Corso di Laurea in Informatica - Università degli Studi di Napoli Federico II
A.A. 2025/2026

UninaFoodLab

Calone Francesco N86005555

D'Angelo Mario N86005477

Codice gruppo: **OOBD39**

Insegnamento di Programmazione Object-Oriented

Indice

1 Introduzione **2**
1.1 Obiettivi del Progetto . . . 2

2 Tool utilizzati **2**
2.1 Maven . . . 2
2.2 JavaFX . . . 2
2.3 PostgreSQL . . . 3

3 Architettura del Progetto **3**
3.1 Struttura delle Directory . . . 3

4 Gestione della GUI **4**
4.1 Struttura della GUI . . . 4
4.2 Personalizzazione grafica con CSS . . . 5
4.3 Accesso e Registrazione . . . 5
4.3.1 Login . . . 5
4.3.2 Registrazione . . . 6
4.4 Ruoli e funzionalità: Utente e Chef . . . 8
4.5 Interazione con l'Utente . . . 8
4.5.1 Homepage Utente . . . 8
4.5.2 Iscrizione ai Corsi . . . 9
4.5.3 Visualizzazione Calendario delle Sessioni . . . 10
4.5.4 Acquisto e Pagamento dei Corsi . . . 10
4.5.5 Gestione Account Utente . . . 11
4.5.6 Gestione Carte Utente . . . 12
4.6 Interazione con lo Chef . . . 13
4.6.1 Homepage dello Chef . . . 14
4.6.2 Creazione Corso . . . 14
4.6.3 Modifica dei Corsi . . . 18
4.6.4 Report Mensile . . . 22
4.6.5 Gestione Account Chef . . . 24

5 Utils **28**
5.1 Principali utility utilizzati . . . 28
5.2 Vantaggi dell'utilizzo di utility . . . 28
5.3 SceneSwitcher . . . 29
5.4 SuccessDialogUtils . . . 30
5.5 CardValidator . . . 31
5.6 ImageClipUtils . . . 32
5.7 FrequenzaSessioniProvider . . . 33
5.8 UnifiedRecipeIngredientUI . . . 33
5.9 ErrorCarimentoPropic . . . 34

6 DTO e DAO **35**
6.1 Elenco dei DTO e DAO implementati . . . 35
6.2 DTO . . . 37
6.2.1 Utente . . . 37

6.2.2 UtenteVisitatore . . . 37
6.2.3 Chef . . . 37
6.2.4 Corso . . . 38
6.2.5 Sessione . . . 39
6.2.6 SessioneOnline . . . 39
6.2.7 SessioniInPresenza . . . 40
6.2.8 Ricetta . . . 40
6.2.9 Ingredienti . . . 41
6.2.10 GraficoChef . . . 41
6.2.11 CartaDiCredito . . . 42
6.3 DAO . . . 43
6.3.1 UtenteDao . . . 43
6.3.2 UtenteVisitatoreDao . . . 44
6.3.3 ChefDao . . . 45
6.3.4 CorsoDao . . . 46
6.3.5 SessioniDao . . . 49
6.3.6 SessioneOnlineDao . . . 50
6.3.7 SessioneInPresenzaDao . . . 51
6.3.8 RicettaDao . . . 52
6.3.9 IngredientiDao . . . 54
6.3.10 GraficoChefDao . . . 55
6.3.11 CartaDiCreditoDao . . . 56
6.3.12 BarraDiRicercaDao . . . 57
6.4 JDBC . . . 58
6.4.1 Connessione al database . . . 58
6.4.2 Gestione delle risorse . . . 58
6.4.3 Dipendenze e configurazione Maven . . . 58
6.4.4 Descrizione dei file principali . . . 59
6.4.5 Best practice e vantaggi . . . 59

1 Introduzione

Il progetto UninaFoodLab nasce con l'obiettivo di offrire una soluzione software moderna e intuitiva per la gestione di corsi culinari. La piattaforma è stata sviluppata seguendo le best practice adottando tecnologie consolidate come JavaFX per la realizzazione dell'interfaccia grafica e Maven per la gestione delle dipendenze e dei processi di build. Il progetto è stato concepito per essere facilmente estendibile e manutenibile, grazie a una chiara separazione dei livelli logici. La documentazione che segue illustra le scelte progettuali, le tecnologie utilizzate e le principali funzionalità implementate, con l'intento di fornire una panoramica completa e professionale del sistema sviluppato.

1.1 Obiettivi del Progetto

Il progetto UninaFoodLab si propone di:

- Fornire una piattaforma intuitiva per la gestione di corsi culinari
- Semplificare la registrazione e la gestione degli utenti e per gli chef

- Facilitare la creazione e la gestione dei corsi, inclusa la pianificazione delle lezioni e la gestione delle ricette
- Semplificare la gestione delle prenotazioni e dei pagamenti per i corsi

2 Tool utilizzati

2.1 Maven

Maven è uno strumento di gestione e automazione dei progetti software, ampiamente utilizzato nell'ecosistema Java. Permette di gestire le dipendenze, automatizzare la compilazione, l'esecuzione dei test e la creazione dei pacchetti eseguibili.

Nel progetto UninaFoodLab, Maven è stato utilizzato per semplificare la configurazione e la gestione delle librerie necessarie, come JavaFX per la realizzazione dell'interfaccia grafica e il driver JDBC per la connessione al database PostgreSQL. Tutte le dipendenze sono dichiarate nel file 'pom.xml', che consente di mantenere il progetto facilmente aggiornabile e portabile.

L'integrazione con JavaFX è stata gestita tramite le apposite dipendenze e plugin, permettendo di compilare ed eseguire l'applicazione con semplici comandi Maven. Questo approccio ha garantito una maggiore efficienza nello sviluppo e una migliore organizzazione del codice.

2.2 JavaFX

JavaFX è una libreria per la creazione di interfacce utente grafiche (GUI) in Java. È stata progettata per fornire un ambiente di sviluppo moderno e ricco di funzionalità, consentendo la creazione di applicazioni desktop e web con un aspetto e un comportamento coerenti.

Nel progetto UninaFoodLab, JavaFX è stato utilizzato per realizzare l'interfaccia grafica dell'applicazione. Grazie a JavaFX, è stato possibile implementare facilmente elementi UI complessi, come tabelle, grafici e controlli personalizzati, migliorando l'usabilità e l'estetica dell'applicazione.

L'integrazione di JavaFX con Maven ha semplificato ulteriormente il processo di sviluppo, consentendo di gestire le dipendenze e le configurazioni necessarie per l'utilizzo della libreria in modo efficiente e organizzato.

2.3 PostgreSQL

PostgreSQL è un sistema di gestione di database relazionali.

Nel progetto UninaFoodLab, PostgreSQL è stato scelto come database principale per la sua capacità di gestire grandi volumi di dati. L'integrazione con Java è stata realizzata tramite JDBC (Java Database Connectivity), che ha permesso di stabilire una connessione tra l'applicazione Java e il database PostgreSQL in modo semplice e diretto.

3 Architettura del Progetto

L'architettura del progetto UninaFoodLab è stata progettata per garantire una chiara separazione dei compiti e una facile manutenibilità. La struttura delle directory riflette i principali livelli logici dell'applicazione, secondo il pattern MVC e le best practice di progettazione.

Le principali suddivisioni sono:

- **Boundary**: contiene le classi responsabili dell'interfaccia grafica e dell'interazione con l'utente, realizzate tramite JavaFX e FXML.

- **Controller**: gestisce la logica applicativa e il flusso degli eventi tra la GUI e i dati.
- **Entity**: suddivisa ulteriormente in *DAO* (Data Access Object) e *DTO* (Data Transfer Object). I DAO si occupano della persistenza e dell'accesso ai dati, mentre i DTO rappresentano le strutture dati scambiate tra i vari livelli.
- **JDBC**: contiene le classi e le utility per la connessione e la gestione del database PostgreSQL.
- **Utils**: raccoglie le classi di supporto e gli strumenti riutilizzabili all'interno del progetto.

Questa organizzazione favorisce la modularità e la scalabilità del sistema, permettendo di isolare le responsabilità e facilitare l'estensione futura. Ogni componente interagisce con gli altri tramite interfacce ben definite, riducendo le dipendenze e migliorando la qualità del codice.

La scelta di suddividere le entity in DAO e DTO consente di gestire in modo efficiente sia la persistenza che il trasferimento dei dati, mentre la presenza di una directory dedicata alle utility semplifica la gestione delle funzionalità trasversali.

3.1 Struttura delle Directory

La struttura delle directory del progetto è organizzata come segue:

- **src/main/java**: contiene il codice sorgente dell'applicazione.
- **src/main/resources**: contiene le risorse dell'applicazione, come file FXML e immagini.
- **src/test/java**: contiene i test automatizzati.

