

Breaking Bread

Tommaso Ticci, Guillermo Caceres

Aprile 2025



Indice

1	Introduzione	3
1.1	Descrizione del progetto	3
1.2	Struttura del progetto e tecnologie usate	3
2	Progettazione	4
2.1	Use Case Diagram	4
2.2	Use Case Template	5
2.3	Mockups	11
2.4	Class Diagram	14
2.4.1	Domain Model	14
2.4.2	Business Logic	14
2.4.3	ORM	15
2.5	ER Diagram	17
2.6	Navigation Diagram	18
2.7	Directories	19
3	Implementazione	20
3.1	Domain Model	20
3.1.1	User	20
3.1.2	Item	20
3.1.3	PaymentMethod	20
3.1.4	Order	20
3.1.5	Order Factory	20
3.2	Business Logic	21
3.2.1	LogInController	21
3.2.2	UserManageOrderController	21
3.2.3	UserViewMenuController	21
3.2.4	UserProfileController	21
3.2.5	AdminDatabaseOptionsController	22
3.2.6	AdminUserControl	22
3.2.7	Admin Order Controller	22
3.2.8	AdminItemController	22
3.3	Object-Relational Mapping	23
3.3.1	ConnectionManager	23
3.3.2	UserDAO	23
3.3.3	AdminDAO	23
3.3.4	PaymentMethodDAO	24
3.3.5	ItemDAO	26
3.3.6	OrderDAO	26
3.4	Database	29
4	Testing	30
4.1	BusinessLogicTest	30
4.2	DomainModelTest	32

1 Introduzione

Progetto realizzato per il superamento del corso di Ingegneria del Software e Laboratorio di Informatica del Corso di Laurea in Ingegneria Informatica. Il progetto consiste in un sistema di classi Java per la gestione dei menu e degli ordini di un fast-food. Realizzato da Tommaso Ticci mat. 7110440 e Guillermo Caceres mat. 7111172. Il progetto è disponibile sulla repository GitHub all'indirizzo: <https://github.com/TommasoTicci/BreakingBread.git>

1.1 Descrizione del progetto

Il progetto permette agli utenti di effettuare ordini dal menu di un fast-food e di gestire i propri ordini. L'utente amministratore può organizzare il menu, inserendo o rimuovendo pietanze, e gestire gli ordini, gli utenti e i loro metodi di pagamento attraverso un'interfaccia dedicata. L'admin ha inoltre accesso a metodi di gestione diretta del database.

1.2 Struttura del progetto e tecnologie usate

Il progetto è interamente realizzato in Java. La gestione dei dati è affidata al DBMS PostgreSQL, integrato mediante l'utilizzo del pattern DAO, che permette di separare la logica di business dalla logica di persistenza dei dati. A questo scopo, il progetto è stato strutturato nei package: **Business Logic**, **Domain Model** e **ORM**.

La progettazione è stata realizzata attraverso diagrammi Use Case Diagram e Class Diagram secondo lo standard UML. Per la fase di testing è stato utilizzato JUnit.

Programmi utilizzati:

- **IntelliJ IDEA**: IDE per la scrittura e compilazione del codice Java
- **GitHub**: piattaforma online per il versionamento e condivisione del codice
- **StarUML**: software per la creazione di vari diagrammi
- **Draw.io**: web app per la creazione di vari diagrammi
- **Lunacy**: software per la creazione di mock-ups

Durante la scrittura del codice sono state usate tecnologie generative AI come **GitHub Copilot**, utilizzate principalmente per la scrittura veloce di codice per la realizzazione di classi semplici.

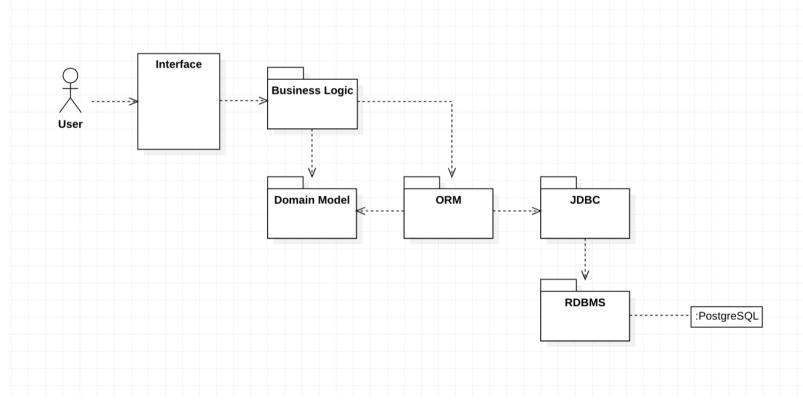


Figure 1: Package Dependency

2 Progettazione

2.1 Use Case Diagram

Come già detto la progettazione è stata realizzata con uno Use Case Diagram (figura 2), dal quale si notano i ruoli dello "User" e del "Admin" e le operazioni che questi posso fare.

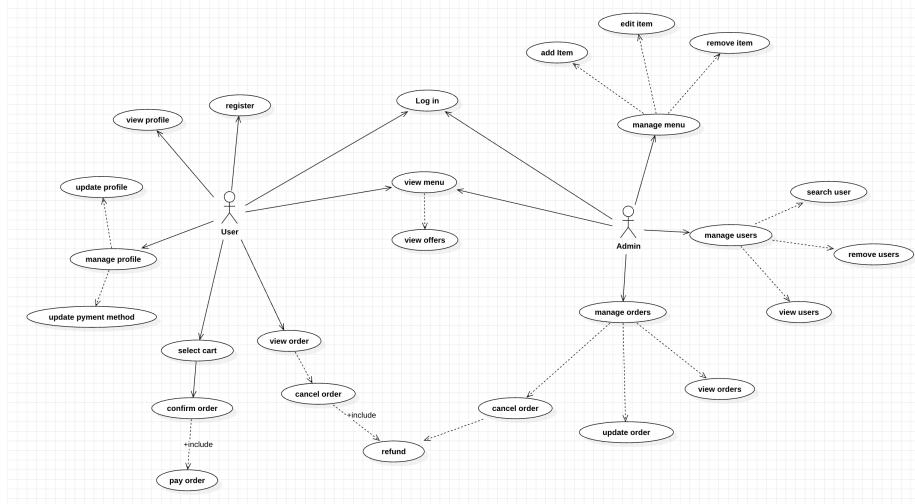


Figure 2: Use Case Diagram

2.2 Use Case Template

Di seguito sono riportati i template di alcuni dei casi d'uso, in ciascuno possiamo trovare: una breve descrizione, il livello del caso d'uso, gli attori coinvolti, le pre-condizioni, le post-condizioni, il flusso base e i flussi alternativi.

```
TST#01: loginTest()
TST#02: registerTest()
TST#03: addItemTest()
TST#04: confirmCurrentOrderTest()
TST#05: cancelOrderTest()
TST#06: changePasswordTest()
TST#07: setPaymentMethod()
TST#08: deleteCompletedOrdersTest()
```

Figure 3: Test

Use Case #1	Accesso al sistema
Brief Description	L'utente accede al sistema (MK#01, TST#01)
Level	User goal
Actors	User, Admin
Pre-conditions	L'utente deve essere nella pagina di login del programma
Basic Flow	<ol style="list-style-type: none"> 1. L'utente inserisce le proprie credenziali di accesso 2. L'utente preme il pulsante di conferma per accedere al sistema 3. Il sistema controlla che le credenziali siano corrette e che ci sia corrispondenza sul database 4. Il sistema autentica l'utente
Alternative Flows	<ol style="list-style-type: none"> 3a. Se le credenziali fornite sono errate, il sistema mostra un messaggio di errore all'utente 4a. Se a fare l'accesso è l'admin, il sistema lo reindirizza alla Admin Page (MK#6)
Post-conditions	L'utente ha accesso al sistema

