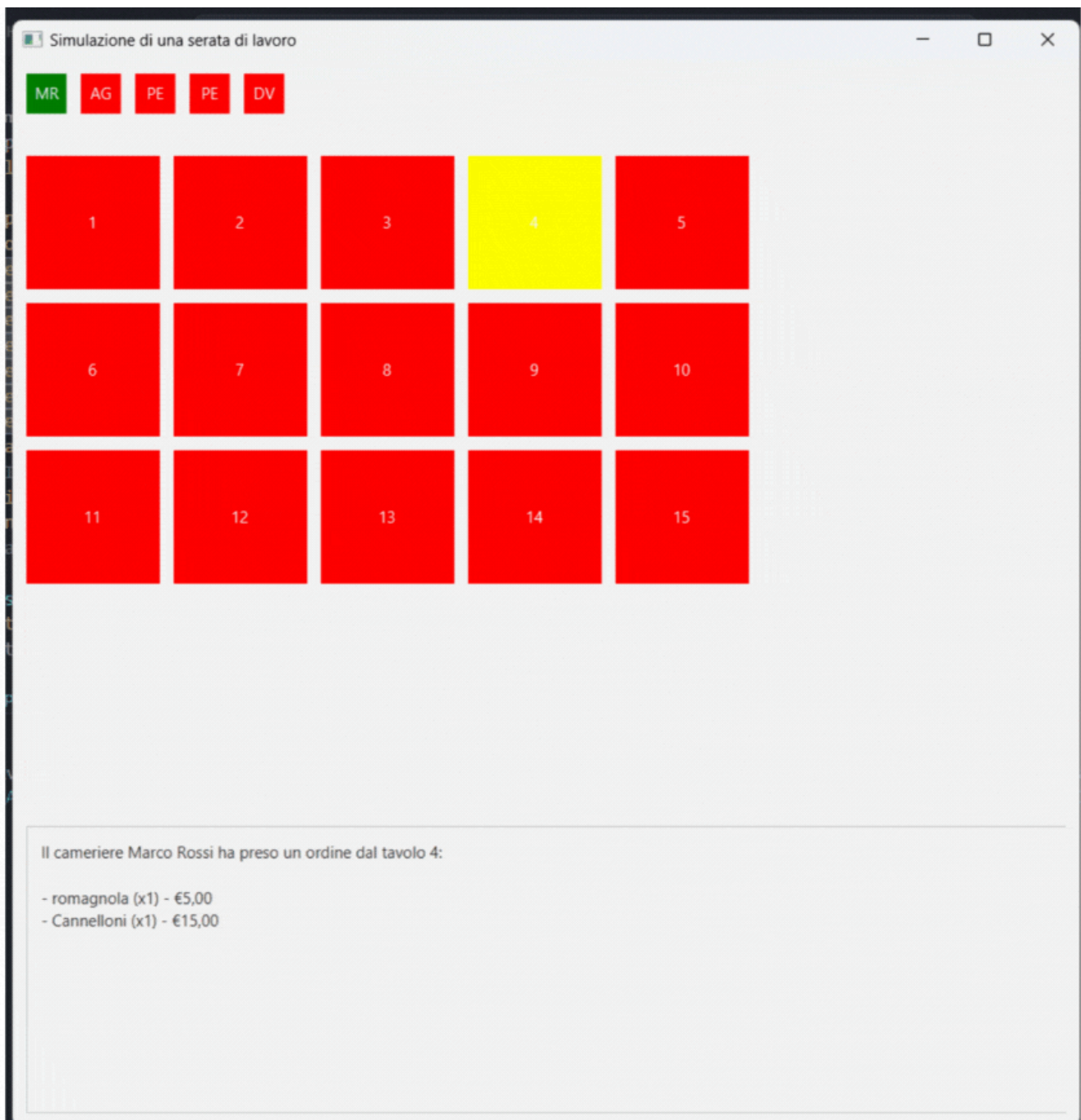
    @Override 3 usages  🧑 Pasquale Ruotolo
    public void setQuantity(int quantity) { decoratedOrder.setQuantity(quantity); }

    @Override 2 usages  🧑 Pasquale Ruotolo
    public double getTotalPrice() { return decoratedOrder.getTotalPrice(); }

    @Override 3 usages  🧑 Pasquale Ruotolo
    public String getNotes() { return decoratedOrder.getNotes(); }

    @Override 3 usages  🧑 Pasquale Ruotolo
    public void setNotes(String notes) { decoratedOrder.setNotes(notes); }
}
```

1.6 - Sistema di automazione



Grazie a questa simulazione possiamo testare il corretto funzionamento del software

- I tavoli **rossi** sono quelli **liberi**
 - I tavoli **gialli** sono quelli che hanno ordinato e **stanno aspettando l'ordine**
 - I tavoli **verdi** sono quelli che hanno consumato l'ordine, e **stanno pagando**.
- Dopo di ciò, il tavolo tornerà rosso.