4 Gestione della GUI

La gestione della Graphical User Interface (GUI) nel progetto UninaFoodLab è stata affidata a JavaFX, una tecnologia che permette di realizzare applicazioni desktop moderne, interattive e dal design professionale. L'utilizzo di JavaFX, insieme ai file FXML e CSS, ha consentito di separare in modo netto la logica di presentazione dalla logica applicativa, favorendo la manutenibilità e l'estendibilità del software.

Ogni sezione dell'applicazione è rappresentata da una scena dedicata, come la schermata di login, la homepage utente, la homepage chef, la gestione dei corsi e la visualizzazione delle ricette. Queste scene sono definite tramite file FXML, che descrivono la struttura e il layout degli elementi grafici, e sono arricchite da fogli di stile CSS che garantiscono coerenza visiva e personalizzazione dell'interfaccia.

La scelta di organizzare la GUI in scene distinte permette di gestire in modo ordinato le diverse funzionalità offerte agli utenti, facilitando sia l'esperienza d'uso che lo sviluppo incrementale del progetto. Inoltre, la presenza di controller dedicati per ciascuna scena assicura una gestione efficace degli eventi e delle interazioni, mantenendo il codice pulito e facilmente testabile.

4.1 Struttura della GUI

La struttura della GUI di UninaFoodLab si basa su una suddivisione modulare in scene, ciascuna dedicata a una funzionalità specifica dell'applicazione. Ogni scena è definita da un file FXML, che descrive in modo dichiarativo la disposizione degli elementi grafici (bottoni, tabelle, form, ecc.) e ne facilita la modifica senza dover intervenire direttamente sul codice Java.

I principali file FXML includono, ad esempio:

- `accountmanagement.fxml`
- `accountmanagementchef.fxml`
- `calendardialog.fxml`
- `cardcorso.fxml`
- `createcourse.fxml`
- `editcourse.fxml`
- `enrolledcourses.fxml`
- `homepagechef.fxml`
- `homepageutente.fxml`
- `loginpage.fxml`
- `logoutdialog.fxml`
- `monthlyreport.fxml`
- `paymentpage.fxml`
- `registerpage.fxml`
- `successdialog.fxml`
- `usercards.fxml`

Ogni file FXML è associato a una classe Boundary e a un Controller dedicato, che si occupano rispettivamente della gestione dell'interfaccia e della logica degli eventi. Questa organizzazione consente di mantenere il codice pulito e facilmente estendibile, favorendo la collaborazione tra sviluppatori e la suddivisione dei compiti.

La personalizzazione grafica è affidata ai fogli di stile CSS, che permettono di differenziare l'aspetto delle varie scene e di adattare l'interfaccia alle esigenze dei diversi ruoli (utente e chef). Ad esempio, la dashboard dello chef presenta strumenti e colori distintivi rispetto a quella dell'utente, rendendo immediata la comprensione delle funzionalità disponibili.

4.2 Personalizzazione grafica con CSS

La personalizzazione dell'interfaccia grafica è stata realizzata tramite fogli di stile CSS dedicati, che permettono di controllare l'aspetto di ogni elemento della GUI in modo flessibile e centralizzato. Ogni scena principale dispone di un proprio file CSS, come `loginpage.css`, `navbar.css`, `chef.css`, `courses.css`, che definiscono colori, font, spaziature e comportamenti visivi.

Questa scelta consente di mantenere una coerenza stilistica tra le diverse schermate e di differenziare l'esperienza utente in base al ruolo. Ad esempio, le dashboard di chef e utente presentano palette di colori e layout specifici, facilitando la navigazione e la comprensione delle funzionalità disponibili. Inoltre, l'utilizzo dei CSS semplifica eventuali modifiche grafiche future, rendendo il progetto facilmente adattabile a nuove esigenze o preferenze estetiche.

4.3 Accesso e Registrazione

Le funzionalità di accesso e registrazione sono fondamentali per la sicurezza e la personalizzazione dell'esperienza utente. Queste operazioni sono gestite tramite scene dedicate e controller specializzati, che garantiscono la validazione dei dati, la gestione degli errori e la corretta distinzione tra i ruoli di chef e utente visitatore.

4.3.1 Login

Il processo di autenticazione avviene tramite la schermata definita in `loginpage.fxml`, gestita da `LoginBoundary.java` e `LoginController.java`. L'utente inserisce le proprie credenziali e, al clic sul pulsante di login, il controller verifica i dati e mostra eventuali messaggi di errore o accede alla dashboard appropriata. Un esempio di metodo di gestione dell'evento potrebbe essere:

Il flusso di login è gestito in modo strutturato e sicuro. Di seguito viene mostrato un estratto reale del codice utilizzato:

```
// LoginBoundary.java
@FXML
private void LoginClick(ActionEvent event) {
    String email = emailField.getText();
    String password = passwordField.getText();
    String errorMsg = controller.handleLogin(event, email, password);
    if (errorMsg != null) {
        errorLabel.setText(errorMsg);
        errorLabel.setVisible(true);
    } else {
        errorLabel.setText("");
        errorLabel.setVisible(false);
    }
}

// LoginController.java
public String handleLogin(ActionEvent event, String email, String password) {
    try {
        Stage stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
        UtenteVisitatoreDao utenteDao = new UtenteVisitatoreDao();
        String tipo = utenteDao.TipoDiAccount(email, password);
        if (tipo == null) {
            return "Tipo di account non riconosciuto.";
        }
        if ("c".equals(tipo)) {
            ChefDao chefDao = new ChefDao();
            Chef chef = new Chef();
            chef.setEmail(email);
            chef.setPassword(password);
            chefDao.recuperaDatiUtente(chef);
            Chef.loggedUser = chef;
            SceneSwitcher.switchToMainApp(stage, "/fxml/homepagechef.fxml", "UninaFoodLab - Hom
            return null;
```

```
        } else if ("v".equals(tipo)) {
            UtenteVisitatore utente = new UtenteVisitatore();
            utente.setEmail(email);
            utente.setPassword(password);
            utenteDao.recuperaDatiUtente(utente);
            UtenteVisitatore.loggedUser = utente;
            SceneSwitcher.switchToMainApp(stage, "/fxml/homepageutente.fxml", "UninaFoodLab - H
            return null;
        } else {
            return "Email o password errati.";
        }
    } catch (IOException e) {
        return "Errore interno. Riprova.";
    }
}
```

In questo modo, il sistema distingue tra chef e utente visitatore, mostrando la dashboard corretta in base al ruolo. Gli eventuali errori vengono gestiti e comunicati all'utente in modo chiaro e immediato.

4.3.2 Registrazione

La registrazione di un nuovo account avviene tramite la schermata `registerpage.fxml`, gestita da `RegisterBoundary.java` e `RegisterController.java`. L'utente può scegliere se registrarsi come Chef o come Utente Visitatore, compilando i campi richiesti e selezionando il tipo di account.

Il controller si occupa di validare i dati inseriti, mostrando messaggi di errore dettagliati in caso di campi mancanti, password non coincidenti, email non valida o dati incoerenti. Se la registrazione va a buon fine, l'utente viene inserito nel database e visualizza una dialog di conferma.

Ecco un estratto del codice reale che gestisce la validazione e la registrazione:

```
// RegisterBoundary.java
@FXML
private void onRegistratiClick(ActionEvent event) {
    // Estrai i dati dalla GUI
    String nome = textFieldNome.getText().trim();
    String cognome = textFieldCognome.getText().trim();
    String email = textFieldEmail.getText().trim();
    String password = textFieldPassword.getText();
    String confermaPassword = textFieldConfermaPassword.getText();
    String genere = comboBoxGenere.getValue();
    String descrizione = textFieldDescrizione.getText().trim();
    String anniEsperienza = textFieldAnniEsperienza.getText().trim();
    boolean utenteSelezionato = radioUtente.isSelected();
    boolean chefSelezionato = radioChef.isSelected();
    var dataNascita = datePickerDataNascita.getValue();

    // Passa i dati al controller
    String errore = controller.validaRegistrazione(
        nome, cognome, email, password, confermaPassword, genere,
```

```
        descrizione, anniEsperienza, utenteSelezionato, chefSelezionato, dataNascita
    );

    if (errore != null) {
        labelErrore.setText(errore);
        labelErrore.setVisible(true);
        return;
    }

    String esito = controller.registraUtente(
        nome, cognome, email, password, genere, descrizione, anniEsperienza,
        utenteSelezionato, chefSelezionato, dataNascita
    );
    if (esito == null) {
        labelErrore.setText("");
        labelErrore.setVisible(false);
        showSuccessMessage("Registrazione completata con successo!");
        onIndietroClick(event);
    } else {
        labelErrore.setText(esito);
        labelErrore.setVisible(true);
    }
}

// RegisterController.java
public String validaRegistrazione(
    String nome, String cognome, String email, String password, String confermaPassword,
    String genere, String descrizione, String anniEsperienza,
    boolean utenteSelezionato, boolean chefSelezionato, LocalDate dataNascita
) {
    // ...validazione campi obbligatori, password, email, tipo account, ecc...
    return valid ? null : messaggioErrore.toString();
}

public String registraUtente(
    String nome, String cognome, String email, String password, String genere,
    String descrizione, String anniEsperienza, boolean utenteSelezionato,
    boolean chefSelezionato, LocalDate dataNascita
) {
    // ...inserimento nel database e gestione errori...
    return null; // oppure messaggio di errore
}
```

Questa logica garantisce che solo dati corretti e coerenti vengano accettati, migliorando la sicurezza e la qualità dell'esperienza utente. In caso di successo, l'utente viene reindirizzato alla schermata di login per accedere con le nuove credenziali.