Table 1: Use Case Template 1 (Accesso al sistema)

Use Case #2	Registrazione al sistema
Brief Description	L'utente crea un nuovo account e si registra al sistema (MK#02, TST#02)
Level	User goal
Actors	User
Pre-conditions	L'utente deve essere nella pagina di registrazione
Basic Flow	<ol style="list-style-type: none"> 1. L'utente inserisce i propri dati e le sue nuove credenziali 2. L'utente preme il pulsante di conferma per registrarsi al sistema 3. Il sistema controlla che i dati forniti siano corretti 4. Il sistema registra il nuovo utente nel database
Alternative Flows	<ol style="list-style-type: none"> 1a. L'utente può inserire il metodo di pagamento, non richiesto obbligatoriamente al momento della registrazione 3a. Se l'utente inserisce dati errati o se l'username o l'email sono già utilizzati da un altro utente, il sistema mostra un messaggio di errore
Post-conditions	L'utente è registrato nel sistema

Table 2: Use Case Template 2 (Registrazione al sistema)

Use Case #3	Inserimento pietanza nel carrello
Brief Description	L'utente inserisce una pietanza nel suo carrello (TST#03)
Level	User goal
Actors	User
Pre-conditions	L'utente deve essere nella pagina del proprio carrello (MK#04)
Basic Flow	<ol style="list-style-type: none"> 1. L'utente preme il pulsante "Add Item" 2. L'utente digita l'id della pietanza che vuole aggiungere al carrello 3. L'utente digita la quantità di tale pietanza 4. Il sistema controlla i dati inseriti
Alternative Flows	<ol style="list-style-type: none"> 4a. Se l'utente inserisce un id non esistente o una quantità negativa, il sistema mostra un messaggio di errore 4b. Se l'utente inserisce una quantità pari a zero, il sistema non inserisce nessuna pietanza
Post-conditions	La pietanza viene aggiunta al carrello

Table 3: Use Case Template 3 (Inserimento pietanza nel carrello)

Use Case #4	Pagamento di un ordine
Brief Description	L'utente effettua il pagamento dell'ordine nel carrello (MK#05, TST04)
Level	Function
Actors	User
Pre-conditions	L'utente ha premuto il pulsante di conferma ordine nella pagina del carrello (MK#04)
Basic Flow	<ol style="list-style-type: none"> 1. Il sistema mostra il riepilogo dell'ordine 2. L'utente digita il CVV della carta collegata 3. Il sistema verifica i dati ed esegue il pagamento 4. Il sistema conferma all'utente che il pagamento è stato effettuato con successo
Alternative Flows	<ol style="list-style-type: none"> 1a. Se il carrello è vuoto, il sistema mostra un messaggio di errore 1b. Se l'utente non ha nessun metodo di pagamento associato all'account, il sistema mostra un messaggio di errore 3a. Se l'utente inserisce dati errati, il sistema mostra un messaggio di errore
Post-conditions	L'ordine effettuato viene aggiunto nel sistema

Table 4: Use Case Template 4 (Pagamento di un ordine)

Use Case #5	Cancellazione di un ordine
Brief Description	L'utente cancella un ordine dal sistema e riceve un rimborso (TST#05)
Level	User goal
Actors	User
Pre-conditions	L'utente deve aver effettuato tale ordine
Basic Flow	<ol style="list-style-type: none"> 1. L'utente digita l'id dell'ordine che vuole cancellare 2. Il sistema verifica l'id inserito 3. Il sistema effettua un rimborso all'utente 4. Il sistema cancella l'ordine e comunica all'utente il successo di tale operazione
Alternative Flows	<ol style="list-style-type: none"> 2a. Se l'id è inesistente oppure l'id dell'ordine esiste ma questo è stato effettuato da un altro utente, il sistema mostra un messaggio di errore 2b. Se l'id dell'ordine ha corrispondenza, ma l'ordine ha uno status diverso da "Recieved", il sistema mostra un messaggio di errore 3a. Se l'utente non ha nessun metodo di pagamento associato all'account, il sistema mostra un messaggio di errore
Post-conditions	L'ordine selezionato viene rimosso dal sistema e l'utente viene rimborsato

Table 5: Use Case Template 5 (Cancellazione di un ordine)

Use Case #6	Modifica della password
Brief Description	L'utente modifica la propria password (TST#06)
Level	User goal
Actors	User
Pre-conditions	L'utente deve essere autenticato
Basic Flow	<ol style="list-style-type: none"> 1. L'utente accede alle impostazioni del proprio account 2. L'utente seleziona l'opzione per la modifica della password 3. L'utente inserisce la nuova password 4. L'utente inserisce la conferma della nuova password 5. Il sistema controlla i dati inseriti 6. Il sistema aggiorna la password dell'utente
Alternative Flows	<ol style="list-style-type: none"> 5a. Se la password è vuota o la conferma password non corrisponde alla nuova password, il sistema mostra un messaggio di errore
Post-conditions	La password dell'utente viene aggiornata

Table 6: Use Case Template 6 (Modifica della password)

Use Case #7		Inserimento del metodo di pagamento
Brief Description	L'utente inserisce un metodo di pagamento nel sistema (TST#07)	
Level	User goal	
Actors	User	
Pre-conditions	L'utente deve essere autenticato	
Basic Flow	<ol style="list-style-type: none"> 1. L'utente accede alle impostazioni del proprio account 2. L'utente seleziona l'opzione per l'inserimento del metodo di pagamento 3. L'utente inserisce i dati del suo metodo di pagamento 4. Il sistema controlla i dati inseriti 5. Il sistema associa il nuovo metodo di pagamento all'account dell'utente 	
Alternative Flows	<ol style="list-style-type: none"> 4a. Se i dati inseriti sono invalidi, il sistema mostra un messaggio di errore 5a. Se l'utente aveva già associato un metodo di pagamento, questo viene sostituito da quello nuovo 	
Post-conditions	Il nuovo metodo di pagamento viene associato all'account dell'utente	

Table 7: Use Case Template 7 (Inserimento del metodo di pagamento)

Use Case #8		Aggiunta di nuova pietanza
Brief Description	L'admin aggiunge una nuova pietanza al menu (MK#7)	
Level	User goal	
Actors	Admin	
Pre-conditions	L'admin deve essere autenticato dal sistema	
Basic Flow	<ol style="list-style-type: none"> 1. L'admin accede alle impostazioni del menu 2. L'admin seleziona l'opzione per l'aggiunta di una nuova pietanza 3. L'admin inserisce i dati e le informazioni della nuova pietanza (MK#7) 4. Il sistema verifica i dati inseriti 5. Il sistema aggiunge la nuova pietanza al menu 	
Alternative Flows	<ol style="list-style-type: none"> 4a. Se i dati inseriti sono invalidi, il sistema mostra un messaggio di errore 	
Post-conditions	La nuova pietanza viene aggiunta al menu	

Table 8: Use Case Template 8 (Aggiunta di una nuova pietanza)

Use Case #9	Modifica dello stato di un ordine
Brief Description	L'admin modifica lo stato di un ordine
Level	User goal
Actors	Admin
Pre-conditions	L'admin deve essere autenticato dal sistema
Basic Flow	<ol style="list-style-type: none"> 1. L'admin accede alla pagina di gestione degli ordini 2. L'admin seleziona l'opzione per la modifica dello stato di un ordine 3. L'admin inserisce l'id dell'ordine che vuole modificare e il nuovo status 4. Il sistema verifica i dati inseriti 5. Il sistema modifica lo stato dell'ordine selezionato
Alternative Flows	<ol style="list-style-type: none"> 4a. Se l'id o lo stato inseriti sono invalidi, il sistema mostra un messaggio di errore
Post-conditions	Lo stato dell'ordine selezionato viene modificato

Table 9: Use Case Template 9 (Modifica dello stato di un ordine)

Use Case #10	Reset del database
Brief Description	L'admin effettua il ripristino del database
Level	Function
Actors	Admin
Pre-conditions	L'admin deve essere autenticato dal sistema
Basic Flow	<ol style="list-style-type: none"> 1. L'admin accede alla pagina di gestione del database 2. L'admin seleziona l'opzione per il reset del database 3. L'admin reinserisce la password per confermare l'operazione 4. Il sistema esegue il ripristino del database
Alternative Flows	<ol style="list-style-type: none"> 3a. Se la password inserita è errata, il sistema mostra un messaggio di errore
Post-conditions	Il database viene ripristinato con successo

Table 10: Use Case Template 10 (Reset del database)

2.3 Mockups

In seguito vengono mostrati alcuni Mockups dell'interfaccia del programma.

