

# Time Saving - Progettazione e Sviluppo di una Web Application in Java EE per la gestione di ordini in locali pubblici

Riccardo Papucci, Alessandro Baroni

20 Aprile 2018



# Indice

<b>1</b>	<b>Introduzione</b>	<b>2</b>
<b>2</b>	<b>Requisiti</b>	<b>3</b>
2.1	Requisiti di Livello Utente . . . . .	3
2.2	Requisiti di Dominio . . . . .	4
2.3	Requisiti Funzionali . . . . .	4
<b>3</b>	<b>Progettazione e sviluppo</b>	<b>5</b>
3.1	Casi d'uso . . . . .	5
3.2	Modello concettuale . . . . .	6
3.3	Page navigation diagram . . . . .	6
3.3.1	Mockups . . . . .	10
3.4	Domain Model . . . . .	12
<b>4</b>	<b>Architettura Implementativa</b>	<b>14</b>
4.1	Domain Logic . . . . .	14
4.1.1	Domain Model . . . . .	14
4.1.2	DAO . . . . .	17
4.1.3	Business Logic . . . . .	20
<b>5</b>	<b>Test</b>	<b>24</b>
<b>6</b>	<b>Interfaccia Web</b>	<b>30</b>
6.1	HomePage . . . . .	30
6.2	Pagine di Gestione del Locale . . . . .	33
6.3	Pagine di Gestione dell'Ordine . . . . .	34
<b>7</b>	<b>Considerazioni finali e sviluppi futuri</b>	<b>38</b>

# Capitolo 1

## Introduzione

Lo scopo dell'elaborato è quello di progettare e realizzare un sistema informatico che permetta di gestire gli ordini di clienti in locali pubblici, tenendo in considerazione anche i diversi ruoli attribuiti agli utenti. Il lavoro si è sviluppato attraverso molteplici fasi: inizialmente l'analisi dei requisiti, una fase di progettazione vera e propria, una di Test e infine un primo approccio all'implementazione dell'interfaccia Web. Questa relazione è articolata in modo da ripercorrere esattamente i passi descritti. Saranno dettagliate le varie fasi e riportati i procedimenti che hanno portato allo sviluppo del sistema.

# Capitolo 2

## Requisiti

### 2.1 Requisiti di Livello Utente

- Il sistema, installato in locali pubblici, deve permettere la gestione di **ordini** di bevande e cibo effettuati da **clienti** ed evasi dagli **operatori** del locale, deve permettere al cliente di **modificare** il proprio ordine in caso di sbaglio, prima del **pagamento**.  
Il sistema deve permettere all'operatore di **inviare una notifica** al cliente che ha effettuato l'ordine, una volta che quest'ultimo è stato evaso.
- Il cliente accede al servizio effettuando un login tramite username e password. In caso di primo utilizzo del servizio, è necessaria la **registrazione**, attraverso l'apposita pagina.  
Il cliente deve poter scegliere e visualizzare i dettagli del **locale** in cui vuole effettuare l'ordine da una lista di pub vicini a lui, deve poter visualizzare e scegliere i **prodotti** a partire da un **menù**, deve essere in grado di **monitorare lo stato** del suo ordine.
- L'operatore deve avere i diritti di utente amministratore, con i quali deve poter **modificare i dettagli del locale** in cui lavora, visualizzare tutti gli ordini effettuati nel locale e modificare il menù del locale, vale a dire **effettuare il CRUD dei prodotti**.

## 2.2 Requisiti di Dominio

- Un **Ordine** deve avere: il nominativo del cliente che lo effettua, l'identificativo della lista dei prodotti ordinati, lo stato (arrivato, in esecuzione, concluso), gli operatori che lo prendono in carico, il locale in cui è stato effettuato.
- I dati di un **Utente** sono: username, password, nome, cognome, indirizzo, numero di telefono, indirizzo e-mail
- Un **Cliente** deve avere i dati della carta di credito con cui effettuerà i pagamenti.
- Un **Operatore** deve avere: il tipo (barista o cuoco) e l'identificativo del locale in cui lavora.
- Un **Locale** deve contenere: il nome, la partita iva, l'indirizzo, una descrizione e il numero di telefono.
- Un **Menù** deve avere una descrizione e deve essere associato a un pub e ai prodotti che ne fanno parte.
- Un **Prodotto** deve avere una descrizione, una foto, un tipo che determini se si tratta di una bevanda (**Drink**) o di cibo (**Food**), il tempo di produzione (per un'eventuale stima dei tempi dell'ordine) ed un prezzo.
- Una **Lista** deve avere, per ogni ordine, tutti i prodotti che lo compongono e la loro quantità.

## 2.3 Requisiti Funzionali

È richiesto l'uso di JPA per la persistenza dei dati in un DBMS relazionale e di CDI per la gestione dei controller. Per quanto riguarda la parte di interfaccia grafica si richiede l'uso di JSF per testare i bean creati attraverso CDI.

# Capitolo 3

## Progettazione e sviluppo

### 3.1 Casi d'uso

In figura 3.1 sono presentati i casi d'uso emersi dallo studio dei requisiti funzionali. Principalmente sono stati individuati due attori che derivano dal ruolo generico **User**: il **cliente** e l'**operatore**. Quest'ultimo si distingue in altri due attori, il **barista** e il **cuoco**.

Qualsiasi utente del sistema deve effettuare il login per poter utilizzare l'applicazione. In questa fase viene effettuata la distinzione tra cliente e operatore: il **cliente**, prima di effettuare il login, deve essersi registrato nel sistema a differenza dell'**operatore**, che, essendo un dipendente del locale, avrà già i dati inseriti nel sistema. Il **cliente**, dopo aver effettuato il login, deve poter scegliere il locale in cui vuole effettuare la prenotazione e devono essere resi disponibili i dettagli del pub. Una volta scelto il locale, l'utente deve poter fare due cose: **effettuare un ordine**, dopo aver visualizzato il menù del locale e aver scelto i prodotti da ordinare, e **visualizzare gli ordini effettuati** in modo da poter controllare quali prodotti abbia acquistato e monitorare lo stato del suo ordine. Il sistema deve anche permettere all'utente di visualizzare i dettagli dei prodotti del menù e deve dargli la possibilità di modificare l'ordine nel caso di errore.

I casi d'uso previsti per l'**operatore** sono ben diversi: partendo dal presupposto che egli sia dipendente di un solo locale, non ha la possibilità di sceglierlo; deve essere in grado di modificare il menù, in particolare, deve poter inserire nuovi prodotti, modificare quelli esistenti ed eliminare quelli che desidera; deve poter modificare le informazioni relative al locale in cui lavora

e visualizzare tutti gli ordini effettuati dai clienti. Dopo che un ordine viene preso in carico dall'operatore, egli deve poter cambiare il suo stato in modo tale da consentire all'utente di sapere se l'ordine da lui effettuato sia stato preso in carico, sia in esecuzione oppure sia concluso. In questo ultimo caso il sistema deve inviare una notifica all'utente per avvertirlo che può andare a ritirare il suo ordine. La differenza tra cuoco e barista è dettata dal tipo dei prodotti dell'ordine di cui si occupano. Il **cuoco**, infatti, potrà visualizzare e gestire solo la parte dell'ordine relativa ai cibi, mentre il **barista** soltanto quella dei drink.

## 3.2 Modello concettuale

In figura 3.2 è presentato il modello concettuale. Uno user può essere un Cliente o un Operatore, a sua volta distinto in cuoco o barista. Ogni operatore solitamente invia molte notifiche, una per ogni ordine concluso, mentre un cliente ne riceve tante quanti sono gli ordini che ha effettuato. Un operatore ha il riferimento ad un solo locale, quello in cui lavora, mentre un cliente deve poter scegliere tra molti locali. Ogni locale avrà un solo menù che contiene molti prodotti, i quali possono essere di due tipologie: cibo o drink. Quando viene creato un ordine, un cliente crea una lista, con riferimento a 1 o molti prodotti. L'ordine quindi, oltre a avere il riferimento al cliente che l'ha creato, deve far riferimento a 1 o 2 operatori, nel caso in cui i prodotti presenti nell'ordine siano di un solo tipo o di entrambi. Infine l'ordine fa riferimento anche ad uno stato, che varia quando l'ordine viene ricevuto, quando viene preso in carico e quando viene evaso. Lo stato finale però deve essere distinto in altri tre casi, che identificano l'ordine concluso solo per i drink, concluso solo per il cibo o concluso per entrambi i tipi di prodotto. Questo permette al cliente di poter ritirare il drink senza dover necessariamente aspettare il cibo o viceversa (nel caso in cui il cliente ordinasse un pasto caldo, ad esempio, aspettare necessariamente i drink sarebbe un disagio).