4.4 Ruoli e funzionalità: Utente e Chef

Nei paragrafi successivi verranno analizzate nel dettaglio le funzionalità e le interazioni specifiche per ciascun ruolo, evidenziando le differenze tra utente visitatore e chef sia dal punto di vista della GUI che della logica applicativa.

4.5 Interazione con l'Utente

L'interazione con l'utente costituisce uno degli aspetti centrali dell'applicazione UninaFoodLab. Attraverso la GUI, l'utente può accedere a tutte le funzionalità offerte dal sistema, come la consultazione dei corsi, la gestione del proprio profilo, la prenotazione e la registrazione, e l'utilizzo delle dashboard dedicate. Ogni azione viene gestita in modo reattivo e intuitivo, grazie all'integrazione tra i file FXML, i controller e le classi Boundary.

La progettazione dell'interazione è stata pensata per garantire semplicità d'uso, immediatezza delle risposte e chiarezza nei feedback, sia in caso di successo che di errore. Nei paragrafi successivi verranno analizzate nel dettaglio le principali funzionalità e i flussi di interazione, con esempi pratici e riferimenti al codice implementato.

4.5.1 Homepage Utente

La homepage utente rappresenta il punto di ingresso principale per l'utente visitatore dopo il login. È progettata per offrire una panoramica chiara e immediata delle funzionalità disponibili, come la ricerca e la visualizzazione dei corsi, la gestione del profilo, la consultazione dei corsi a cui si è iscritti e il logout.

La struttura della homepage è definita nel file `homepageutente.fxml`, che organizza la GUI in una sidebar laterale per la navigazione rapida e una sezione centrale dedicata ai contenuti dinamici. La sidebar permette di accedere velocemente alle principali funzionalità, mentre la sezione centrale mostra i corsi disponibili, i filtri di ricerca e la paginazione.

L'interazione è gestita dalle classi `HomepageUtenteBoundary.java` e `HomepageUtenteController.java`, che si occupano rispettivamente della gestione dell'interfaccia e della logica applicativa. Ad esempio, la ricerca dei corsi avviene tramite i filtri di categoria e frequenza, e i risultati vengono visualizzati in modo paginato grazie al controller.

La homepage mostra anche le informazioni dell'utente (nome, cognome, immagine profilo) e offre un'esperienza personalizzata, con feedback immediati sulle azioni compiute (spinner di caricamento, messaggi di errore, aggiornamento dinamico dei contenuti).

4.5.2 Iscrizione ai Corsi

L'iscrizione ai corsi è una delle funzionalità centrali per l'utente. La scena `enrolledcourses.fxml` offre una panoramica dei corsi a cui l'utente è già iscritto, con la possibilità di filtrare per categoria e frequenza tramite i rispettivi `ComboBox`. La paginazione consente di navigare tra i corsi in modo semplice e intuitivo.

L'interazione è gestita dalle classi `EnrolledCoursesBoundary.java` e `EnrolledCoursesController.java`. La boundary si occupa di visualizzare i dati e gestire gli eventi della GUI, mentre il controller implementa la logica di caricamento, filtraggio e visualizzazione dei corsi iscritti. Ad esempio, il metodo `loadEnrolledCourses()` recupera dal database i corsi a cui l'utente è iscritto, applica i filtri selezionati e aggiorna dinamicamente la visualizzazione:

```
public void loadEnrolledCourses() {
    if (boundary != null) boundary.showLoadingIndicator();
```

```
    Thread t = new Thread(() -> {
        try {
            UtenteVisitatore utente = UtenteVisitatore.loggedUser;
            // Recupera i corsi dal database
            utente.getUtenteVisitatoreDao().RecuperaCorsi(utente);
            List<Corso> corsi = utente.getCorsi();
            // Applica i filtri e aggiorna la GUI
            // ...
        } catch (Exception e) {
            e.printStackTrace();
        }
    });
    t.setDaemon(true);
    t.start();
}
```

La boundary gestisce anche la visualizzazione del numero totale di corsi iscritti, lo spinner di caricamento e la navigazione tra le pagine. L'utente può tornare alla homepage, accedere alla gestione account o effettuare il logout tramite i pulsanti laterali.

Questa organizzazione garantisce un'esperienza utente fluida, con feedback immediati e una gestione efficiente anche in presenza di molti corsi iscritti.

Inoltre, la modularità delle componenti come `CardCorsoBoundary` permette di riutilizzare la stessa logica di visualizzazione dei corsi in diverse parti dell'applicazione, mantenendo coerenza e riducendo la duplicazione del codice. La sicurezza e la coerenza dei dati sono garantite dal caricamento asincrono tramite thread dedicati, che evitano blocchi dell'interfaccia e gestiscono correttamente le eccezioni.

La paginazione e i filtri rendono la navigazione tra i corsi iscritti scalabile anche per utenti con molte iscrizioni, mentre la personalizzazione dell'esperienza (visualizzazione nome, immagine profilo, feedback visivi) contribuisce a rendere l'interazione più efficace e gradevole. Tutte queste caratteristiche sono gestite e coordinate all'interno della scena `enrolledcourses.fxml` e delle relative classi boundary e controller.

4.5.3 Visualizzazione Calendario delle Sessioni

L'utente può consultare il calendario delle lezioni e sessioni direttamente dalla scena `calendardialog.fxml`, che offre una vista mensile interattiva e dettagliata. La GUI è composta da una griglia che rappresenta i giorni del mese, una sidebar con i dettagli delle lezioni selezionate e pulsanti per la navigazione tra i mesi. La logica di visualizzazione e interazione è gestita dalle classi `CalendarDialogBoundary.java` e `CalendarDialogController.java`, che si occupano di popolare il calendario con le sessioni reali dell'utente, gestire la selezione dei giorni e mostrare i dettagli delle lezioni.

Quando l'utente seleziona un giorno, vengono visualizzate tutte le lezioni previste per quella data, con informazioni su orario, durata, tipo di sessione (online o in presenza) e descrizione. Il controller gestisce la mappa delle lezioni e aggiorna dinamicamente la GUI, garantendo un'esperienza reattiva e personalizzata. Esempio di codice per la selezione di una data e visualizzazione delle lezioni:

```
private void selectDate(VBox cell, LocalDate date) {
    if (selectedDayCell != null) { /* ... */ }
    selectedDayCell = cell;
    cell.getStyleClass().add("selected");
```

```
    selectedDate = date;
    showLessonDetails(date);
}
```

La modularità della scena consente di adattare la visualizzazione sia per utenti che per chef, mostrando le sessioni pertinenti e permettendo la conferma della presenza per le lezioni in presenza.

4.5.4 Acquisto e Pagamento dei Corsi

L'acquisto di un corso avviene tramite la scena `paymentpage.fxml`, che offre una procedura guidata e sicura per il pagamento online. L'utente può scegliere tra carte di credito salvate o inserire una nuova carta, con validazione automatica dei dati (nome, numero, scadenza, CVC) e feedback immediato sugli errori. La logica di pagamento è gestita dalle classi `PaymentPageBoundary.java` e `PaymentPageController.java`, che si occupano di validare i dati, gestire la persistenza delle carte e iscrivere l'utente al corso selezionato.

La validazione dei dati avviene sia tramite pattern regolari che tramite la classe `CardValidator.java`, che verifica il tipo di carta (Visa/Mastercard) e la validità della scadenza. Esempio di codice per la validazione della scadenza:

```
public static boolean isValidExpiryDate(String expiry) {
    try {
        String[] parts = expiry.split("/");
        int month = Integer.parseInt(parts[0]);
        int year = Integer.parseInt(parts[1]);
        // ...conversione anno e controllo validità...
        LocalDate expiryDate = LocalDate.of(year, month, 1).plusMonths(1).minusDays(1);
        LocalDate today = LocalDate.now();
        return expiryDate.isAfter(today) || expiryDate.isEqual(today);
    } catch (Exception e) {
        return false;
    }
}
```

Al termine della procedura, l'utente riceve un feedback di successo tramite dialog dedicato e il corso viene aggiunto ai corsi iscritti. La modularità della boundary consente di gestire sia carte nuove che salvate, con possibilità di persistenza nel database e visualizzazione delle carte disponibili.