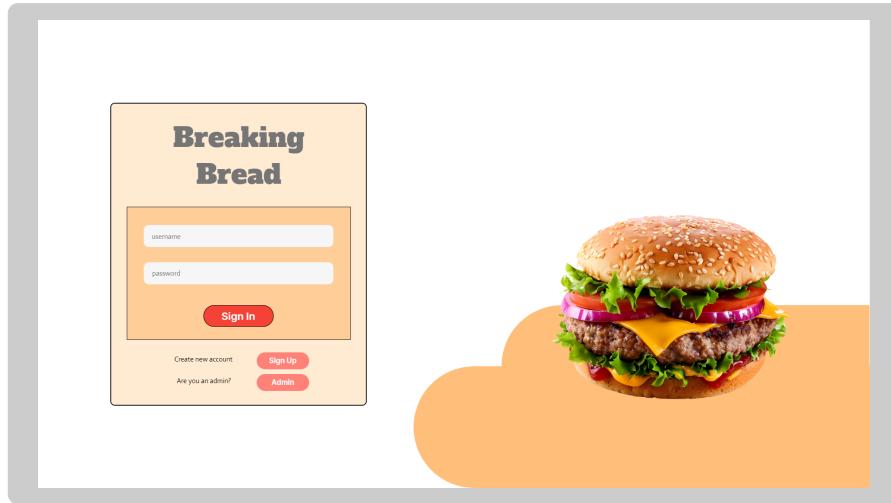


Figure 4: Mockup della pagina iniziale di login - MK#1

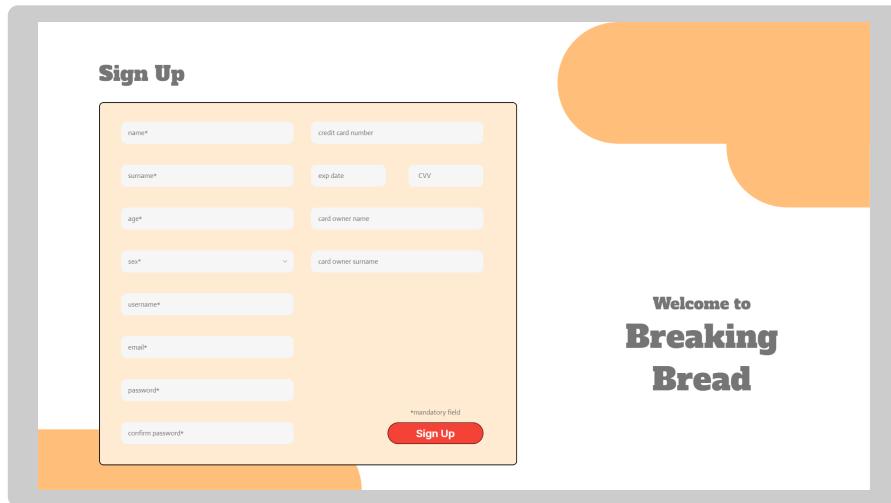


Figure 5: Mockup della pagina di registrazione - MK#2

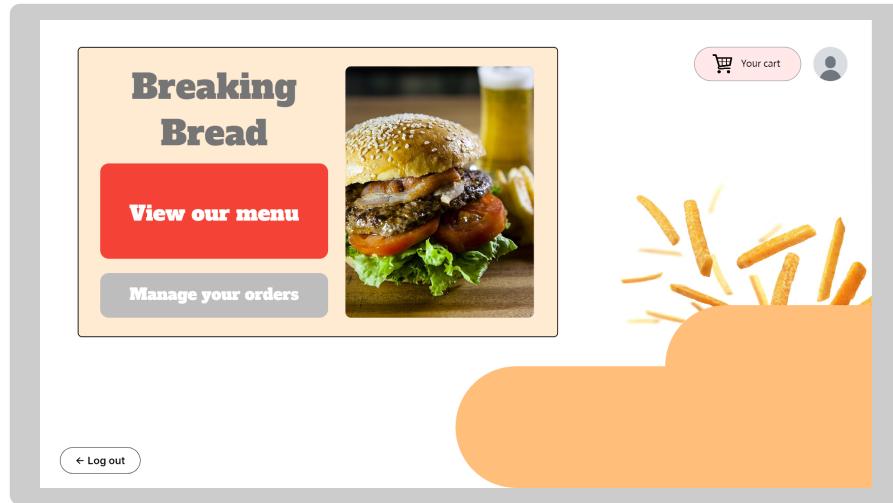


Figure 6: Mockup della homepage dell'utente - MK#3

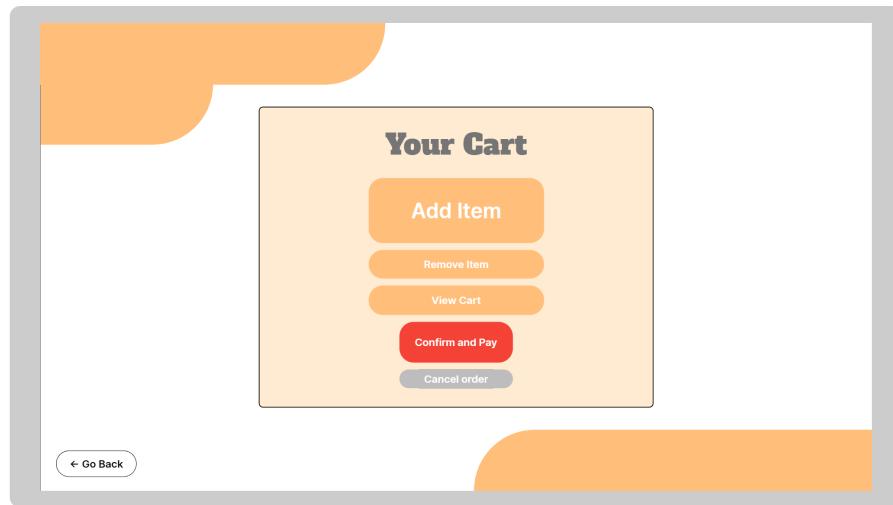


Figure 7: Mockup della pagina del carrello dell'utente - MK#4

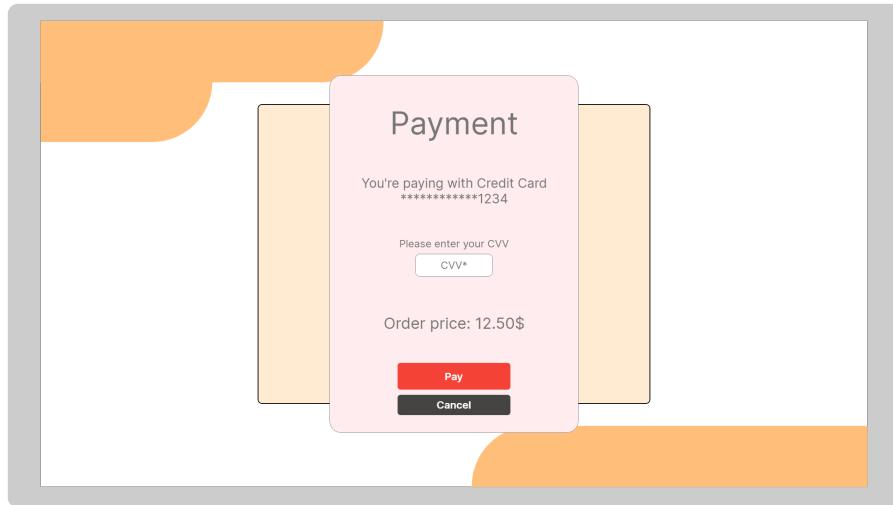


Figure 8: Mockup della pagina di pagamento di un ordine - MK#5



Figure 9: Mockup della homepage dell'admin - MK#6

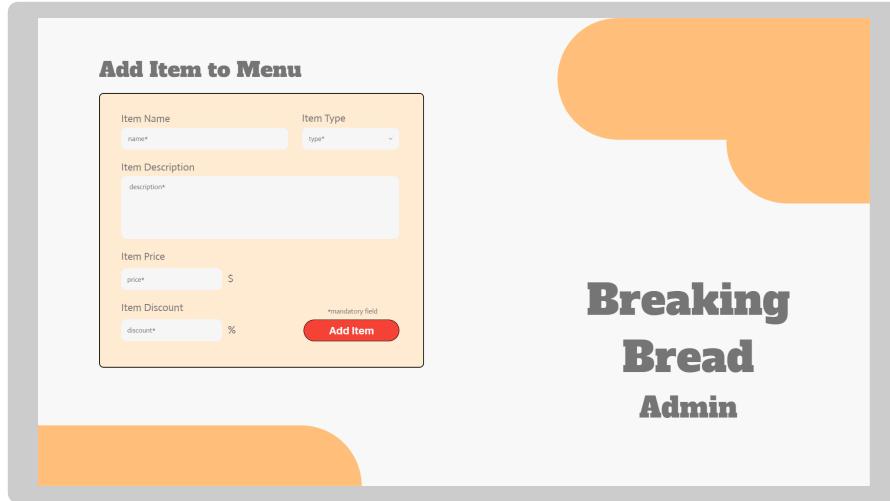


Figure 10: Mockup della pagina per l'aggiunta di una pietanza - MK#7

2.4 Class Diagram

Come già anticipato il progetto è strutturato in tre package, con lo scopo di separare le responsabilità e seguire il principio "Separation of Concerns", l'obiettivo è di avere un codice mantenibile e più facilmente testabile.

2.4.1 Domain Model

Il livello Domain Model ha lo scopo di rappresentare le entità principali (figura 11), definendo per ognuna i propri attributi e i metodi strettamente legati alla classe, come nel caso del PaymentMethod.

2.4.2 Business Logic

La Business Logic rappresentata nello (figura 12) gestisce il comportamento principale del sistema, separando le funzioni tra utenti e amministratori. Il suo scopo è coordinare le operazioni legate a ordini, gestione del profilo, autenticazione, visualizzazione del menu/offerte, e operazioni amministrative come la gestione del database, utenti, ordini e articoli. In sostanza, definisce le regole e i processi chiave che guidano il funzionamento dell'applicazione.

2.4.3 ORM

Il livello ORM (figura 13) ha lo scopo di gestire l'interazione tra l'applicazione e il database. Fornisce metodi per accedere, salvare, modificare ed eliminare dati riguardanti utenti, ordini, articoli, metodi di pagamento e amministratori. Utilizza un singleton chiamato ConnectionManager per garantire una gestione centralizzata e sicura della connessione al database. In sintesi, questo livello traduce gli oggetti del dominio in dati persistenti nel database e viceversa.