## 3.3 Page navigation diagram

In figura 3.3 viene presentato il page navigation diagram del sistema, costruito a partire dai casi d'uso.

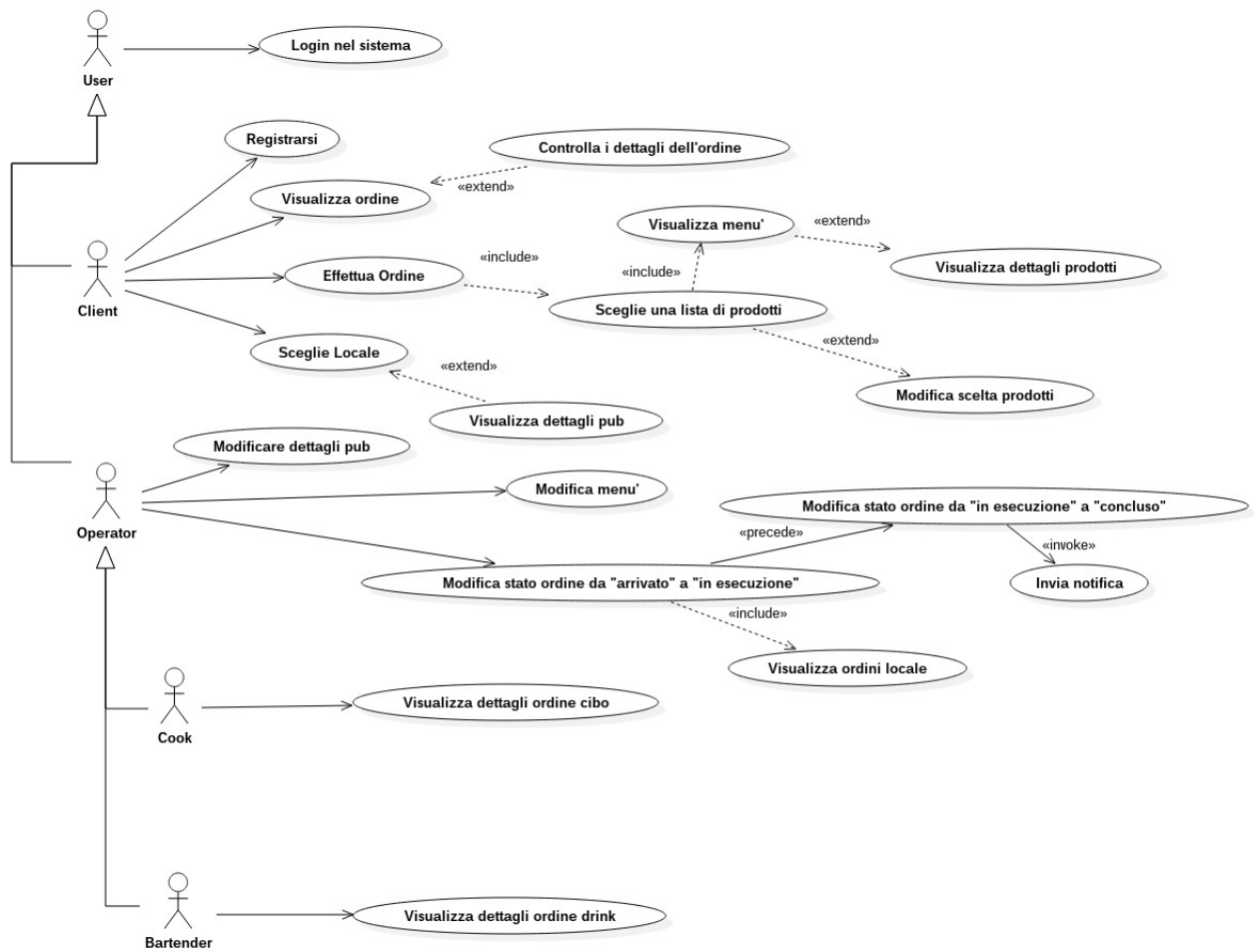


Figura 3.1: Use case diagram



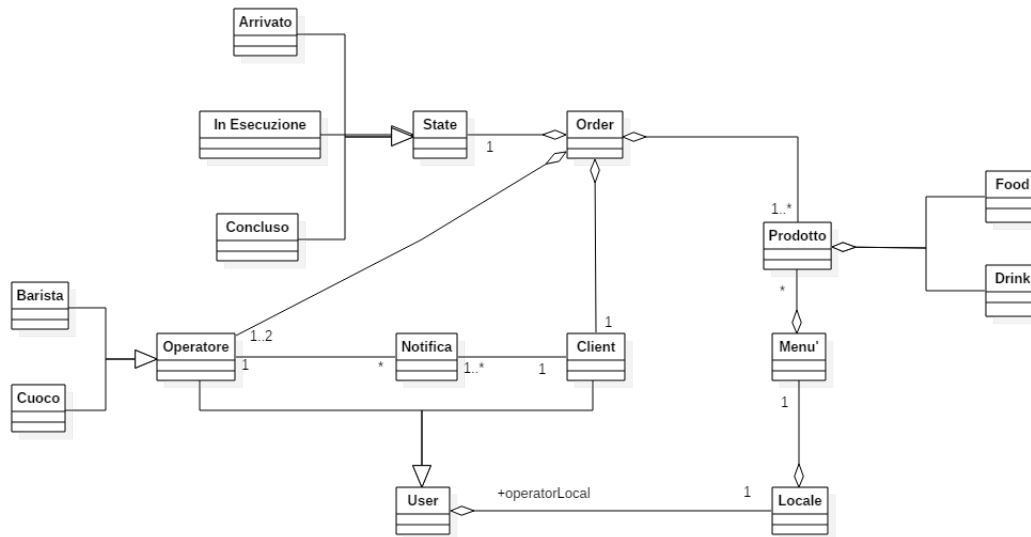


Figura 3.2: Modello concettuale

La Welcome Page dell'applicazione è la **Login Page**, in cui un utente può effettuare il login inserendo il suo id e la sua password o raggiungere la Sign up Page, qualora non fosse ancora registrato.

Nella **Sign Up Page** l'utente può inserire i propri dati e registrarsi. In questo caso verrà creato un Cliente e non un operatore. Una volta che la registrazione è avvenuta con successo, il cliente viene reindirizzato nella Login Page in cui può eseguire il login ed essere indirizzato verso la Homepage.

Da qui in poi ogni pagina ha il pulsante Back per tornare alla pagina precedente, il pulsante Home per andare alla HomePage, il pulsante Logout per tornare alla pagina di Login e il pulsante View Orders che indirizza l'utente verso la OrderList Page. Le frecce per questi indirizzamenti nel grafico sono state omesse per semplicità e le funzionalità di ogni pagina cambiano a seconda del tipo di utente che ha effettuato il login, sia esso un Cliente o un Operatore.

Tramite la **Home Page** l'utente può essere indirizzato in altre pagine in cui può effettuare i casi d'uso preposti:

- Un cliente può visualizzare le informazioni relative ai pub a lui più vicini, può scegliere quello che desidera e visualizzarne il menù per

poter iniziare l'ordine, può visualizzare gli ordini che ha effettuato.

- Un operatore può scegliere se modificare il menu del pub in cui lavora, se visualizzare tutti i suoi ordini, oppure se vuole modificare i dettagli del locale.

Nella **Pub info Page**:

- Un cliente può solo visualizzare le informazioni relative al pub quali la descrizione, l'indirizzo, la partita IVA e la foto.
- Un operatore può, oltre che visualizzare, anche modificare le informazioni del pub.

Nella **Menu Page**:

- Un cliente può determinare la quantità dei prodotti che vuole acquistare e visualizzare i dettagli di ogni prodotto.
- Un operatore può modificare le informazioni dei prodotti che compongono il menù, può creare un nuovo prodotto o eliminare uno di quelli già presenti.

Nella **Product Page**:

- Un cliente può visualizzare i dettagli del prodotto scelto.
- L'operatore non ha accesso a questa pagina.

Nella **Summary Page**:

- Il cliente accede a questa pagina dopo aver scelto i prodotti che vuole acquistare. Qui vede riassunti i prodotti con le quantità ed il prezzo. A questo punto il cliente decide se vuole effettuare l'acquisto oppure modificare l'ordine. In questo ultimo caso viene reindirizzato verso la Menu Page, in cui può effettuare le modifiche.
- L'operatore non ha accesso a questa pagina, dato che non effettua ordini.