4.5.5 Gestione Account Utente

La gestione dell'account utente è una funzionalità centrale che consente all'utente di visualizzare e modificare i propri dati personali, aggiornare la password, cambiare la foto profilo e accedere alle proprie carte di pagamento. Tutte queste operazioni sono gestite tramite la scena `accountmanagement.fxml`, che offre una GUI intuitiva e suddivisa in sezioni tematiche: informazioni personali, sicurezza, foto profilo e azioni aggiuntive.

La logica applicativa è implementata nelle classi `AccountManagementBoundary.java` e `AccountManagementCont`
La boundary si occupa di gestire gli eventi della GUI e di delegare le operazioni al controller, che interagisce con il database per il recupero e la modifica dei dati utente. Ad esempio, all'inizializzazione vengono caricati i dati reali dell'utente loggato e visualizzati nei rispettivi campi:

```
private void loadUserData() {
    if (loggedUser != null) {
        utenteDao.recuperaDatiUtente(loggedUser);
        originalName = loggedUser.getNome();
        originalSurname = loggedUser.getCognome();
        originalEmail = loggedUser.getEmail();
        originalBirthDate = loggedUser.getDataDiNascita();
        boundary.getNameField().setText(originalName);
        boundary.getSurnameField().setText(originalSurname);
        boundary.getEmailField().setText(originalEmail);
        boundary.getBirthDatePicker().setValue(originalBirthDate);
        boundary.getUserNameLabel().setText(originalName + " " + originalSurname);
        boundary.setProfileImages(propic);
    }
}
```

L'utente può modificare i dati personali (nome, cognome, email, data di nascita) e salvare le modifiche, che vengono validate (ad esempio, controllo email valida e età minima) e poi aggiornate nel database. È possibile anche cambiare la password, con verifica della password attuale e conferma della nuova password. In caso di errore, vengono mostrati messaggi di feedback chiari e immediati.

La gestione della foto profilo avviene tramite un file chooser che consente di selezionare un'immagine dal proprio dispositivo. Il controller si occupa di validare il formato, salvare l'immagine nella directory dedicata e aggiornare il percorso nel database, mostrando la preview aggiornata nella GUI:

```
public void changePhoto() {
    FileChooser fileChooser = new FileChooser();
    // ...configurazione fileChooser...
    File selectedFile = fileChooser.showOpenDialog(stage);
    if (selectedFile != null) {
        // ...validazione e copia file...
        loggedUser.setUrl_Propic(relativePath);
        utenteDao.ModificaUtente(loggedUser);
        boundary.setProfileImages(relativePath);
    }
}
```

La scena offre anche la possibilità di accedere alla pagina delle carte utente, tornare alla homepage o ai corsi iscritti, e di effettuare il logout tramite pulsanti dedicati. Tutte le operazioni sono accompagnate da dialog di conferma o successo, garantendo un'esperienza utente sicura e trasparente.

La modularità tra boundary e controller, insieme all'uso di metodi getter/setter e all'integrazione con il database, consente una gestione efficiente e scalabile dell'account utente, con possibilità di estensione per future funzionalità.

4.5.6 Gestione Carte Utente

La gestione delle carte di pagamento è accessibile tramite la scena `usercards.fxml`, che consente all'utente di visualizzare, aggiungere ed eliminare le proprie carte salvate. L'interfaccia è suddivisa in due sezioni principali: il form per l'aggiunta di una nuova carta e la lista delle carte già memorizzate.

La logica applicativa è gestita dalle classi `UserCardsBoundary.java` e `UserCardsController.java`. All'apertura della pagina, vengono caricate dal database tutte le carte associate all'utente e visualizzate nella lista. Ogni carta mostra il nome del titolare, le ultime quattro cifre, la data di scadenza e il circuito (Visa/Mastercard), con una grafica chiara e badge identificativo.

Per aggiungere una nuova carta, l'utente deve compilare il form con nome, numero, scadenza e CVV. La validazione dei dati avviene sia tramite pattern regolari che tramite la classe `CardValidator.java`, che verifica il tipo di carta e la validità della scadenza. Esempio di validazione:

```
if (!CardValidator.isValidCardType(number)) {
    boundary.showFieldError("cardNumber", "Tipo di carta non supportato. Accettiamo solo Visa e
    return;
}
if (!CardValidator.isValidExpiryDate(expiry)) {
    boundary.showFieldError("expiry", "La carta è scaduta o la data non è valida");
    return;
}
```

Se la carta è valida, viene salvata nel database e associata all'utente tramite le DAO dedicate. Un dialog di successo conferma l'operazione e la lista viene aggiornata in tempo reale.

L'utente può eliminare una carta selezionata dalla lista, con conferma e aggiornamento immediato della visualizzazione. Tutti i campi del form sono dotati di feedback visivi per errori e formattazione automatica (ad esempio, spaziatura del numero carta e formato scadenza MM/AA).

La modularità tra boundary e controller garantisce una gestione sicura e intuitiva dei metodi di pagamento, con possibilità di estensione per future integrazioni (es. impostazione carta predefinita, supporto ad altri circuiti). L'integrazione con il database assicura la persistenza e la coerenza dei dati tra le varie scene dell'applicazione.

Criteri di validazione delle carte La procedura di aggiunta e salvataggio di una carta di credito prevede una serie di controlli formali e logici per garantire la correttezza e la sicurezza dei dati inseriti. I principali criteri di validazione sono:

- **Nome del titolare**: deve essere composto da almeno nome e cognome, solo lettere e spazi, lunghezza compresa tra 2 e 50 caratteri.
- **Numero carta**: deve essere composto da 16 cifre, senza caratteri speciali o lettere. La formattazione automatica inserisce uno spazio ogni 4 cifre per facilitare la lettura.
- **Tipo di carta**: sono accettate solo carte Visa (iniziano con 4) e Mastercard (iniziano con 5 o con prefisso tra 2221 e 2720). Il controllo avviene tramite la funzione `isValidCardType()`:

```
public static boolean isValidCardType(String cardNumber) {
    if (cardNumber == null) return false;
    if (cardNumber.startsWith("4")) return true;
    if (cardNumber.startsWith("5")) return true;
    if (cardNumber.length() >= 4) {
        String prefix = cardNumber.substring(0, 4);
        int prefixNum = Integer.parseInt(prefix);
        if (prefixNum >= 2221 && prefixNum <= 2720) return true;
    }
    return false;
}
```

- **Data di scadenza**: deve essere nel formato MM/AA, con mese tra 01 e 12 e anno non scaduto. La funzione `isValidExpiryDate()` verifica che la data sia valida e non precedente a quella attuale.
- **CVC**: deve essere composto da 3 o 4 cifre numeriche.

In caso di errore, la GUI mostra un messaggio specifico accanto al campo errato, impedendo il salvataggio della carta fino alla correzione. Questi criteri garantiscono che solo carte valide e utilizzabili vengano memorizzate e associate all'utente.

4.6 Interazione con lo Chef

l'interazione con lo chef è un aspetto cruciale del progetto UninaFoodLab. Gli chef sono i professionisti che gestiscono i corsi culinari, e la loro interfaccia è progettata per essere intuitiva e funzionale. La sezione dedicata agli chef consente loro di:

- Creare e gestire i propri corsi, inclusa la pianificazione delle lezioni, online o/e in presenza, e la definizione delle ricette
- Visualizzare le prenotazioni dei corsi e gestire le richieste degli utenti
- Aggiornare le informazioni sui corsi e sulle ricette, garantendo che gli utenti abbiano sempre accesso alle informazioni più aggiornate

Questa interazione è facilitata da un'interfaccia grafica realizzata con JavaFX, che offre un'esperienza utente fluida e reattiva. Gli chef possono accedere alla loro sezione tramite un login sicuro, garantendo che solo gli utenti autorizzati possano gestire i corsi e le informazioni ad essi associate.

4.6.1 Homepage dello Chef

L'homepage dello chef rappresenta il punto di ingresso principale per la gestione dei corsi e delle attività didattiche. La scena `homepagechef.fxml` è progettata per offrire una panoramica chiara e interattiva dei corsi creati dallo chef, con funzionalità di ricerca, filtraggio e paginazione.

La GUI è suddivisa in una sidebar laterale per la navigazione (accesso rapido a creazione corso, report mensile, gestione account e logout) e una sezione centrale che mostra i corsi in formato card. Ogni card visualizza le informazioni principali del corso (titolo, descrizione, date, frequenza, tipologie di cucina, numero massimo di partecipanti, immagine e dati chef).

La logica applicativa è gestita dalle classi `HomepageChefBoundary.java` e `HomepageChefController.java`. All'inizializzazione vengono caricati i dati reali dello chef loggato e i corsi associati, con possibilità di filtrare per categoria e frequenza tramite le `ComboBox` dedicate.