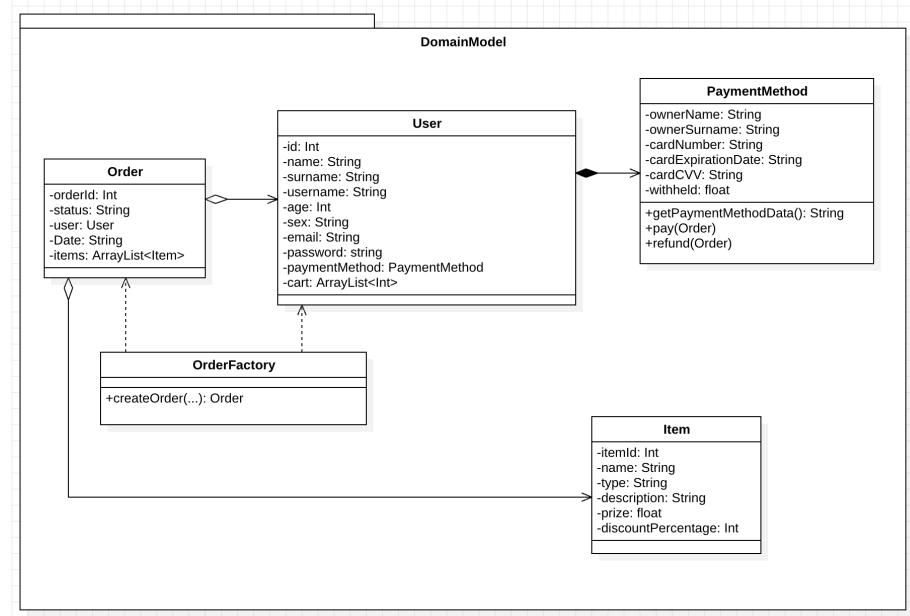


Figure 11: Domain Model

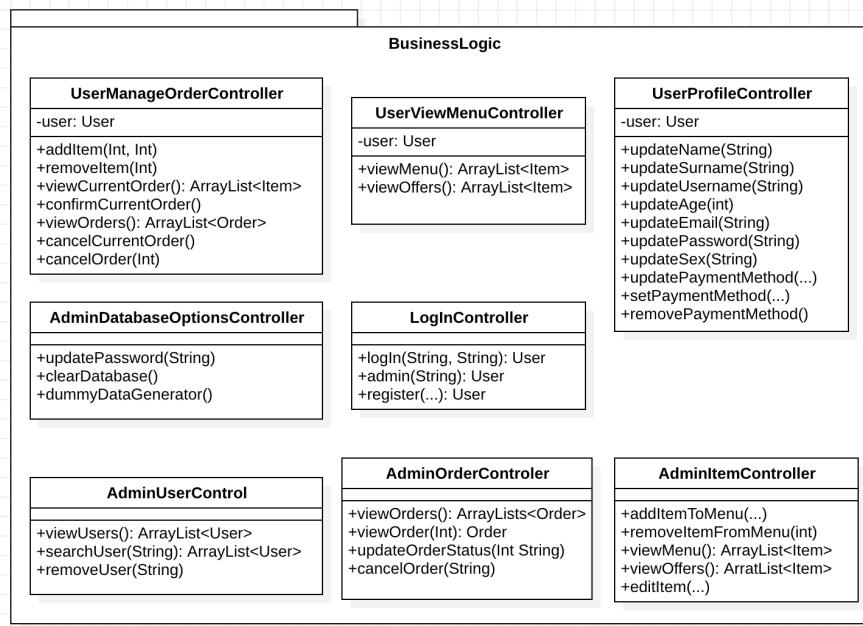


Figure 12: Business Logic

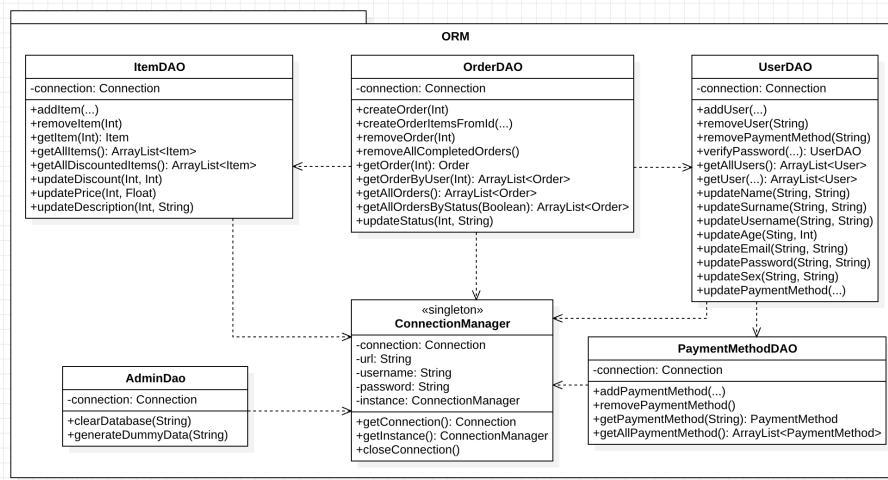


Figure 13: ORM

2.5 ER Diagram

La progettazione del database è stata realizzata attraverso uno schema concettuale ER (Entity-Relationship), mostrato nello (figura 14). In questo diagramma sono raffigurate tutte le entità principali del sistema e le relative relazioni. Lo schema rappresenta la struttura logica del database prima della successiva fase di implementazione fisica in PostgreSQL, per la quale abbiamo invece usato uno schema relazionale (figura 15).

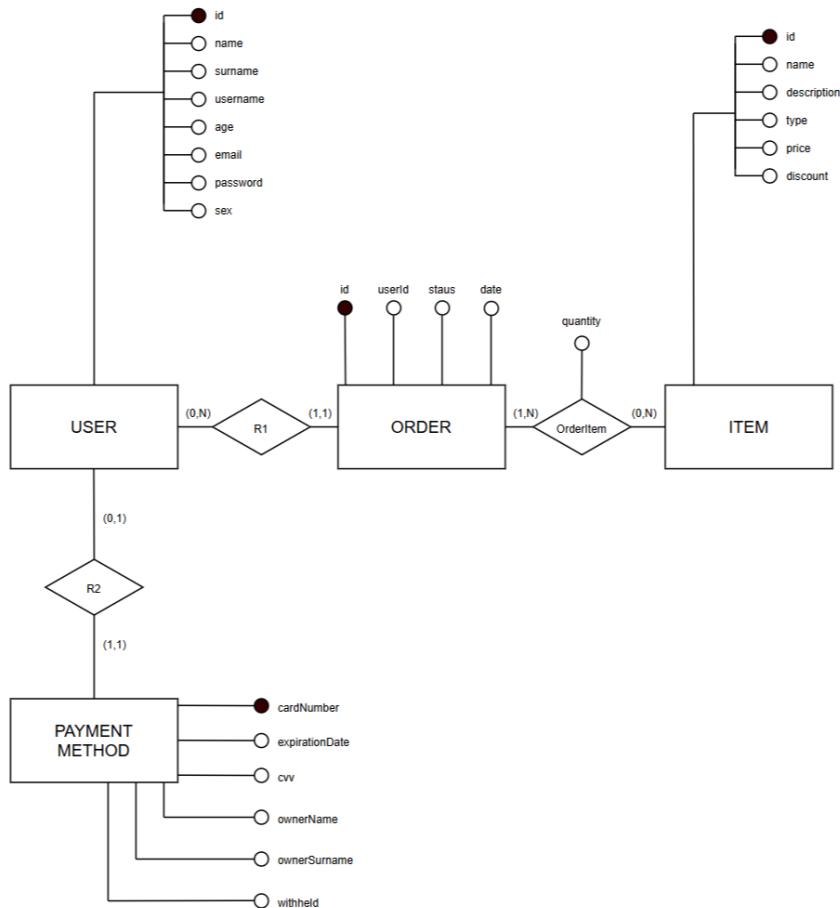


Figure 14: ER Diagram

```

User (id, name, surname, username, age, email, password, sex, paymentMethod)
PaymentMethod (cardNumber, expirationDate, cvv, ownerName, ownerSurname, withheld)
Item (id, name, description, type, price, discount)
Order (id, userid, status, date)
OrderItem (orderid, itemid, quantity)

```

Figure 15: Relational Model

2.6 Navigation Diagram