Nella **Order List Page**:

- Il cliente, dopo aver effettuato l'ordine, viene indirizzato in questa pagina dall'homepage oppure dal menù a tendina raggiungibile cliccando il pulsante in alto a destra di ogni altra pagina. Qui il cliente può visualizzare tutti gli ordini che ha effettuato e monitorare il loro stato, visualizzare il tempo stimato per la preparazione e andare a visualizzare il contenuto dell'ordine.
- Ogni operatore può modificare lo stato degli ordini da "Ordine pagato in attesa" a "Ordine in Esecuzione".
- L'operatore cuoco può vedere tutti gli ordini effettuati nel locale in cui è presente almeno un prodotto cibo. Può modificare lo stato dell'ordine da "Ordine in Esecuzione" a "Ordine cucina concluso Ordine drink in esecuzione" nel caso in cui ci sia almeno un drink nell'ordine che non è concluso dal barista oppure da "Ordine drink concluso Ordine cucina in esecuzione" a "Ordine concluso".
- Per l'operatore barista vale la stessa regola. Può vedere tutti gli ordini in cui è presente almeno un prodotto drink effettuati nel locale. Può modificare lo stato dell'ordine da "Ordine in Esecuzione" a "Ordine drink concluso Ordine cucina in esecuzione" nel caso in cui ci sia almeno un cibo nell'ordine che non è concluso dal cuoco oppure da "Ordine cucina concluso Ordine drink in esecuzione" a "Ordine concluso".

Nella **Product Detail Page**:

- Il cliente visualizza la lista dei prodotti acquistati nell'ordine scelto, insieme alla quantità e al prezzo.
- L'operatore cuoco visualizza tutti e soli i prodotti cibo dell'ordine scelto.
- L'operatore barista visualizza tutti e soli i prodotti drink dell'ordine scelto.

### 3.3.1 Mockups

Dopo aver creato il Page Navigation Diagram, è stato costruito un mockup per ogni pagina. In figura 3.4 sono presentati i mockups delle pagine principali dell'applicazione. Questi sono stati utili in fase di costruzione dell'interfaccia attraverso JSF.

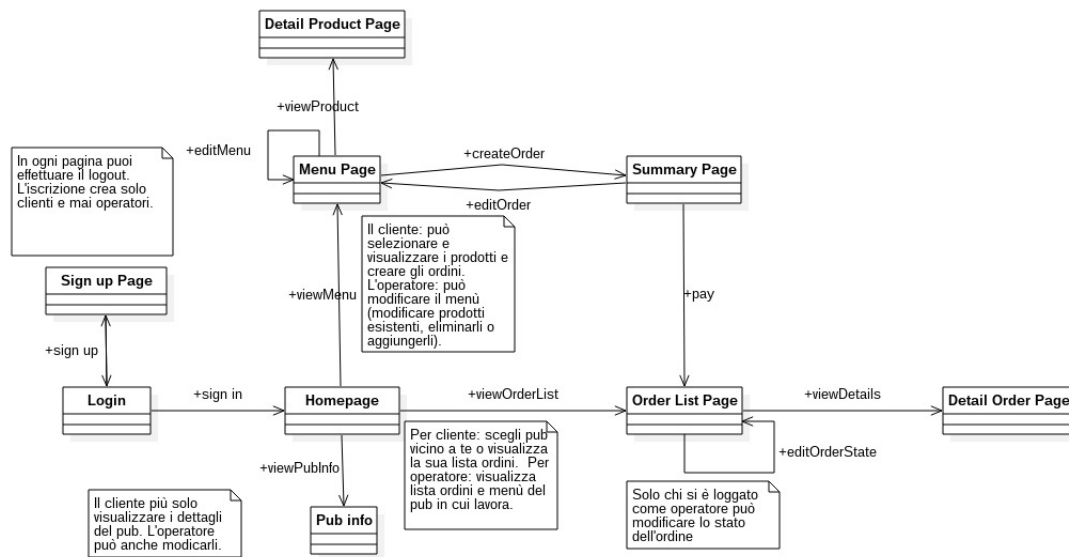


Figura 3.3: Page Navigation diagram

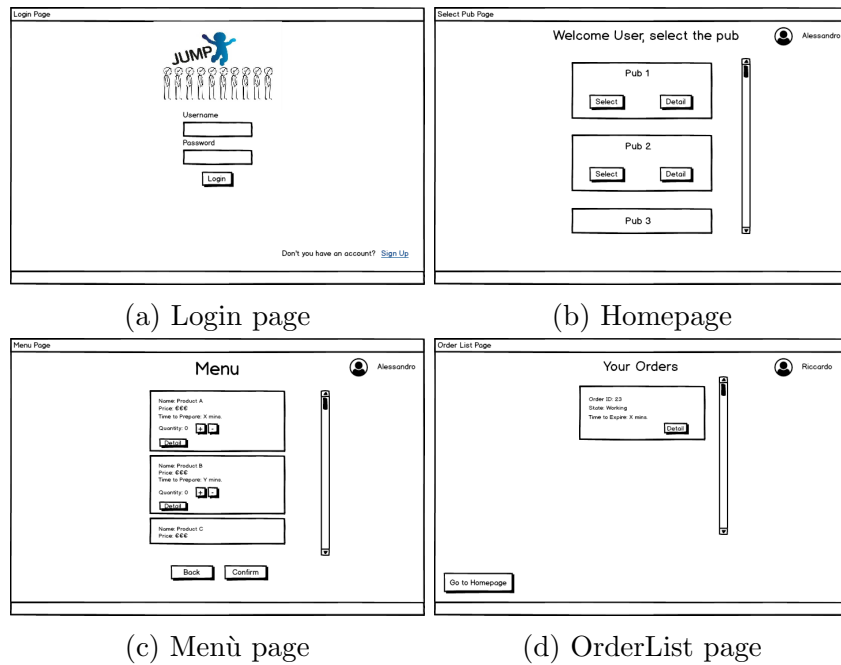


Figura 3.4: Mockups

## 3.4 Domain Model

Partendo dal modello concettuale è stato costruito il **modello di dominio** in figura 3.5, considerando le caratteristiche di Java EE e le annotazioni JPA.

- Le entità **Cliente** e **Operatore** ereditano dall'entità **User** e differiscono per pochi attributi. Infatti il Cliente avrà i dati di pagamento che l'Operatore non ha, dato che non esegue gli acquisti. L'operatore ha invece un tipo (barista o cuoco) e il riferimento al locale in cui lavora. Gli altri attributi di User sono i classici username, password e email e altre informazioni sull'utente.
- L'entità **Ordine** è l'entità centrale del sistema. Lo stato dell'ordine che, a differenza del modello concettuale, è stato trasformato in attributo. Attraverso *ottieniDescState()* gli utenti potranno visualizzare in quale stato è l'ordine, ovvero
  - Ordine pagato in attesa
  - Ordine Bar in Esecuzione
  - Ordine Cucina in Esecuzione
  - Ordine Bar e Cucina in esecuzione
  - Ordine Cucina concluso, Ordine Bar in esecuzione
  - Ordine Bar concluso, Ordine Cucina in esecuzione
  - Ordine concluso

*ottieniTempoAttesa()* ha come scopo far visualizzare agli utenti una stima del tempo che serve per completare l'ordine. Per semplicità è stato calcolato la somma dei tempi di attesa di ogni prodotto dell'ordine. Ovviamente l'ordine ha il riferimento al locale in cui è stato effettuato, e attraverso questo è possibile ricavare gli operatori che ci lavorano. Infine *addProduct()* serve ad aggiungere all'ordine i prodotti con le relative quantità.

- L'entità **OPAssociation** crea una tabella accessoria ORDER\_PRODUCTS, la quale associa ad ogni coppia ordine-prodotto la quantità che il cliente vuole acquistare.

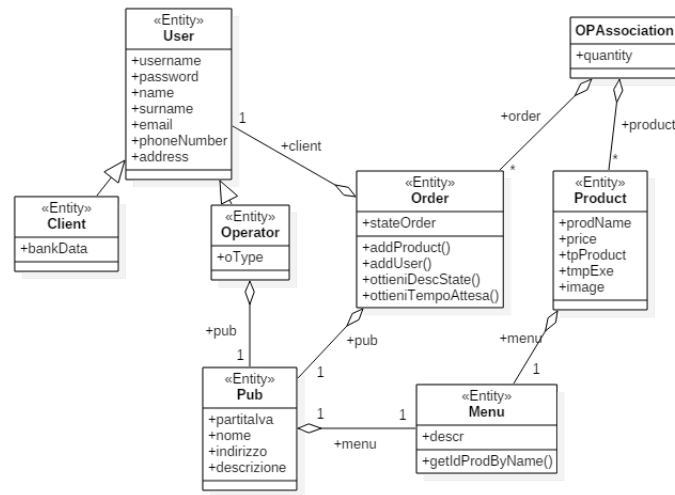


Figura 3.5: Modello di dominio

- L'entità **Product** ha tutte le informazioni che deve avere un prodotto, come il nome, il prezzo, il tempo di preparazione, l'immagine e il tipo di prodotto (cibo o drink). Quest'ultimo, a differenza del modello concettuale, è un attributo. Product fa infine riferimento a un menù.
- L'entità **Menù** ha una descrizione e una relazione 1 a 1 con il pub. Infatti si assume che il pub abbia solo 1 menù.
- L'entità **Pub** ha come attributi tutte le informazioni sul pub, come il nome, l'indirizzo e la partita IVA, e il riferimento a 1 menù.