La visualizzazione dei corsi è dinamica e supporta la paginazione, con un massimo di 12 card per pagina. Il controller gestisce il caricamento dei corsi dal database, l'applicazione dei filtri e l'aggiornamento della GUI:

```
for (Corso corso : corsi) {
    // Applica i filtri
    if (!matchCategoria || !matchFrequenza) continue;
    FXMLLoader loader = new FXMLLoader(getClass().getResource("/fxml/cardcorso.fxml"));
    Node card = loader.load();
    CardCorsoBoundary boundary = loader.getController();
    boundary.setChefMode(true);
```

```
    boundary.setCorso(corso);
    // ...imposta dati corso...
    allCourseCards.add(card);
}
updateCourseCards();
```

La paginazione consente di navigare tra le pagine dei corsi tramite i pulsanti "Precedente" e "Successiva", con aggiornamento del numero di pagina visualizzato. In assenza di corsi, viene mostrato un messaggio di invito alla creazione del primo corso.

L'homepage chef offre inoltre accesso diretto alle funzionalità di creazione corso, report mensile e gestione account, garantendo un'esperienza utente fluida e personalizzata. Tutte le operazioni sono accompagnate da feedback visivi e dialog di conferma, con integrazione tra boundary, controller e database.

4.6.2 Creazione Corso

La creazione di un nuovo corso rappresenta una delle funzionalità più articolate e centrali per lo chef. La scena `createcourse.fxml` offre una form completa e guidata, suddivisa in sezioni tematiche che coprono tutti gli aspetti necessari per la definizione di un corso di cucina professionale.

Struttura della GUI La GUI è organizzata in una sidebar per la navigazione e una sezione centrale con la form di creazione. Le sezioni principali sono:

- **Immagine del corso**: upload e preview dell'immagine rappresentativa.
- **Informazioni base**: nome, descrizione, tipologie di cucina (almeno una obbligatoria), prezzo, numero massimo di partecipanti.
- **Date e programmazione**: selezione intervallo date, frequenza delle sessioni, tipo di lezione (in presenza, telematica, entrambi).
- **Dettagli lezioni in presenza**: giorni della settimana, orario, durata, luogo (città, via, CAP).
- **Dettagli lezioni telematiche**: applicazione utilizzata, meeting code, giorni, orario, durata.
- **Modalità ibrida**: configurazione avanzata per corsi che prevedono sia lezioni in presenza che online, con UI dinamica per la selezione dei giorni e la gestione delle sessioni.
- **Ricette e ingredienti**: per ogni sessione, possibilità di associare una o più ricette, con ingredienti dettagliati e quantità.

Logica applicativa e flussi La logica di creazione corso è gestita dalle classi `CreateCourseBoundary.java` e `CreateCourseController.java`. All'inizializzazione vengono caricati i dati dello chef loggato, le opzioni di frequenza, tipologie di cucina e applicazioni telematiche dal database. La form è reattiva: la selezione della frequenza e del tipo di lezione aggiorna dinamicamente le sezioni visibili e i campi obbligatori.

La validazione dei dati è molto rigorosa e avviene sia lato GUI (formattazione, pattern, limiti di lunghezza) che lato controller. Esempi di validazione:

```
// Controlla che il prezzo sia un numero positivo con max 2 decimali
private boolean isValidPrice(String priceText) {
    if (priceText == null || priceText.trim().isEmpty()) return false;
    return priceText.matches("\\d+(\\.\\d{1,2})?");
}

// Controlla che il CAP sia composto da 5 cifre
private boolean isValidCAP(String capText) {
    return capText != null && capText.matches("\\d{5}");
}

// Controlla che la durata sia tra 1 e 480 minuti (max 8 ore)
private boolean isValidDurationRange(String durationText) {
    try {
        int durata = Integer.parseInt(durationText);
        return durata >= 1 && durata <= 480;
    } catch (NumberFormatException e) {
        return false;
    }
}

// Controlla che la descrizione sia tra 1 e 60 parole
private boolean isValidDescription(String text) {
    if (text == null) return false;
    int wordCount = text.trim().split("\\s+").length;
    return wordCount >= 1 && wordCount <= 60;
}
```

Ogni metodo di validazione viene invocato in tempo reale durante la compilazione della form, bloccando la creazione del corso e mostrando un messaggio di errore contestuale se il dato non è valido. Questo garantisce che i dati inseriti siano sempre corretti e coerenti con i vincoli di business.

Calcolo automatico delle date delle sessioni Il calcolo delle date delle sessioni in presenza è fondamentale per la corretta programmazione del corso. Il controller implementa una logica che, dato un intervallo di date e una lista di giorni della settimana selezionati, genera tutte le date corrispondenti:

```
private List<LocalDate> calcolaDateSessioniPresenza(LocalDate inizio, LocalDate fine, List<Stri
    List<LocalDate> dateSessioni = new ArrayList<>();
    if (inizio == null || fine == null || giorniSettimana == null || giorniSettimana.isEmpty()
    LocalDate data = inizio;
    while (!data.isAfter(fine)) {
        String giorno = getItalianDayName(data); // es. "Lunedì"
        if (giorniSettimana.contains(giorno)) {
            dateSessioni.add(data);
        }
        data = data.plusDays(1);
    }
```

```
    return dateSessioni;
}
```

Questa funzione viene utilizzata per generare la UI delle ricette e per la persistenza delle sessioni nel database. In modalità ibrida, il calcolo è ancora più avanzato: la UI consente di configurare ogni sessione separatamente, evitando sovrapposizioni e garantendo la coerenza del calendario.

Spiegazione: Il metodo scorre tutte le date tra inizio e fine, controlla se il giorno corrisponde a uno di quelli selezionati (ad esempio, "Lunedì" o "Giovedì") e, in caso positivo, aggiunge la data alla lista delle sessioni. Questo permette di programmare corsi con frequenze e giorni personalizzati, adattandosi alle esigenze dello chef e degli utenti.

Gestione delle sessioni La programmazione delle sessioni è uno degli aspetti più avanzati. In base alla frequenza e ai giorni selezionati, il controller calcola automaticamente tutte le date delle lezioni, sia in presenza che online. Per la modalità ibrida, la UI consente di configurare ogni sessione separatamente, evitando sovrapposizioni e garantendo la coerenza del calendario.

Esempio di calcolo delle date:

```
private List<LocalDate> calcolaDateSessioniPresenza(LocalDate inizio, LocalDate fine, List<Stri
```

La boundary aggiorna la UI delle ricette per ogni sessione, permettendo di associare ricette e ingredienti specifici.

Gestione ricette e ingredienti Per ogni sessione, lo chef può aggiungere una o più ricette, ciascuna con una lista di ingredienti e quantità. La UI consente di aggiungere, modificare e rimuovere ricette e ingredienti in modo dinamico. La validazione garantisce che ogni ricetta sia completa e corretta prima del salvataggio.

Salvataggio e persistenza Al termine della compilazione, il controller valida nuovamente tutti i dati e salva il corso nel database, insieme alle sessioni, ricette e ingredienti associati. Il salvataggio avviene in modo transazionale, garantendo la coerenza dei dati. In caso di successo, viene mostrato un dialog di conferma e lo chef viene reindirizzato all'homepage.

Esempi di codice e riferimenti

```
public void createCourse() {
    // Validazione campi
    // Calcolo date sessioni
    // Salvataggio corso, sessioni, ricette, ingredienti
    // Feedback e redirect
}
```

I file principali coinvolti sono: `createcourse.fxml`, `CreateCourseBoundary.java`, `CreateCourseController.jav` le DAO per corso, sessioni, ricette e ingredienti.

Utilizzo delle Mappe nella Creazione del Corso Nella logica di creazione corso, le **mappe** (`Map`) sono il fulcro della gestione strutturata dei dati tra UI, controller e database. Di seguito una spiegazione dettagliata di come vengono utilizzate e dei vantaggi architetturali che offrono:

- `Map<LocalDate, ObservableList<Ricetta>>`: questa mappa associa ogni data di sessione (sia in presenza che ibrida) alla lista di ricette da preparare. Quando lo chef seleziona giorni e intervallo date, il controller genera tutte le date possibili e la boundary crea per ciascuna un container UI dedicato. L'utente può aggiungere, modificare o rimuovere ricette per ogni data, e la mappa viene aggiornata in tempo reale. In fase di salvataggio, il controller itera la mappa per persistere ogni sessione e le relative ricette nel database, garantendo che la struttura logica della programmazione sia mantenuta.

- `Map<String, CheckBox>`: questa mappa collega ogni giorno della settimana a un `CheckBox` nella UI. Serve per gestire la selezione dinamica dei giorni, validare che il numero di giorni selezionati sia coerente con la frequenza scelta, e aggiornare la UI in modo reattivo. Ad esempio, se la frequenza è "bisettimanale", la logica consente di selezionare solo due giorni e la mappa permette di abilitare/disabilitare i checkbox in modo automatico.

- `Map<LocalDate, ObservableList<Ricetta>>` per le sessioni ibride: in modalità "Entrambi", la mappa viene usata per gestire separatamente le ricette di ogni sessione, sia online che in presenza. La UI consente di configurare ogni sessione con dettagli specifici (tipo, orario, ricette), e la mappa garantisce che non ci siano sovrapposizioni o errori di associazione. Questo approccio permette una programmazione avanzata e personalizzata, adattabile a corsi complessi.