Il diagramma di navigazione (figura 16) descrive la struttura dell'applicazione, suddividendo le funzionalità tra utenti e amministratori. Dalla schermata iniziale, gli utenti possono accedere, registrarsi o entrare come admin. L'area utente permette di visualizzare il menu, gestire il carrello, controllare ordini e modificare il profilo. Gli amministratori hanno accesso a strumenti per gestire utenti, modificare il menu, monitorare ordini e intervenire sul database. Ogni sezione include sempre la possibilità di tornare indietro, garantendo un flusso intuitivo. Il sistema è progettato per coprire l'intero ciclo degli ordini, dall'acquisto alla gestione amministrativa.

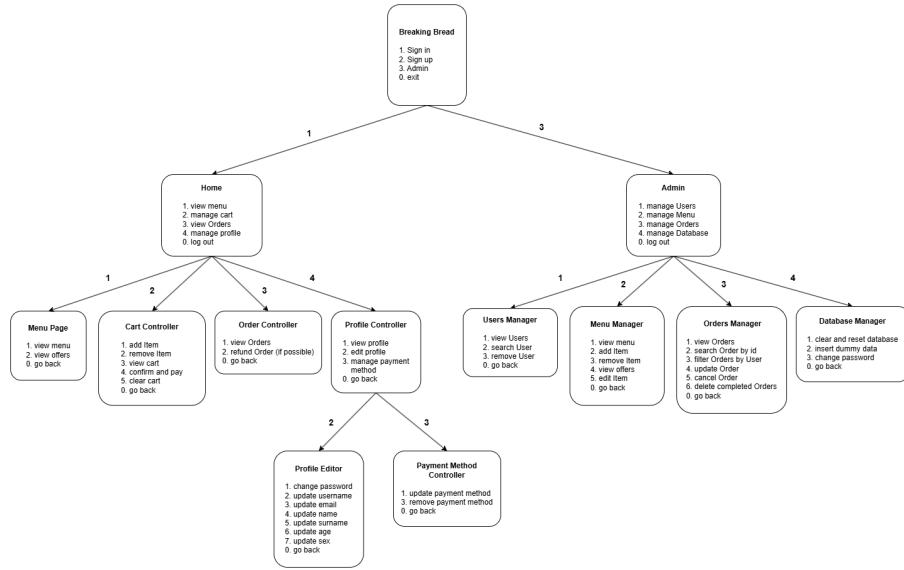


Figure 16: Navigation Diagram

2.7 Directories

Il progetto è strutturato in varie directories, divise a seconda del loro scopo.
Le principali sono:

- **docs**: contiene il report e i vari diagrammi del programma
- **lib**: contiene le varie librerie necessarie al programma
- **src/main**: contiene le classi Java
- **src/test**: contiene le classi per lo Unit Testing