# Capitolo 4

## Architettura Implementativa

### 4.1 Domain Logic

Per implementare la **logica di dominio** è stato utilizzato il modello 3-Tier nelle figure 4.1 e 4.2.

- **Domain Model:** contiene tutte le entità del sistema con i relativi mapping su database. Questa parte è implementata utilizzando la specifica JPA ed implementate tramite il framework Hibernate.
- **DAO:** ogni entità nel Domain Model possiede un suo DAO, che gestisce le comunicazioni con il database per il recupero e la persistenza delle entità attraverso l'Entity Manager. I DAO forniscono un ulteriore livello di astrazione tra Dominio e Database. In essi sono contenute tutte le query JPQL utilizzate dal sistema.
- **Business Logic:** implementa la logica di controllo delle pagine web attraverso i Controller. Questi utilizzano i DAO per il recupero e la persistenza delle entità del dominio. Le funzioni dei controller operano sulle entità e sono alla base dei casi d'uso.

#### 4.1.1 Domain Model

Facendo riferimento all'UML prodotto e, in particolar modo, alla figura 3.5, è stato implementato il modello di dominio. Le relazioni di aggregazione con il diamante e la molteplicità 1 a molti si sono tradotte nell'annotazione *@OneToMany*; tra Pub e Operator, ad esempio, c'è una relazione 1 a

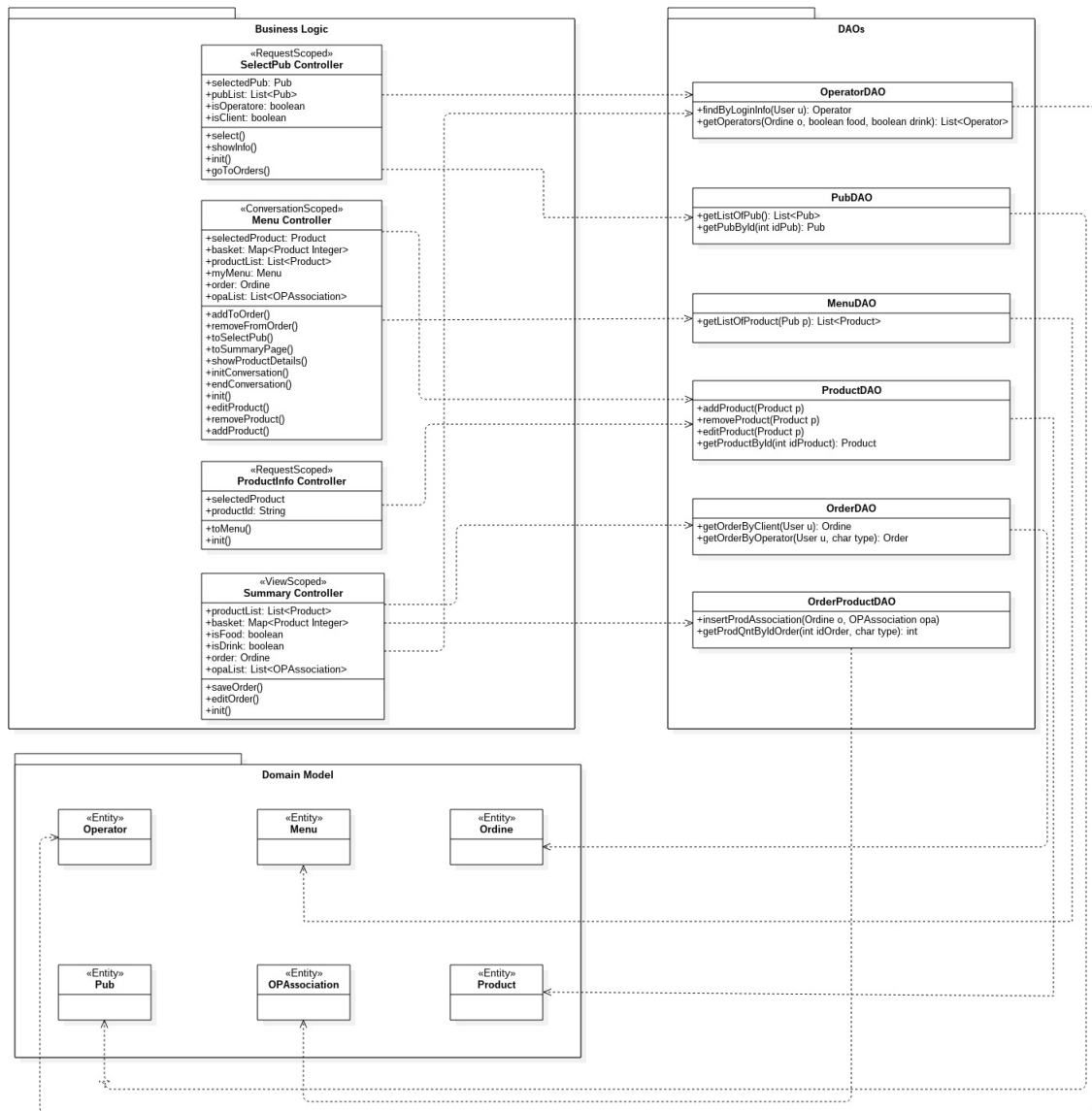


Figura 4.1: 3-Tier Model per le pagine SelectPub, Menu, ProductInfo e Summary Controller