Gestione e interazione con la UI: Le mappe sono strettamente integrate con la UI JavaFX. Ogni volta che l'utente interagisce con la form (aggiunge una ricetta, seleziona un giorno, modifica una sessione), la mappa viene aggiornata e la UI riflette immediatamente i cambiamenti. Questo garantisce una validazione continua e una coerenza tra i dati visualizzati e quelli che verranno salvati.

Validazione e robustezza: L'utilizzo delle mappe semplifica la validazione dei dati: è possibile verificare rapidamente se tutte le sessioni hanno almeno una ricetta, se i giorni selezionati sono corretti, e se non ci sono duplicati o errori logici. La struttura a mappa consente controlli efficienti e riduce la possibilità di errori utente.

Persistenza e flessibilità: In fase di salvataggio, il controller itera le mappe per creare le query di inserimento nel database, associando ogni sessione alle sue ricette e ingredienti. Questo approccio è facilmente estendibile: se in futuro si volessero aggiungere nuove tipologie di dati (es. allergeni, note, feedback), basterebbe estendere la struttura della mappa senza modificare la logica centrale.

Esempio di codice pratico:

```
Map<LocalDate, ObservableList<Ricetta>> sessioniPresenza = boundary.getSessionePresenzaRicette(
for (Map.Entry<LocalDate, ObservableList<Ricetta>> entry : sessioniPresenza.entrySet()) {
    LocalDate data = entry.getKey();
    ObservableList<Ricetta> ricette = entry.getValue();
    // ...salvataggio sessione e ricette...
}
```

Approccio ibrido e personalizzazione: Nella modalità "Entrambi", la mappa consente di gestire sessioni con caratteristiche diverse (online/presenza), orari e ricette specifiche. La UI genera dinamicamente i container per ogni sessione, e la mappa tiene traccia delle configurazioni, permettendo una programmazione avanzata e personalizzata.

Vantaggi architetturali:

- Separazione tra logica di presentazione (UI) e logica di business (controller/database)

- Facilità di estensione e manutenzione
- Validazione efficiente e centralizzata
- Maggiore robustezza e coerenza dei dati
- Esperienza utente migliorata grazie a feedback immediati e UI dinamica

In sintesi, l'uso delle mappe nella creazione del corso è una scelta architetturale che garantisce flessibilità, robustezza e scalabilità, semplificando la gestione di dati complessi e migliorando l'esperienza utente.

Personalizzazione e modularità La modularità della boundary e del controller consente di estendere facilmente la form per nuove funzionalità (es. nuove tipologie di lezione, filtri avanzati, gestione allergeni). La UI è pensata per essere accessibile e personalizzabile, con feedback visivi, tooltips e messaggi di aiuto contestuali.

4.6.3 Modifica dei Corsi

La modifica dei corsi consente allo chef di aggiornare le informazioni di un corso già esistente, mantenendo la coerenza dei dati e la tracciabilità delle modifiche. La scena `editcourse.fxml` offre una form strutturata e intuitiva, che riprende la logica della creazione ma con vincoli specifici per la modifica.

Struttura della GUI La GUI è composta da una sezione header con il nome del corso (non modificabile), una form centrale suddivisa in sezioni (informazioni generali, date e orari, giorni della settimana, location, sessioni online, ricette), e una sezione di azioni (annulla, salva modifiche).

I campi non modificabili (nome corso, date, tipo corso, frequenza) sono disabilitati per garantire la coerenza storica. I campi modificabili includono descrizione, numero massimo partecipanti, orari, durata, location, ricette e ingredienti.

Logica applicativa e flussi La logica di modifica è gestita dalle classi `EditCourseBoundary.java` e `EditCourseController.java`. All'apertura della scena, vengono caricati i dati reali del corso dal database e popolati nei rispettivi campi. La UI si adatta dinamicamente al tipo di corso (in presenza, telematica, entrambi), mostrando solo le sezioni rilevanti.

La validazione dei dati avviene sia lato GUI che controller. Esempi di validazione:

```
// CAP: deve essere composto da 5 cifre
private boolean isValidCAP(String capText) {
    return capText != null && capText.matches("\\d{5}");
}
// Descrizione: tra 1 e 60 parole
private boolean isValidDescription(String text) {
    if (text == null) return false;
    int wordCount = text.trim().split("\\s+").length;
    return wordCount >= 1 && wordCount <= 60;
}
```

La form consente il salvataggio solo se almeno un campo è stato modificato rispetto ai dati originali, grazie a un binding tra i campi e il tasto "Salva Modifiche".

Utilizzo delle Mappe nella Modifica dei Corsi Anche nella logica di modifica dei corsi, le **mappe** (`Map`) svolgono un ruolo centrale per gestire in modo strutturato l'associazione tra sessioni, ricette e ingredienti. In particolare:

- `Map<LocalDate, ObservableList<Ricetta>>`: questa mappa viene utilizzata per caricare e gestire le ricette associate a ciascuna sessione in presenza. Quando il controller recupera i dati dal database, costruisce una mappa che collega ogni data di sessione alla lista di ricette corrispondenti. Questo consente di visualizzare, modificare o rimuovere ricette in modo dinamico per ogni sessione futura.
- `Map<LocalDate, ObservableList<Ricetta>>` per le sessioni ibride: in corsi di tipo "Entrambi", la mappa permette di gestire separatamente le ricette per ogni sessione, sia online che in presenza, garantendo coerenza e flessibilità nella modifica.
- `Map<String, CheckBox>`: se la UI prevede la selezione dei giorni, la mappa collega i giorni della settimana ai rispettivi checkbox, facilitando la validazione e l'aggiornamento della UI.

Gestione dinamica e validazione: Durante la modifica, la mappa viene aggiornata ogni volta che lo chef aggiunge, modifica o elimina una ricetta o un ingrediente. Questo garantisce che la UI sia sempre sincronizzata con i dati effettivi e che la validazione sia centralizzata ed efficiente.

Persistenza e coerenza: In fase di salvataggio, il controller itera la mappa per aggiornare solo le sessioni e le ricette effettivamente modificate, evitando errori e ridondanze. Questo approccio garantisce la coerenza tra i dati visualizzati e quelli persistiti nel database.

Esempio di codice pratico:

```
Map<LocalDate, ObservableList<Ricetta>> sessioniPresenza = boundary.getSessioniPresenzaModifica
for (Map.Entry<LocalDate, ObservableList<Ricetta>> entry : sessioniPresenza.entrySet()) {
    LocalDate data = entry.getKey();
    ObservableList<Ricetta> ricette = entry.getValue();
    // ...aggiornamento/modifica sessione e ricette...
}
```

Vantaggi:

- Gestione strutturata e modulare delle modifiche
- Validazione centralizzata e robusta
- Aggiornamento efficiente solo dei dati modificati
- Maggiore flessibilità per corsi complessi (ibridi, multi-sessione)
- Esperienza utente migliorata grazie a UI dinamica e feedback immediati

In sintesi, l'utilizzo delle mappe nella modifica dei corsi permette di mantenere la coerenza tra UI e database, semplifica la gestione delle modifiche e garantisce robustezza e scalabilità anche in scenari complessi.

Gestione delle sessioni e ricette Le sessioni (in presenza, telematiche, ibride) vengono caricate dal database e visualizzate nella UI. Solo le sessioni future sono editabili, mentre quelle passate sono visualizzate in sola lettura. Per ogni sessione, lo chef può modificare le ricette e gli ingredienti associati, aggiungere nuove ricette o rimuovere quelle non più necessarie.

La boundary gestisce la UI dinamica delle ricette tramite container dedicati, permettendo di aggiungere, modificare e rimuovere ricette e ingredienti in modo intuitivo. Esempio di aggiunta ricetta:

```
public void addRecipeToContainer(String recipeName, String[] ingredients,
                                 String[] quantities, String[] units, LocalDate sessionDate) {
    // Crea box ricetta e aggiungi ingredienti
}
```

La validazione garantisce che ogni ricetta sia completa e corretta prima del salvataggio.

Salvataggio e persistenza Al salvataggio, il controller confronta i dati modificati con quelli originali, aggiorna solo i campi effettivamente cambiati e salva le modifiche nel database. Il salvataggio avviene in modo transazionale, garantendo la coerenza dei dati. In caso di successo, viene mostrato un dialog di conferma e lo chef viene reindirizzato all'homepage.