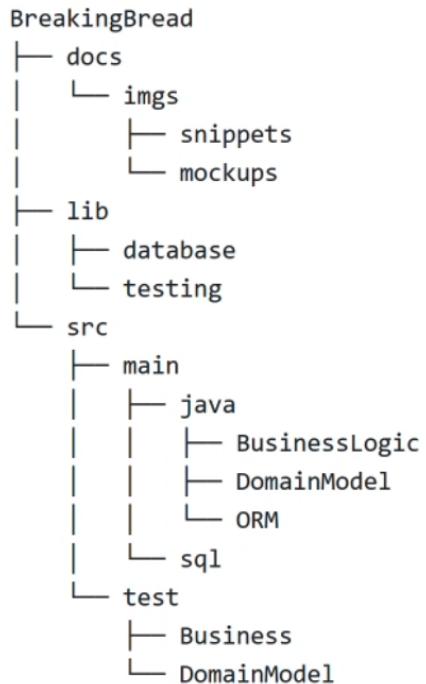


Figure 17: Directories

3 Implementazione

3.1 Domain Model

3.1.1 User

La classe "User" rappresenta l'entità utente, definito sugli attributi: id (univoco per ogni utente), name, surname, username, age, sex, email, password, paymentMethod(riferimento alla classe amonima), cart (ArrayList contenente id degli item).

3.1.2 Item

La classe "Item" rappresenta l'entità pietanza, definita sugli attributi: itemId (univoco per ogni pietanza), name, type, description, prizediscountPercentage.

3.1.3 PaymentMethod

La classe "PaymentMethod" rappresenta il metodo di pagamento, in questo caso una carta di credito. Definita sugli attributi: ownerName, ownerSurname, cardNumber, cardExpirationDate, cardCVV, withheld. Questa classe implementa i metodi :

- **pay()** il quale prende fra i parametri un riferimento a "order" da cui ricava la il costo totale dell'ordine e si accerta che vengano inseriti correttamente dall'utente i codici per validare il pagamento.
- **refund()** questo metodo verifica che sull'ordine che riceve fra i parametri si possa effettuare un rimborso, basandosi sull'attributo status di "Order".

3.1.4 Order

La classe "Order" rappresenta un ordine, è definito sugli attributi: orderId(univoco per ogni ordine), status, user, Date, items(ArrayList di "Item"),

3.1.5 Order Factory

La classe OrderFactory offre un metodo statico `createOrder()` per generare oggetti Order già validati, controllando che tutti i parametri richiesti (ID, utente, prodotti, stato e data) siano presenti. Questo approccio standardizza la creazione degli ordini (figura 18)

```

public class OrderFactory {
    public static Order createOrder(int orderId, User user, ArrayList<Item> items, String
status, String date) {
        if (user == null || items == null || status == null || date == null) {
            return null;
        }
        Order order = new Order();
        order.setOrderId(orderId);
        order.setUser(user);
        order.setItems(items);
        order.setStatus(status);
        order.setDate(date);
        return order;
    }
}

```

Figure 18: OrderFactory

3.2 Business Logic

3.2.1 LogInController

Questa classe si occupa di gestire il "Log in" permettendo l'accesso o la registrazione a utenti comuni, mentre garantisce l'accesso all'admin alla sua area riservata

3.2.2 UserManageOrderController

Lo scopo di questa classe è permettere agli utenti di gestire i propri ordini, sia quelli passati che quello corrente. La gestione è possibile grazie ai metodi: `addltem()`, `removeltem()`, `viewCurrentOrder()`, `confirmCurrentOrder()`, `cancelCurrentOrder()`, `ViewOrders()`, `cancelOrder()` i quali agiscono solo su gli ordini correnti tranne per gli ultimi due che agiscono su ordini passati,

3.2.3 UserViewMenuController

Questa classe permette di visualizzare il menu, un lista di Item, e il menu delle offerte, una lista di Item selezionati fra quelli che hanno il campo "discountPercentage" diverso da zero.

3.2.4 UserProfileController

Questa classe permette all'utente di controllare e gestire la propria area personale, aggiornando dati vecchi o gestendo metodi di pagamento, necessari per poter effettuare un ordine.

3.2.5 AdminDatabaseOptionsController

la AdminDatabaseOptionsController permette all'utente Admin di gestire le propri credenziali o di svuotare il database.

3.2.6 AdminUserControl

In questa area si permette all'utente admin di gestire tutti gli utenti registrati alla applicazione. Qui l'utente Admin ha la possibilità di avere una lsita di tutti gli utenti, grazie al metodo viewUsers().

3.2.7 Admin Order Controller

Come la precedente classe anche questa è pensata per l'utente Admin, qui l'admin ha infatti la possibilità di gestire gli ordini degli utenti attraverso metodi come viewOrders() o cancelOrder().

3.2.8 AdminItemController

Questa classe come il nome suggerisce permette sempre all'utente Admin di gestire gli Item che che utenti comuni si ritroveranno nel menu.

3.3 Object-Relational Mapping

Il package `main.java.ORM` contiene le classi che si occupano dell'Object-Relational Mapping, ovvero delle operazioni di scrittura e lettura di dati nel database.

3.3.1 ConnectionManager

La classe ConnectionManager, la quale implementa un design pattern "Singleton" in quanto è richiesta una sola istanza di tale, si occupa dell'importante compito di gestire le connessioni con il database tramite il metodo `getConnection()`. Questa classe contiene, inoltre, tutti i dati necessari per effettuare l'accesso al database, ovvero l'URL, username e password.

```
public Connection getConnection() throws SQLException, ClassNotFoundException {
    if (connection == null || connection.isClosed()) {
        try {
            Class.forName("org.postgresql.Driver");
            connection = DriverManager.getConnection(URL, USERNAME, PASSWORD);
            System.out.println("Connection to database established successfully!");
        } catch (SQLException | ClassNotFoundException e) {
            System.err.println("Error: " + e.getMessage());
            throw e;
        }
    }
    return connection;
}
```

Figure 19: Metodo `getConnection()`

3.3.2 UserDAO

La classe UserDAO gestisce gli utenti e la connessione della classe User al database. Questa classe implementa vari metodi, tra cui `addUser()`, `removeUser()`, `verifyPassword()`, alcuni metodi per ottenere i dati da uno o più utenti, come `getUser()` e `getAllUsers()`, e vari metodi per l'aggiornamento dei dati personali e di accesso dell'utente.

3.3.3 AdminDAO

La classe AdminDAO si occupa di effettuare le operazioni di gestione del database a disposizione dell'Admin. In questa classe abbiamo il metodo `clearDatabase()`, il quale si occupa dell'esecuzione di una stringa SQL che ripristina il database, e il metodo `generateDummyData()`, il quale serve per generare dei dati fittizi all'interno del database.

```

public UserDAO() {
    try {
        this.connection = ConnectionManager.getInstance().getConnection();
    } catch (SQLException | ClassNotFoundException e) {
        System.err.println("Error: " + e.getMessage());
    }
}

public void addUser(String name, String surname, String username, int age, String sex, String email, String password) throws SQLException {
    String sql = "INSERT INTO Users (name, surname, username, age, sex, email, password) VALUES (?, ?, ?, ?, ?, ?, ?)";
    try (PreparedStatement preparedStatement = connection.prepareStatement(sql)) {
        preparedStatement.setString(1, name);
        preparedStatement.setString(2, surname);
        preparedStatement.setString(3, username);
        preparedStatement.setInt(4, age);
        preparedStatement.setString(5, sex);
        preparedStatement.setString(6, email);
        preparedStatement.setString(7, password);

        preparedStatement.executeUpdate();
        System.out.println("User added successfully.");
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    }
}