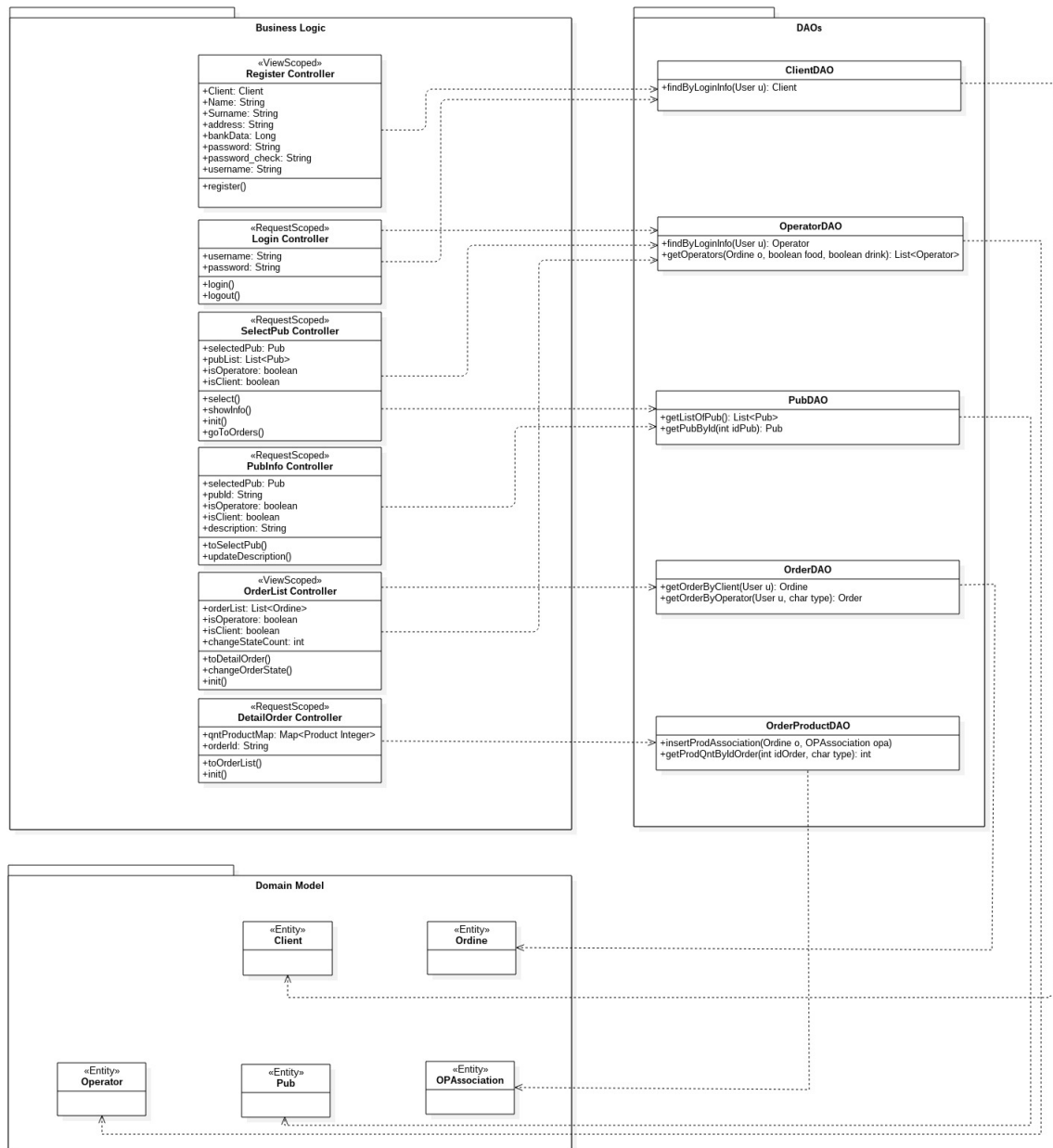


Figura 4.2: 3-Tier Model per le pagine Register, Login, SelectPub, PubInfo, OrderList e DetailOrder Controller

molti, perciò la classe Pub ha una lista di operatori taggata con l'annotazione `@OneToMany` mentre la classe ha un attributo *local* taggato con `@ManyToOne`. Nella classe Pub, attraverso *mappedBy = "local"*, nel database la tabella Operator avrà una colonna chiamata *idLocale.FK* che rimanderà all'id del pub in cui lavora, senza aver bisogno di una tabella aggiuntiva. Per vincolare alcuni attributi a non essere nulli quando vengono persistiti nel database, è stata usata la notazione JPA `@NotNull`, come ad esempio gli attributi **username,password** e **e-mail** dell'entità User.

Nel database non c'è la tabella User ma ci sono le tabelle Cliente e Operatore con tutti i loro attributi e quelli ereditati. Per fare ciò l'entità User ha il tag `@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)`.

La relazione tra Product e Ordine è molti a molti e, per implementarla, è stata utilizzata l'entità OPAssociation, che ha come riferimento l'ordine e il prodotto. Queste sono le chiavi primarie della tabella e sono entrambi legati con relazione `@ManyToOne`, e ad ogni coppia ordine prodotto è associata una quantità.

Per aggiungere prodotti all'ordine e le relative quantità, sull'ordine viene chiamato il seguente metodo:

---

```
OPAssociation addProduct(Product product, int quantity){
    OPAssociation association = new OPAssociation();
    association.setProduct(product);
    association.setOrder(this);
    association.setIdProduct(product.getIdProduct());
    association.setQuantity(quantity);
    if (this.products == null) {
        this.products = new ArrayList<>();
    }
    product.getOrders().add(association);
    return association;
}
```

---

### 4.1.2 DAO

Per sviluppare i DAO è stata costruita una classe **BaseDAO**, che è stata estesa da tutti i DAO sviluppati. Essa implementa i metodi *save()*, *update()*, *delete()* e *findById()* che quindi sono i metodi base di ogni DAO, inoltre mantiene il riferimento all'EntityManager. Per far questo, la classe sfrutta

i generics di Java, dove il tipo generico rappresenta la classe del dominio corrispondente ad un particolare DAO.

---

```
//Inizio della dichiarazione di BaseDao
public abstract class BaseDao<E> implements Serializable

//Inizio della dichiarazione di OrderDAO
public class OrderDAO extends BaseDao<Ordine>
```

---

A seconda delle necessità dei controller della Business Logic, sono stati implementati i metodi di ogni DAO. Degna di nota è la funzione `getProdQntByIdOrder(int idOrder, char type)` di **OrderProductDAO**. Questo metodo è utilizzato dal controller nella pagina di dettagli del prodotto. Il sistema così riesce a ricostruire le coppie (Prodotto, quantità) dall'ID dell'ordine e fa visualizzare i suoi dettagli ai diversi utenti definiti da *type*. Il cliente, infatti, vedrà tutti i prodotti che ha ordinato con le relative quantità, un cuoco vedrà solamente i prodotti di tipo cibo mentre il barista solo i prodotti di tipo drink.

---

```
public Map<Product, Integer> getProdQntByIdOrder(int idOrder,
    char type) {
    List<OPAssociation> prodAssociations = null;
    Map<Product, Integer> qntProductMap = new HashMap<Product,
    Integer>();

    try {
        prodAssociations = entityManager
            .createQuery("from OPAssociation opa where opa.
idOrder=:idOrdine", OPAssociation.class)
            .setParameter("idOrdine", idOrder).getResultList();
        if (type == 'b') { // barista
            for (OPAssociation opAssoc : prodAssociations) {
                if (opAssoc.getProduct().getTpProduct() == 'd') {
                    qntProductMap.put(opAssoc.getProduct(), opAssoc.
getQuantity());
                }
            }
        } else if (type == 'c') { // cuoco

            for (OPAssociation opAssoc : prodAssociations) {
                if (opAssoc.getProduct().getTpProduct() == 'f') {
                    qntProductMap.put(opAssoc.getProduct(), opAssoc.
getQuantity());
                }
            }
        }
    } catch (Exception e) {
        // gestione eccezione
    }
}
```

```

    }
  }
  } else { // cliente
    for (OPAssociation opAssoc : prodAssociations) {
      qntProductMap.put(opAssoc.getProduct(), opAssoc.
getQuantity());
    }
  }
} catch (Exception ex) {
  ex.printStackTrace();
}
return qntProductMap;
}

```

---

Altri 2 metodi degni di nota sono `getOrderByClient(User client)` e `getOrderByOperator(Operator operator)` di **OrderDAO**. Queste funzioni sono utilizzate dal controller nella pagina in cui vengono visualizzati tutti gli ordini di un utente. Simile al caso precedente, la prima permette al client di visualizzare tutti gli ordini che ha effettuato, in ogni locale; la seconda invece permette a un cuoco di visualizzare tutti gli ordini effettuati nel locale in cui lavora che contengono almeno 1 prodotto di tipo cibo, mentre per il barista tutti gli ordini con almeno 1 prodotto di tipo drink.

---

```

public List<Ordine> getOrderByClient(User client) {

    List<Ordine> orderList = new ArrayList<Ordine>();

    orderList = entityManager.createQuery("from Ordine o
where o.client.idUser= :clientId", Ordine.class)
        .setParameter("clientId", client.getIdUser()).
getResultList();

    return orderList;

}

```

---

```

public List<Ordine> getOrderByOperator(Operator operator) {

    String condizione = "";
    char tpOperator=operator.getoType();

```

```

List<Ordine> orderList = new ArrayList<Ordine>();

if (tpOperator == 'b') { //barista
    condizione = "(o.barman is null or o.barman.
idUser = :userId) and o.tipoOrdine in ('d', 'm') and o.local=
:local";
} else if (tpOperator == 'c') { //cuoco
    condizione = "(o.cook is null or o.cook.idUser =
:userId) and o.tipoOrdine in ('f', 'm') and o.local= :local"
;
}
String query = "from Ordine o where " + condizione;
orderList = entityManager
    .createQuery(query, Ordine.class)
    .setParameter("local", operator.getLocal())
    .setParameter("userId", operator.getIdUser())
    .getResultList();
return orderList;
}

```

---

### 4.1.3 Business Logic

Per l'implementazione della Business Logic è stato utilizzato, insieme a Java, le annotazioni CDI che hanno permesso l'injection dei DAO e dei Bean nei Controller. Sono stati realizzati 9 controller, uno per ogni pagina.

I **Controller** sono stati annotati con 3 diversi scope:

- **RequestScope**, per i controller le cui pagine devono solamente far visualizzare delle informazioni agli utenti e che, quindi, terminano il loro lifecycle non appena la pagina JSF viene generata.
- **ViewScope**, che li tiene in vita fintanto che la pagina web rimane attiva, per i controller che devono modificare e persistere entità sul database.
- **ConversationScope**, il cui inizio e la cui fine vengono regolati, rispettivamente, da *beginConversation()* e *endConversation()*. Un solo controller è di questo tipo, che mantiene le informazioni per più pagine.