Esempi di codice e riferimenti Il metodo `saveCourse()` è il cuore della logica di salvataggio delle modifiche. Di seguito un esempio esteso e commentato:

```
public void saveCourse() {
    // 1. Controlla se ci sono modifiche
    if (!hasAnyFieldChanged()) return;

    // 2. Validazione dei campi modificabili
    if (!isValidDescription(descriptionArea.getText())) {
        boundary.showError(descriptionErrorLabel, "La descrizione deve essere tra 1 e 60 parole
        return;
    }
    if (!isValidCAP(capField.getText())) {
        boundary.showError(capErrorLabel, "Il CAP deve essere composto da 5 cifre.");
        return;
    }
    // ...altre validazioni...

    // 3. Aggiorna solo i dati modificati
    if (!Objects.equals(courseNameField.getText().trim(), currentCourse.getNome())) {
        currentCourse.setNome(courseNameField.getText().trim());
    }
    if (!Objects.equals(descriptionArea.getText().trim(), currentCourse.getDescrizione())) {
        currentCourse.setDescrizione(descriptionArea.getText().trim());
    }
    // ...aggiorna altri campi...

    // 4. Gestione delle sessioni e ricette
    // Aggiorna solo le sessioni future e le ricette associate
```

```
    for (SessioniInPresenza sessione : currentCourse.getSessioniInPresenza()) {
        if (sessione.getData().isAfter(LocalDate.now())) {
            // Aggiorna ricette e ingredienti
            // ...codice di aggiornamento...
        }
    }
    // ...gestione sessioni telematiche e ibride...

    // 5. Salvataggio nel database tramite DAO
    try {
        CorsoDao corsoDao = new CorsoDao();
        corsoDao.updateCorso(currentCourse);
        // Aggiorna sessioni e ricette
        // ...codice di persistenza...
        boundary.showCourseUpdateSuccessDialog();
        boundary.goBack(); // Torna all'homepage chef
    } catch (Exception e) {
        boundary.showAlert("Errore", "Impossibile salvare le modifiche: " + e.getMessage());
    }
}
```

Spiegazione: Il metodo segue questi passaggi:

1. Verifica che ci siano modifiche effettive rispetto ai dati originali.
2. Valida i campi modificabili, mostrando errori contestuali se necessario.
3. Aggiorna solo i dati effettivamente cambiati, mantenendo la coerenza storica.
4. Gestisce l'aggiornamento delle sessioni e delle ricette, modificando solo quelle future.
5. Salva tutte le modifiche nel database tramite le DAO dedicate, mostrando un dialog di successo o errore.

Questo approccio garantisce efficienza, sicurezza e tracciabilità delle modifiche, evitando errori e mantenendo la coerenza dei dati tra le varie componenti dell'applicazione.

I file principali coinvolti sono: `editcourse.fxml`, `EditCourseBoundary.java`, `EditCourseController.java`, le DAO per corso, sessioni, ricette e ingredienti.

Personalizzazione e modularità La modularità della boundary e del controller consente di estendere facilmente la form di modifica per nuove funzionalità (es. gestione avanzata delle sessioni, filtri, validazioni aggiuntive). La UI è pensata per essere accessibile e personalizzabile, con feedback visivi, tooltips e messaggi di aiuto contestuali.

4.6.4 Report Mensile

Il Report Mensile è una funzionalità avanzata pensata per offrire allo chef una panoramica dettagliata delle proprie attività, performance e guadagni nel corso di un mese specifico. La scena `monthlyreport.fxml` presenta una dashboard interattiva e ricca di statistiche, grafici e indicatori chiave, permettendo di monitorare l'andamento dei corsi e delle sessioni svolte.

Struttura della GUI La GUI è suddivisa in:

- Sidebar di navigazione con profilo chef, logo, e pulsanti per le principali azioni (corsi, creazione corso, report, account, logout).
- Sezione centrale con selettori di mese/anno, pulsante di aggiornamento, e titolo dinamico del report.
- Card statistiche: corsi totali, sessioni online, sessioni pratiche, guadagno mensile.
- Grafici: PieChart per la distribuzione delle sessioni, BarChart per le ricette per sessione.
- Statistiche ricette: media, massimo, minimo, totale ricette realizzate.

Logica applicativa La logica è gestita dalle classi `MonthlyReportBoundary.java` e `MonthlyReportController.` All'inizializzazione, vengono caricati i dati dello chef loggato e popolati i controlli della scena. Il controller gestisce la selezione di mese/anno, il caricamento dei dati dal database tramite le DAO dedicate (`GraficoChefDao`), e l'aggiornamento dinamico delle statistiche e dei grafici.

Focus sul metodo `loadReportData`: Questo metodo è il cuore della logica del report mensile. Si occupa di:

1. Recuperare il mese e l'anno selezionati dalla UI.
2. Interrogare la DAO per ottenere tutte le statistiche richieste (numero corsi, sessioni, ricette, guadagni, ecc.).
3. Calcolare eventuali metriche derivate (es. totale ricette = (sessioni online + pratiche) × media ricette).
4. Aggiornare le label e i grafici della scena in modo reattivo.

Esempio esteso e commentato:

```
private void loadReportData() {
    // 1. Recupera mese e anno selezionati dalla UI
    Integer selectedYear = yearComboBox.getValue();
    int mese = monthComboBox.getSelectionModel().getSelectedIndex() + 1;
    int anno = (selectedYear != null) ? selectedYear : LocalDate.now().getYear();

    // 2. Interroga la DAO per tutte le statistiche
    GraficoChef grafico = new GraficoChef();
    grafico.setNumeroMassimo(graficoChefDao.RicavaMassimo(chef, mese, anno));
    grafico.setNumeroMinimo(graficoChefDao.RicavaMinimo(chef, mese, anno));
    grafico.setMedia(graficoChefDao.RicavaMedia(chef, mese, anno));
    grafico.setNumeriCorsi(graficoChefDao.RicavaNumeroCorsi(chef, mese, anno));
    grafico.setNumeroSessioniInPresenza(graficoChefDao.RicavaNumeroSessioniInPresenza(chef, mes
    grafico.setNumerosessionitelematiche(graficoChefDao.RicavaNumeroSesssioniTelematiche(chef,
    double monthlyEarnings = graficoChefDao.ricavaGuadagno(chef, mese, anno);

    // 3. Calcola metriche derivate
    int totalRecipes = (int) Math.round((grafico.getNumeroSessioniInPresenza() + grafico.getNum
```

```
    // 4. Aggiorna le label statistiche
    updateStatistics(grafico, monthlyEarnings, totalRecipes);

    // 5. Aggiorna i grafici (PieChart e BarChart)
    updateChartsData(grafico, monthlyEarnings);
}
```

Spiegazione: Il metodo `loadReportData` garantisce che tutte le informazioni visualizzate siano sempre aggiornate e coerenti con i dati reali del database. Ogni chiamata alle DAO è ottimizzata per restituire solo i dati necessari per il periodo selezionato, evitando calcoli ridondanti e migliorando le performance della UI. L'aggiornamento dei grafici avviene in modo reattivo, senza ricreare gli oggetti, per una migliore esperienza utente.

Dati visualizzati Le statistiche includono:

- Numero totale di corsi attivi nel mese
- Numero di sessioni online e pratiche
- Guadagno mensile totale
- Media, massimo, minimo e totale ricette realizzate per sessione

I dati sono estratti tramite metodi come `RicavaNumeroCorsi`, `RicavaNumeroSessioniInPresenza`, `RicavaNumeroSesssioniTelematiche`, `ricavaGuadagno`, `RicavaMedia`, `RicavaMassimo`, `RicavaMinimo` della classe `GraficoChefDao`.

Gestione dei grafici I grafici sono aggiornati in modo dinamico:

- **PieChart**: mostra la proporzione tra sessioni online e pratiche.
- **BarChart**: visualizza la media, il massimo e il minimo di ricette per sessione.

I dati sono aggiornati senza ricreare gli oggetti, garantendo fluidità e reattività della UI.

Flusso di aggiornamento Al cambio di mese/anno o al click su "Aggiorna Report", il controller ricarica i dati e aggiorna tutte le statistiche e i grafici. Il binding tra i controlli e i dati garantisce che la UI sia sempre coerente con le informazioni correnti.

Codice di esempio

```
public void updateReport() {
    updateMonthYearLabel();
    loadReportData();
}
```

Personalizzazione e modularità La modularità della boundary e del controller consente di estendere facilmente il report per nuove metriche (es. andamento storico, filtri avanzati, esportazione dati). La UI è pensata per essere accessibile, con feedback visivi e messaggi contestuali.

File e metodi principali

- `monthlyreport.fxml`: definizione della scena e dei controlli.
- `MonthlyReportBoundary.java`: gestione della UI e degli eventi.
- `MonthlyReportController.java`: logica di caricamento dati, aggiornamento statistiche e grafici.
- `GraficoChefDao.java`: accesso ai dati statistici e di guadagno dal database.

In sintesi, il Report Mensile offre allo chef uno strumento potente per monitorare le proprie attività, analizzare le performance e ottimizzare la gestione dei corsi, con una UI moderna e dati sempre aggiornati.

4.6.5 Gestione Account Chef

La gestione dell'account chef è una funzionalità centrale che consente allo chef di visualizzare e modificare i propri dati personali e professionali in modo sicuro e intuitivo. La scena `accountmanagementchef.fxml` offre una form strutturata e suddivisa in sezioni, con campi dedicati e controlli di validazione.