```

Figure 20: Costruttore e metodo `addUser()`

3.3.4 PaymentMethodDAO

La classe PaymentMethodDAO gestisce l'inserimento e la cancellazione di metodi di pagamento all'interno del database tramite, rispettivamente, i metodi `addPaymentMethod()` e `removePaymentMethod()`.

```

public void addPaymentMethod(PaymentMethod paymentMethod) {
    String sql = "INSERT INTO PaymentMethod (cardNumber, expirationDate, cvv, ownerName, ownerSurname,
withheld) VALUES (?, ?, ?, ?, ?, ?)";

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setString(1, paymentMethod.getCardNumber());
        pstmt.setString(2, paymentMethod.getCardExpiryDate());
        pstmt.setString(3, paymentMethod.getCardCVV());
        pstmt.setString(4, paymentMethod.getOwnerName());
        pstmt.setString(5, paymentMethod.getOwnerSurname());
        pstmt.setFloat(6, paymentMethod.getWithheld());

        pstmt.executeUpdate();
        System.out.println("Payment method added successfully.");
    } catch (SQLException e) {
        System.err.println("Error adding payment method: " + e.getMessage());
    }
}

public void removePaymentMethod(String cardNumber) {
    String sql = "DELETE FROM PaymentMethod WHERE cardNumber = ?";

    try (PreparedStatement pstmt = connection.prepareStatement(sql)) {
        pstmt.setString(1, cardNumber);

        int rowsAffected = pstmt.executeUpdate();
        if (rowsAffected > 0) {
            System.out.println("Payment method with card number " + cardNumber + " removed successfully.");
        } else {
            System.out.println("No payment method found with card number " + cardNumber + ".");
        }
    } catch (SQLException e) {
        System.err.println("Error removing payment method: " + e.getMessage());
    }
}

```

Figure 21: Metodi `addPaymentMethod()` e `removePaymentMethod()`

3.3.5 ItemDAO

La classe UserDAO gestisce gli item (pietanze) e la connessione della classe Item al database. Tra i vari metodi implementati da questa classe ci sono `addItem()`, `removeItem()`, `getItem()` (che ritorna un item dato l'id), `getAllItems()` (che ritorna tutti gli items) e `getAllDiscountedItems()` (che ritorna tutti gli item aventi uno sconto). Ci sono, inoltre, vari metodi che permettono la modifica di alcuni attributi di Item, come `updateDescription()`, `updatePrice()` e `updateDiscount()`.

3.3.6 OrderDAO

La classe OrderDAO si occupa di gestire sia la tabella Orders che la tabella OrdersItem (relazione 1:N, che collega un ordine a 1 o più Items). I metodi che implementa la classe sono: `createOrder()`, `removeOrder()`, `removeCompletedOrders()` (il quale rimuove tutti gli ordine con status == "Completed"), `getOrder()`, `getAllOrders()`, `getOrdersByUser()` e `updateStatus()`. Tutti i metodi "getter" richiedono l'utilizzo di un altro DAO, in particolare dell'ItemDAO. Il metodo `getAllOrdersByStatus()` riceve in ingresso una variabile booleana e restituisce tutti gli ordini completati se la variabile è "true" oppure tutti gli ordini non ancora completati se la variabile è "false". Un altro metodo che richiede particolarmente attenzione è il metodo `createOrderItemFromIds()`: dato un orderId e una lista di itemId, questo metodo popola la tabella OrdersItem inserendo per ciascun articolo il numero di volte che compare nella lista come quantità (quantity); in breve, inserisce nel database gli articoli relativi a un ordine.

```

public void createOrderItemFromIds(int orderId, ArrayList<Integer> itemsId) throws SQLException,
ClassNotFoundException {
    PreparedStatement pStatement = null;

    Map<Integer, Integer> itemCount = new HashMap<>();
    for (Integer itemId : itemsId) {
        itemCount.put(itemId, itemCount.getOrDefault(itemId, 0) + 1);
    }

    String insertStatement = "INSERT INTO OrdersItem (orderid, itemid, quantity) VALUES (?, ?, ?)";

    try {
        pStatement = con.prepareStatement(insertStatement);

        for (Map.Entry<Integer, Integer> entry : itemCount.entrySet()) {
            pStatement.setInt(1, orderId);
            pStatement.setInt(2, entry.getKey());
            pStatement.setInt(3, entry.getValue());
            pStatement.addBatch();
        }

        pStatement.executeBatch();
        System.out.println("Items inserted/updated successfully in Order.");
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    } finally {
        if (pStatement != null) pStatement.close();
    }
}

```

Figure 22: Metodo `createOrderItemFromIds()`

```

public ArrayList<Order> getAllOrdersByStatus(boolean completed) throws SQLException, ClassNotFoundException
{
    PreparedStatement orderStatement = null;
    PreparedStatement itemsStatement = null;
    ResultSet orderResult = null;
    ResultSet itemsResult = null;
    ArrayList<Order> orders = new ArrayList<>();
    String s = "Completed";

    try {
        // Recupero i dettagli dell'ordine
        String orderSQL;
        if (completed) {
            orderSQL = String.format("SELECT * FROM Orders WHERE status = ?");
        } else {
            orderSQL = String.format("SELECT * FROM Orders WHERE status <> ?");
        }
        orderStatement = con.prepareStatement(orderSQL);
        orderStatement.setString(1, s);
        orderResult = orderStatement.executeQuery();

        while (orderResult.next()) {
            ArrayList<Item> items = new ArrayList<>();
            int orderId = orderResult.getInt("id");
            int userId = orderResult.getInt("userid");
            String status = orderResult.getString("status");
            String date = orderResult.getDate("date").toString();

            User user = new UserDAO().getUser(userId);

            // 2. Recupero gli item associati all'ordine
            String itemsSQL = String.format("SELECT * FROM OrdersItem WHERE orderid = %d", orderId);
            itemsStatement = con.prepareStatement(itemsSQL);
            itemsResult = itemsStatement.executeQuery();

            while (itemsResult.next()) {
                int itemId = itemsResult.getInt("itemid");
                for (int i = 0; i < itemsResult.getInt("quantity"); i++) {
                    items.add(new ItemDAO().getItem(itemId));
                }
            }

            // 3. Creazione dell'oggetto Order con la lista di item
            orders.add(OrderFactory.createOrder(orderId, user, items, status, date));
        }
    } catch (SQLException e) {
        System.err.println("Error: " + e.getMessage());
    } finally {
        if (orderResult != null) orderResult.close();
        if (itemsResult != null) itemsResult.close();
        if (orderStatement != null) orderStatement.close();
        if (itemsStatement != null) itemsStatement.close();
    }
}

return orders;
}

```

Figure 23: Metodo `getAllOrdersByStatus()`

3.4 Database

Il database è stato progettato in modo da rispettare i requisiti del sistema e contiene le tabelle `Users`, `PaymentMethod`, `Item`, `Orders` e `OrdersItem` (figura 14). I file relativi al database sono contenuti in `src/main/sql` e sono:

- `database.sql`, il cui compito è quello di cancellare le vecchie tabelle (tramite il comando `DROP TABLE IF EXISTS`) e ricrearne nuove (vuote)
- `dummydata.sql`, il quale contiene le istruzioni SQL per l'inserimento di dati fintizi nel database, utili effettuare test più mirati e particolari

```
-- Elimina le tabelle se esistono
DROP TABLE IF EXISTS Item, OrdersItem, Orders, PaymentMethod, Users
CASCADE;
-- Crea la tabella Users senza la chiave esterna a PaymentMethod
CREATE TABLE Users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    surname VARCHAR(50),
    username VARCHAR(50) UNIQUE NOT NULL,
    age INT CHECK (age >= 0),
    email VARCHAR(100) UNIQUE NOT NULL,
    password TEXT NOT NULL,
    sex CHAR(1),
    paymentMethod VARCHAR(16) -- Colonna senza vincolo di chiave esterna
);

-- Crea la tabella PaymentMethod
CREATE TABLE PaymentMethod (
    cardNumber VARCHAR(16) PRIMARY KEY,
    expirationDate VARCHAR(5) NOT NULL,
    cvv VARCHAR(4) NOT NULL,
    ownerName VARCHAR(50) NOT NULL,
    ownerSurname VARCHAR(50) NOT NULL,
    withheld FLOAT DEFAULT 0.0
);

-- Aggiungi la chiave esterna a Users dopo aver creato PaymentMethod
ALTER TABLE Users
ADD CONSTRAINT paymentMethod_fk
FOREIGN KEY (paymentMethod)
REFERENCES PaymentMethod(cardNumber)
ON DELETE SET NULL;
```

Figure 24: Cancellazione di tutte le tabelle e creazione della tabella `Users` e `PaymentMethod`

4 Testing

Dato che il progetto è strutturato in tre package, anche il testing segue questa pratica, i test sono infatti divisi in package: DomainModelTest, BusinessLogicTest, manca a questo proposito un ORMLTest, la cui mancanza è dovuta a che già nella BusinessLogicTest se ne fa un ampio testing.

4.1 BusinessLogicTest

Di seguito sono riportati i test della Business Logic, in viola sono riportati i nomi delle classi, mentre in arancione i metodi di test che queste implementano.

```
AdminUserControllerTest
    removeUserTest()
    searchUserTest()

AdminOrderControllerTest
    deleteCompletedOrdersTest()
    CancelOrderTest()
    viewOrdersByUserTest()

AdminItemControllerTest
    viewOffersTest()
    editItemTest()
    removeItemFromMenuTest()

UserManagerOrderControllerTest
    confirmCurrentOrderTest()
    cancelOrderTest()
    addItemTest()
    viewCurrentOrderTest()

LoginControllerTest
    loginTest()
    RegisterTest()

UserProfileControllerTest
    setPaymentMethod()
    removePaymentMethodTest()
```

Figure 25: BusinessLogicTest

Tutte le classi di test implementano un metodo `setUp()` e uno `tearDown()`, questo con lo scopo di mettere i metodi effettivi di test nelle condizioni e con le risorse adeguate per verificare il codice. In modo particolare la `setUp()` si occupa di allocare gli oggetti e aggiungere occorrenze nel database da testare, mentre la `tearDown()` rimuove dal database le occorrenze usate nei test (figura 26).

Questo lo si fa per lasciare intonzo il database dopo l'esecuzione dei test.

Di seguito sono mostrati i test della LoginControllerTest, nei quali si verifica il corretto funzionamento dei metodi `login()` e `register()` della classe LoginController (figura 27). Nella (figura 28) è invece riportato il test effettuato sulla classe AdminUserController, nella quale si verificano i metodi `removeUser()` e `searchUser()`

```

@BeforeEach
void setUp() throws SQLException, ClassNotFoundException{
    userDAO = new UserDAO();
    userDAO.addUser("nameTest", "surnameTest", "usernameTest", 25, "M", "test@email.com",
"password","1234567812345678","05/26", "123",0.01F,"nameTest","surnameTest");
    user = userDAO.getUser("usernameTest");
    userManagerOrderController = new UserManagerOrderController(user);

    itemDAO = new ItemDAO();
    itemId1 =itemDAO.addItem("itemTest1","descriptionTest","typeTest",10.0f,0);
    itemId2 =itemDAO.addItem("itemTest2","descriptionTest","typeTest",10.0f,0);
    itemId3 =itemDAO.addItem("itemTest3","descriptionTest","typeTest",10.0f,0);

    orderDAO = new OrderDAO();
}

@AfterEach
void tearDown() throws SQLException, ClassNotFoundException {
    userDAO.removeUser("usernameTest");
    itemDAO.removeItem(itemId1);
    itemDAO.removeItem(itemId2);
    itemDAO.removeItem(itemId3);
}
}

```

Figure 26: Esempio Uso SetUP, TearDown

```

@Test
void loginTest() throws SQLException ,ClassNotFoundException {
    User RealUser= loginController.login("usernameTest","password");
    assertNotNull(RealUser);

    User UserWithWrongPassword= loginController.login("usernameTest","wrongPassword");
    assertNull(UserWithWrongPassword);

    User FakeUser= loginController.login("fakeUser","FakePassword");
    assertNull(FakeUser);
}

@Test
void RegisterTest() throws SQLException ,ClassNotFoundException {
    User User=
loginController.register("pino","gini","pinos","pino@gmail.com",33,"password","m");
    assertNotNull(User);
    assertEquals("pinos",User.getUsername());
    userDAO.removeUser("pinos");
}

```

Figure 27: LoginControllertest

```

    @Test
    void searchUserTest() throws SQLException, ClassNotFoundException {
        ArrayList<User> user = adminUserController.searchUser("usernameTest");
        assertNotNull(user.get(0));
        tearDown();
    }

    @Test
    void removeUserTest() throws SQLException, ClassNotFoundException {
        adminUserController.removeUser("usernameTest");
        assertNull(userDAO.getUser("usernameTest"));
    }
}

```

Figure 28: AdminUserControllerTest

4.2 DomainModelTest

Anche il package DomainModel è stato testato, Con lo scopo di verificarne la correttezza, sono stati eseguiti i seguenti test (figura 29), i test si sono concentrati sulle classi Order e PaymentMethod dato che queste fra le classi del DomainModel sono quelle che implementano metodi e che quindi corrono il rischio di danneggiarsi.

```

OrderTest
    testGetItems()
    testGetTotal()
    testGetOrderId
    testGetTotalWithMultipleSameItems()
    testGetDate()
    test GetUser()
    testGetTotalWithSingleItem()
    testGetStatus()
    testGetTotalWithEmptyItems()
    testSetItems()

PaymentMethodTest
    testPayCancelledByUser()
    testRefundSuccess()
    testPaySuccess()
    testPayWithWithheld()
    testPayWrongCVV()
    testRefundFailed()

```

Figure 29: DomainModelTest

A seguire sono riportati alcuni dei test sulla classe "Order", in (figura 30) sono riportati i test : `testGetTotal()`, `testGetTotalWithEmptyItems()`, `testGetTotalWithSingleItem()`.

```

@Test
void testGetTotal() {
    // Calculate expected total manually:
    // Cheeseburger: 5.99 - 0% = 5.99
    // Chicken Nuggets: 3.99 - 10% = 3.591
    // Soda: 2.49 - 0% = 2.49
    // Ice Cream: 1.99 - 5% = 1.8905
    // Total = 5.99 + 3.591 + 2.49 + 1.8905 ≈ 13.9615
    float expectedTotal = 5.99f + 3.591f + 2.49f + 1.8905f;
    float actualTotal = order.getTotal();

    assertEquals(expectedTotal, actualTotal, 0.001);
}

@Test
void testGetTotalWithEmptyItems() {
    order.setItems(new ArrayList<>());
    assertEquals(0.0f, order.getTotal());
}

@Test
void testGetTotalWithSingleItem() {
    ArrayList<Item> singleItem = new ArrayList<>();
    singleItem.add(new Item(6, "Family Meal", "2 burgers, nuggets and fries", "Combo",
15.99f, 15));
    order.setItems(singleItem);

    float expected = 15.99f * 0.85f; // 15% discount
    assertEquals(expected, order.getTotal(), 0.001);
}

```

Figure 30: OrderTest