Sono stati implementati anche 2 **Bean** annotati **@SessionScoped**, dai quali i controller possono ricavare informazioni valide per tutta la sessione. In particolare:

- **UserSessionBean**, il quale viene settato durante il login e contiene il riferimento all'utente e al suo tipo (cliente, cuoco, barista).
- **SelectPubBean**, inizializzato non appena il cliente seleziona il pub in cui vuole effettuare l'ordine e che mantiene il riferimento a questo locale.

### Login e Register Controller

Il **Login Controller** ha essenzialmente la funzione *login()* che controlla se esiste l'utente con id e password inseriti e setta lo **UserSessionBean**. Il **Register Controller** ha invece la funzione *register()* la quale controlla che i dati necessari per l'iscrizione siano validi e, in caso affermativo, persiste un client nel database. Per questo è stato annotato come **@ViewScoped**, a differenza del primo che è annotato come **@RequestScoped**.

### SelectPub e PubInfo Controller

Il **SelectPub Controller** è il controller dell'Homepage, che controlla se l'utente è un operatore o un client. Nel primo caso viene settato il locale in cui l'operatore lavora, mentre nel secondo crea una lista di pub che l'utente, tramite la funzione *select()*, può scegliere. In fase di progettazione il sistema era stato pensato in modo tale da far visualizzare solamente i locali vicini al client ma, per semplicità, crea la lista con tutti i pub del database.

---

```
@PostConstruct
public void init() {

    isOperatore = false;
    isClient = false;

    if (userSessionBean.getType() != 'u') { //operatore
        isOperatore = true;
        operatorPub = operatorDao
            .findByLoginInfo(userSessionBean.getUser())
            .getLocal();
    } else { //cliente
```

```

        pubList= new ArrayList<Pub>();
        pubList = pubDao.getListOfPub();
        isClient = true;
    }
}

```

---

Il **PubInfo Controller** invece permette di visualizzare i dettagli del pub selezionato a un client e permette di modificare le informazioni del pub a un operatore. Per questo il controller è annotato come `@ViewScoped`, mentre **Select-Pub Controller**, che permette solo di visualizzare dei pub, è `@RequestScoped`. L'id del pub viene ottenuto tramite l'annotazione `@HttpParam("id")`.

## Menu, Summary e ProductInfo Controller

La gestione del menu è demandata al **Menu Controller**, annotato come `@ConversationScoped`. Una volta selezionato il pub d'interesse, infatti, ha inizio il lifecycle del controller, che terminerà quando il cliente riterrà conclusa la creazione dell'ordine. A tal proposito, è stato necessario effettuare l'injection del controller del menu in tutti quei controller che vengono considerati durante la creazione di un ordine.

Se un cliente è limitato alla visualizzazione e selezione dei prodotti che compongono il menu, un operatore ha invece la possibilità di modificare la composizione del menu. Per far questo, sono presenti i metodi *addProduct()*, *editProduct()* e *removeProduct()*.

Nel caso di un Client, in fase di inizializzazione viene creata un'istanza dell'entità **Ordine**, la quale viene settata con il Client loggato e col Pub che ha selezionato. Dopo che l'utente ha selezionato tutte le quantità dei prodotti che vuole ordinare, entra in gioco il **Summary Controller**, il quale permette il riepilogo dei prodotti con relative quantità tenute in vita dal Menu Controller. Qui viene chiamato *SaveOrder()* che prima persiste l'ordine precedentemente creato specificando se contiene solo drink, solo cibo o entrambi tipi di prodotti e poi persiste le entità **OPAssociation** con le quantità scelte per ogni coppia (Ordine, Prodotto) create nel Menu Controller.

Il **ProductInfo Controller** ha solo il compito di mostrare al Client i dettagli del prodotto e per questo è annotato come `@RequestScoped`, diversamente dal **SummaryController**, il quale deve persistere, che è annotato con `@ViewScoped`.

## OrderList e DetailOrder Controller

**OrderList Controller** ha il compito di mostrare tutti gli ordini effettuati, nel caso di un Client, mentre nel caso di un Operatore tutti gli ordini effettuati nel pub in cui lavora e attinenti al tipo di operatore.

---

```
@PostConstruct
public void init() {

    ...

    if (userSessionBean.getType() == 'c' || userSessionBean.
        getType() == 'b') { //operatore

        setOrderList(orderDao.getOrderByOperator((Operator)
            userSession));
        isOperatore = true;

    } else { //client

        setOrderList(orderDao.getOrderByClient(userSession));
        isClient = true;

    }

}
```

---

L'operatore, tramite il metodo *changeOrderState()*, permette di modificare lo stato dell'ordine e di associare, a quel determinato ordine, l'operatore che ha chiamato il metodo. In questo modo è l'operatore che ha la responsabilità di associarsi a un ordine e non il contrario. Poi viene aggiornato l'ordine nel database e, per questo, il controller è annotato come `@ViewScoped`.

Infine **DetailOrder Controller** ricostruisce le coppie (Prodotto, quantità) di un ordine selezionato e mostra i dettagli dell'ordine all'utente. Perciò, diversamente dal precedente, è annotato come `@RequestScoped`.



# Capitolo 5

## Test

La fase di testing è stata eseguita paralelamente all'implementazione del software, in modo da assicurare che ogni nuova classe Java inserita si comportasse nella maniera attesa. In tutti i test eseguiti è stato verificato il corretto funzionamento dei metodi appartenenti alle varie classi. In particolare sono stati testati i Controller, i DAO e le classi del modello, attraverso i framework di **JUnit** e **Mockito**. Quest'ultimo, in combinazione con i *FieldUtils* di Apache, ha permesso di disaccoppiare il testing dei controller dal funzionamento dei DAO. Per una migliore gestione delle eccezioni in fase di test è stata usata la libreria **AssertJ** che permette di trattare le eccezioni in maniera analoga agli *Assert* di JUnit.

Attraverso il programma **JaCoCo** è stato possibile misurare la coverage, visibile in figura 5.1.

Di seguito viene presentato una parte dei test effettuati su **Menu Controller** in cui vengono testate le funzioni che aggiungono e rimuovono prodotti che il client vuole ordinare. In fase di inizializzazione viene creato un hashmap con 3 prodotti inizializzati alla quantità 0 e viene usato Mockito e FieldUtils sul MenuDAO.













Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
it.unifi.swa.controller		49%		45%	149	210	276	460	114	156	2	9
it.unifi.swa.dao		32%		5%	34	46	72	116	15	27	3	8
it.unifi.swa.domain		75%		36%	25	127	43	203	13	109	1	9
it.unifi.swa.bean.producer		0%		n/a	2	2	6	6	2	2	1	1
it.unifi.swa.bean.startup		100%		n/a	0	2	0	155	0	2	0	1
it.unifi.swa.bean		100%		100%	0	10	0	12	0	9	0	2
Total	1.287 of 3.190	60%	118 of 184	36%	210	397	397	952	144	305	7	30

Figura 5.1: Risultati dei test

---

```
@Before
public void init() throws InitializationError {

    ...

    menuDao=mock(MenuDAO.class);

    p1=new Product();
    p2=new Product();
    p3=new Product();

    list.add(p1);
    list.add(p2);
    list.add(p3);

    for (Product element : list) {
        basket.put(element, 0);
    }

    try {
        FieldUtils.writeField(menuController, "menuDao", menuDao,
            true);
        ...
    } catch (IllegalAccessException e) {
        throw new InitializationError(e);
    }

}
```

---

Inizialmente il valore di ogni elemento del carrello è 0 e, ogni volta che viene aggiunto un prodotto, la sua quantità aumenta di 1.

---

```
@Test
public void addToOrderTest() {

    ...

    for (Map.Entry<Product, Integer> entry : menuController.
        getBasket().entrySet()) {
        assertTrue(entry.getValue().equals(0));
    }
}
```

```

    }

    menuController.addToOrder(p1);

    for (Map.Entry<Product, Integer> entry : menuController.
getBasket().entrySet()) {

        if (entry.getKey().equals(p1)) {
            assertTrue(entry.getValue().equals(1));
        } else {
            assertTrue(entry.getValue().equals(0));
        }
    }

    menuController.addToOrder(p1);

    ...

    menuController.addToOrder(p2);

    for (Map.Entry<Product, Integer> entry : menuController.
getBasket().entrySet()) {

        if (entry.getKey().equals(p1)) {
            assertTrue(entry.getValue().equals(2));
        } if (entry.getKey().equals(p2)) {
            assertTrue(entry.getValue().equals(1));
        } if (entry.getKey().equals(p3)) {
            assertTrue(entry.getValue().equals(0));
        }
    }

}

```

---

Allo stesso modo, quando viene decrementato un prodotto, la sua quantità diminuisce di 1. È testato in seguito il caso in cui venga rimosso un prodotto la cui quantità sia già 0, attendendo che essa continui a rimanere tale senza avere un valore negativo.

---

```

@Test
public void removeFromOrderTest() {

    ...

```

```

menuController.addToOrder(p1);

for (Map.Entry<Product, Integer> entry : menuController.
getBasket().entrySet()) {

    if (entry.getKey().equals(p1)) {
        assertTrue(entry.getValue().equals(1));
    } else {
        assertTrue(entry.getValue().equals(0));
    }
}

menuController.removeFromOrder(p1);

for (Map.Entry<Product, Integer> entry : menuController.
getBasket().entrySet()) {
    assertTrue(entry.getValue().equals(0));
}

menuController.removeFromOrder(p1);

for (Map.Entry<Product, Integer> entry : menuController.
getBasket().entrySet()) {
    assertTrue(entry.getValue().equals(0));
}

}

```

---

Infine viene presentato di seguito un test per il metodo `getOrderByOperator(Operator o)` della classe **OrderDAO** presentato nella sezione dei DAO. Viene simulata una situazione in cui ci sono 2 barman che lavorano nel solito pub, in cui un client effettua due ordini.

---

```

@Before
protected void init() throws InitializationError {

    ...
    pub= new Pub();
    client=new Client();

    barman1=new Operator();
}

```

```

        barman1.setType('b');
        barman1.setLocal(pub);
        barman2=new Operator();
        barman2.setType('b');
        barman2.setLocal(pub);

        Product p1= new Product();
        Product p2= new Product();
        Product p3= new Product();

        order1=new Ordine();
        order2=new Ordine();

        order1.setClient(client);
        order1.setLocal(pub);
        order1.addProduct(p1, 1);
        order1.addProduct(p2, 2);
        order1.addProduct(p3, 1);

        order2.setClient(client);
        order2.setLocal(pub);
        order2.addProduct(p1, 2);
        order2.addProduct(p3, 4);

        //persist
    }

```

---

Prima un ordine è solo drink mentre l'altro solo food. Quindi entrambi i barman, che lavorano nel solito pub e che vedono solo gli ordini in cui c'è almeno un drink (drink 'd' o misto 'm'), vedranno solo order1. Nel momento in cui barman1 prende in carico order1, barman2 non vedrà più ordini. Quando order2, che non ha nessun operatore in carico, diventa un ordine di tipo misto, barman2 riuscirà a vederlo e barman1 li vedrà entrambi.

---

```

@Test
public void getOrderByBarmanTest() {

    order1.setTipoOrdine('d');
    order2.setTipoOrdine('f');

    orderDao.update(order1);
    orderDao.update(order2);
}

```

```
orderList.add(order1);

assertEquals(orderDao.getOrderByOperator(barman1), orderList);
assertEquals(orderDao.getOrderByOperator(barman2), orderList);

order.setBarman(barman1);
assertEquals(orderDao.getOrderByOperator(barman1), orderList);
assertTrue(orderDao.getOrderByOperator(barman2).isEmpty());

order2.setTipoOrdine('m');
orderDao.update(order2);

orderList.add(order2);
assertEquals(orderDao.getOrderByOperator(barman1), orderList);
assertFalse(orderDao.getOrderByOperator(barman2).isEmpty());
}
```

---

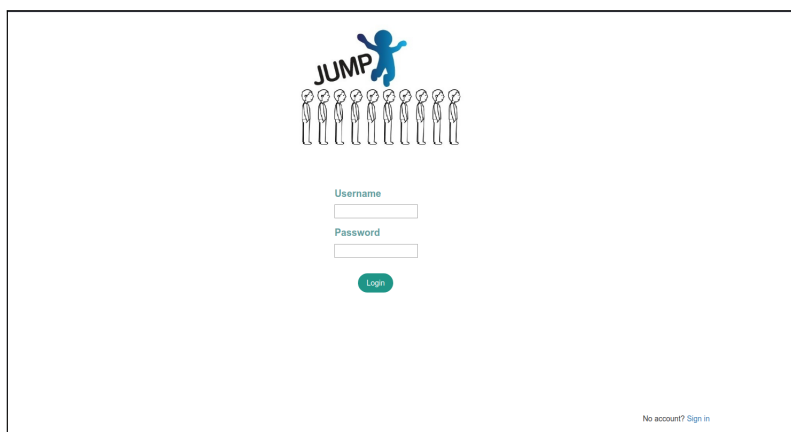
# Capitolo 6

## Interfaccia Web

Durante lo svolgimento del lavoro, per valutare il corretto funzionamento dei Controller e dei DAO, sono state sviluppate alcune pagine JSF dell'interfaccia web. Lo sviluppo dell'interfaccia ha permesso il test di tutto il sistema e, in particolare, delle annotazioni CDI, che non era stato effettuato nei test dei controller. Questo ha permesso di correggere alcuni errori (nei controller, nei dao e nel modello) molto difficili da individuare con i semplici test di unità. Nelle Figure 6.1 e 6.2 sono riportate, rispettivamente, le pagine di login e registrazione.

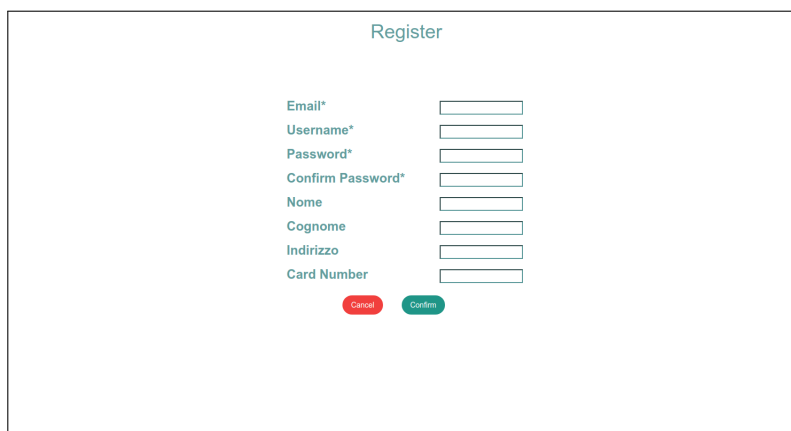
### 6.1 HomePage

L'interfaccia per la home page è mostrata nelle Figure 6.3 e 6.4. Questa pagina assume caratteristiche differenti a seconda del tipo di utente che ha effettuato il login: nel caso di un cliente, viene data la possibilità di selezionare il pub da una lista di possibili scelte e di visualizzare le informazioni ad essi relative; nel caso in cui sia un operatore ad aver effettuato il login, verrà mostrato il locale in cui è assegnato con la possibilità di modificare il menù del locale stesso, di visualizzare gli ordini effettuati in esso e di modificare le proprie informazioni.



The login page features the JUMP logo at the top center, which consists of the word "JUMP" in a bold, sans-serif font next to a blue stick figure jumping. Below the logo is a row of ten small, identical stick figures standing side-by-side. Underneath the figures are two input fields: the first is labeled "Username" and the second is labeled "Password". Below these fields is a green "Login" button. In the bottom right corner, there is a link that says "No account? Sign in".

Figura 6.1: Pagina di Login



The registration page is titled "Register" at the top center. It contains a series of input fields for user registration: "Email\*", "Username\*", "Password\*", "Confirm Password\*", "Nome", "Cognome", "Indirizzo", and "Card Number". Each label is followed by an input field. At the bottom of the form, there are two buttons: a red "Cancel" button and a green "Confirm" button.

Figura 6.2: Pagina di Registrazione





Figura 6.3: HomePage Cliente

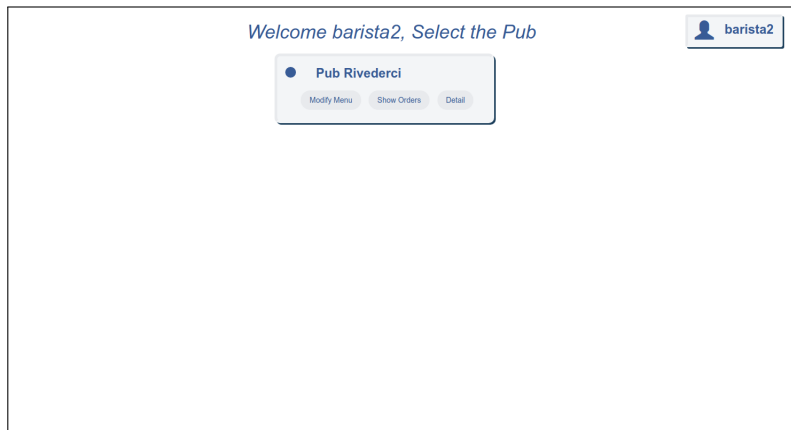


Figura 6.4: HomePage Operatore

## 6.2 Pagine di Gestione del Locale

Le pagine di modifica e visualizzazione del locale hanno permesso di testare l'uso dei parametri http e l'annotazione `@ViewScoped`. La pagina di visualizzazione delle informazioni di un locale è mostrata in Figura 6.5

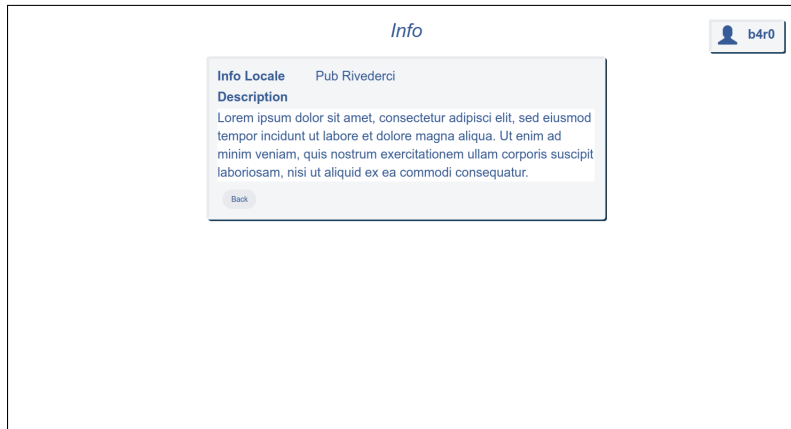


Figura 6.5: Informazioni del Locale



Figura 6.6: Dettaglio del Prodotto

## 6.3 Pagine di Gestione dell'Ordine

Le pagine relative alla gestione dell'ordine sono:

- la pagina di visualizzazione del menù e di modifica dell'ordine
- la pagina di visualizzazione del sommario dell'ordine
- la pagina di visualizzazione degli ordini effettuati

Fare queste pagine ha permesso di testare il corretto funzionamento dell'annotazione `@ConversationScoped`, usata in `MenuController`. Questo controller, infatti, è iniettato da CDI in ciascuno dei controller delle pagine sopra elencate. Alcune delle pagine sono mostrate nelle figure 6.7, 6.9 e 6.10

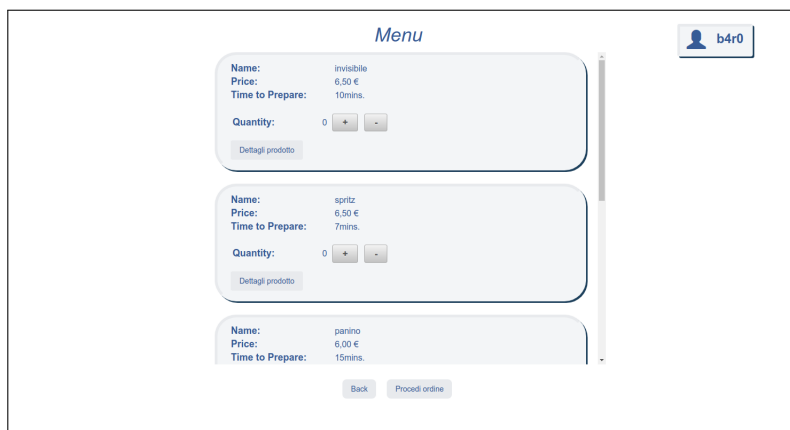


Figura 6.7: Pagina di Menù

The screenshot shows a web interface titled "Menu" for operator "barista2". It contains two main form sections. The first section is for editing an existing menu item, with fields for Name (filled with "Schiocciata"), Price (filled with "6.0"), Time to Prepare (filled with "9" and "mins."), and Image (filled with "https://www.biancofelvi.it/"). Below these fields are two buttons: "Modifica prodotto" and "Elimina prodotto". The second section is for adding a new menu item, with empty fields for Name, Price, Time to Prepare (with a "mins." label), and Image. Below these fields is a "Conferma" button. At the bottom of the page are "Back" and "Cancel" buttons.

Figura 6.8: Pagina di modifica del Menù (solo per operatore)

The screenshot shows a web interface titled "Your Order" for operator "b4r0". It displays a summary of the current order with two items. The first item is "spritz" with a price of "6.50 €", a preparation time of "7 mins.", and a quantity of "2". The second item is "panino" with a price of "6.00 €", a preparation time of "15 mins.", and a quantity of "1". At the bottom of the page are two buttons: "Modifica Online" and "Paga".

Figura 6.9: Sommario dell'Ordine

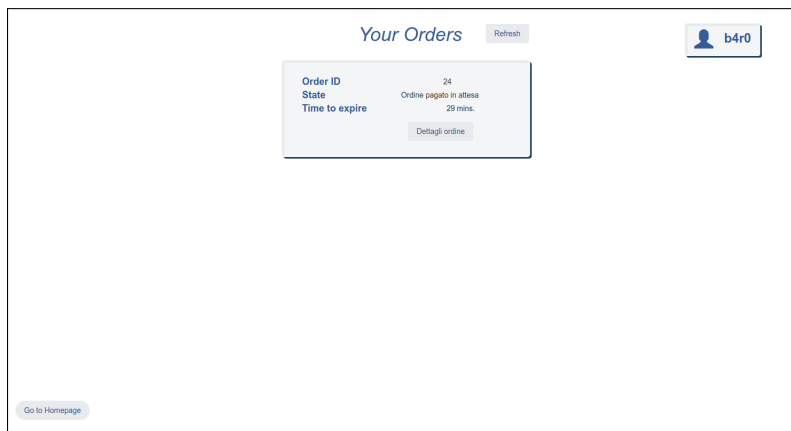


Figura 6.10: Pagina con l'elenco degli ordini effettuati

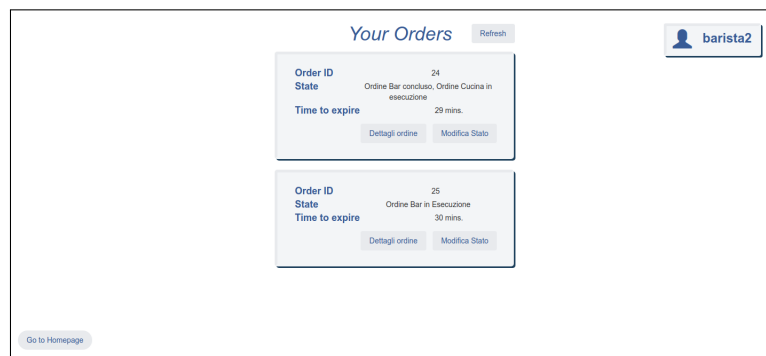


Figura 6.11: Pagina di gestione degli ordini dell'operatore

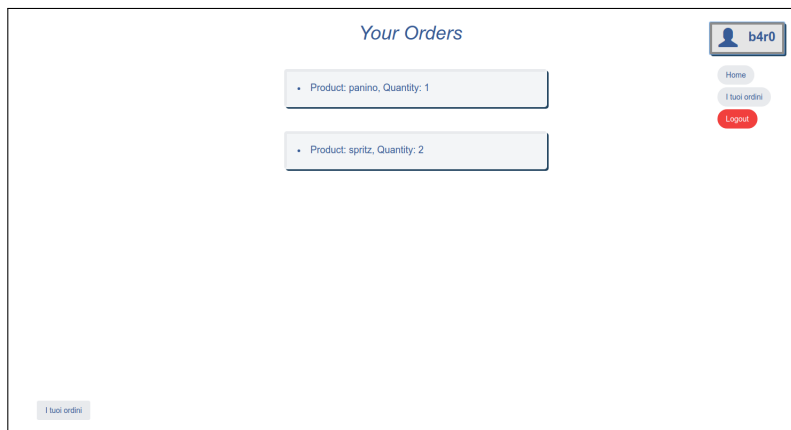


Figura 6.12: Dettaglio dell'ordine effettuato

## Capitolo 7

# Considerazioni finali e sviluppi futuri

In questo elaborato, è stata modellata, sviluppata e testata una Web Application per la gestione di ordini in locali pubblici. Questa applicazione è però una versione beta dato che avrebbe bisogno di importanti implementazioni aggiuntive per un buon funzionamento nei locali veri. Sviluppi futuri sono un aumento della percentuale di coverage nel testing, una migliore implementazione del sistema che notifica un cliente della modifica dello stato dell'ordine e un miglioramento dell'interfaccia, seguito da uno usability test che verifichi come un vero utente interagisce con il sistema.