Cosa si può modificare Lo chef può aggiornare:

- **Foto profilo**: caricamento e anteprima immediata, con salvataggio automatico nel database e nelle risorse.
- **Nome e cognome**: campi testuali, validati per non essere vuoti.
- **Email**: campo testuale, validato per formato email corretto.
- **Data di nascita**: selezione tramite DatePicker, con controllo sull'età minima (18 anni).
- **Descrizione professionale**: textarea per presentazione e competenze.
- **Anni di esperienza**: campo numerico, accetta solo valori interi tra 0 e 50.
- **Password**: cambio password con verifica della password attuale, nuova password e conferma.

Struttura della GUI La GUI è suddivisa in:

- Sidebar di navigazione (corsi, creazione corso, report, account, logout).
- Sezione centrale con foto profilo, pulsante "Cambia Foto", e form suddivisa in "Informazioni Personali", "Informazioni Professionali" e "Sicurezza".
- Pulsanti "Annulla" e "Salva Modifiche" per gestire i flussi di salvataggio e ripristino.

Logica applicativa e validazione La logica è gestita dalle classi `AccountManagementChefBoundary.java` e `AccountManagementChefController.java`. All'inizializzazione, vengono caricati i dati reali dello chef loggato dal database e popolati nei rispettivi campi. La validazione avviene in tempo reale e solo sui campi modificati:

- Nome, cognome, email, descrizione: non vuoti, email con "@".
- Data di nascita: almeno 18 anni.
- Anni di esperienza: solo numeri, tra 0 e 50.
- Password: verifica password attuale, nuova password e conferma.
- Foto profilo: verifica formato immagine (PNG, JPG, JPEG, GIF), copia automatica in resources.

Solo se almeno un campo è stato modificato rispetto ai dati originali, il pulsante "Salva Modifiche" consente il salvataggio. In caso di errore di validazione, viene mostrato un messaggio contestuale.

Flusso di salvataggio Al click su "Salva Modifiche":

1. Vengono confrontati i dati attuali con quelli originali.
2. Viene eseguita la validazione sui campi modificati.
3. Se la foto profilo è stata cambiata, viene salvata e aggiornata nel database.
4. Se la password è stata cambiata, viene verificata e aggiornata.
5. Tutte le modifiche vengono salvate tramite la DAO (`ChefDao.ModificaUtente`).
6. Viene mostrato un dialog di conferma o errore.

Il pulsante "Annulla" ripristina i dati originali e cancella eventuali modifiche non salvate.

Codice di esempio **Focus sul metodo `saveChanges`:** Questo metodo è il cuore della logica di salvataggio delle modifiche all'account chef. Si occupa di:

1. Confrontare i dati attuali con quelli originali per rilevare le modifiche.
2. Validare solo i campi effettivamente cambiati (nome, cognome, email, data di nascita, descrizione, anni di esperienza, foto profilo, password).
3. Gestire la modifica della foto profilo: verifica formato, copia in resources, aggiornamento path nel database.
4. Gestire la modifica della password: verifica password attuale, nuova password e conferma.
5. Salvare tutte le modifiche tramite la DAO (`ChefDao.ModificaUtente`).
6. Mostrare un dialog di conferma o errore all'utente.

Esempio esteso e commentato:

```
public void saveChanges() {
    // 1. Confronta i dati attuali con quelli originali
    boolean changed = false;
    if (name != null && !name.equals(originalName) && !name.trim().isEmpty()) {
        loggedChef.setNome(name);
        changed = true;
    }
    // ...ripeti per cognome, email, descrizione, anni di esperienza...

    // 2. Validazione email
    if (email != null && !email.equals(originalEmail) && !email.trim().isEmpty()) {
        if (!email.contains("@")) {
            boundary.showErrorMessage("Inserisci un'email valida.");
            return;
        }
        loggedChef.setEmail(email);
        changed = true;
    }

    // 3. Validazione data di nascita
    if (birthDateValue != null && !birthDateValue.equals(originalBirthDate)) {
        if (Period.between(birthDateValue, LocalDate.now()).getYears() < 18) {
            boundary.showErrorMessage("Devi avere almeno 18 anni.");
            return;
        }
        loggedChef.setDataDiNascita(birthDateValue);
        changed = true;
    }

    // 4. Validazione anni di esperienza
    if (experienceYears != null && !experienceYears.equals(originalExperienceYears) && !experie
        try {
            int years = Integer.parseInt(experienceYears.trim());
            if (years < 0 || years > 50) {
                boundary.showErrorMessage("Gli anni di esperienza devono essere tra 0 e 50.");
                return;
            }
            loggedChef.setAnniDiEsperienza(years);
            changed = true;
        } catch (NumberFormatException e) {
            boundary.showErrorMessage("Gli anni di esperienza devono essere un numero valido.")
            return;
        }
    }

    // 5. Gestione foto profilo
    if (fotoModificata) {
        // Verifica formato, copia file, aggiorna path nel database
```

```
        loggedChef.setUrl_Propic(nuovoPath);
        changed = true;
    }

    // 6. Gestione password
    if (voglioCambiarePwd) {
        if (!loggedChef.getPassword().equals(currentPwd)) {
            boundary.showErrorMessage("Password attuale errata.");
            return;
        }
        if (!newPwd.equals(confirmPwd)) {
            boundary.showErrorMessage("Le nuove password non coincidono.");
            return;
        }
        loggedChef.setPassword(newPwd);
        changed = true;
    }

    // 7. Salvataggio tramite DAO
    if (!changed) {
        boundary.showInfoMessage("Nessuna modifica da salvare.");
        return;
    }
    try {
        chefDao.ModificaUtente(loggedChef);
        boundary.showSuccessMessage("Modifiche salvate con successo.");
    } catch (Exception e) {
        boundary.showErrorMessage("Errore durante il salvataggio: " + e.getMessage());
    }
}
```

Spiegazione: Il metodo `saveChanges` garantisce che solo i dati effettivamente modificati vengano validati e salvati, riducendo il rischio di errori e ottimizzando la sicurezza. La gestione separata di foto profilo e password assicura che ogni aspetto dell'identità chef sia aggiornato in modo sicuro e controllato. In caso di errori di validazione o salvataggio, l'utente riceve feedback immediato tramite dialog contestuali.

File e metodi principali

- `accountmanagementchef.fxml`: definizione della scena e dei controlli.
- `AccountManagementChefBoundary.java`: gestione della UI e degli eventi.
- `AccountManagementChefController.java`: logica di caricamento dati, validazione e salvataggio.
- `ChefDao.java`: accesso e modifica dei dati chef nel database.

In sintesi, la gestione account chef garantisce sicurezza, flessibilità e personalizzazione, permettendo allo chef di mantenere sempre aggiornati i propri dati e di gestire la propria identità professionale in modo semplice e sicuro.

5 Utils

Nel progetto UninaFoodLab, la componente **utils** raccoglie tutte le classi di utilità e supporto che facilitano la gestione di operazioni comuni, la validazione dei dati, la manipolazione delle immagini, la gestione delle scene e il feedback all'utente. Questi utility sono pensati per essere riutilizzabili, modulari e indipendenti dalla logica di business, contribuendo a mantenere il codice pulito e manutenibile.

5.1 Principali utility utilizzati

Di seguito vengono descritti i principali utility implementati e utilizzati nel progetto:

- `CardValidator`: gestisce la validazione delle carte di pagamento, controllando formato, tipo, scadenza e coerenza dei dati inseriti.
- `ImageClipUtils`: fornisce metodi per la manipolazione delle immagini profilo, come il ritaglio circolare e la gestione delle preview.
- `SceneSwitcher`: semplifica la navigazione tra le diverse scene JavaFX, gestendo il cambio pagina, il passaggio di parametri e la visualizzazione di dialog.
- `SuccessDialogUtils`: utility per la visualizzazione di dialog di conferma, successo o errore, con feedback visivo all'utente.
- `FrequenzaSessioniProvider`: fornisce in modo centralizzato le opzioni di frequenza delle sessioni di corso, recuperandole dal database e gestendo la cache interna.
- `UnifiedRecipeIngredientUI`: genera dinamicamente la UI per la gestione di ricette e ingredienti, offrendo modularità e notifiche automatiche al controller.
- `ErrorCaricamentoPropic`: gestisce le eccezioni relative al caricamento delle immagini profilo, fornendo messaggi di errore specifici e facilitando il debug.

Ognuna di queste classi è stata progettata per essere facilmente estendibile e integrabile nelle varie componenti dell'applicazione, favorendo la separazione delle responsabilità e la riusabilità del codice.

5.2 Vantaggi dell'utilizzo di utility

L'uso di utility nel progetto UninaFoodLab offre diversi vantaggi:

- **Modularità**: le utility sono indipendenti dalla logica di business, consentendo di concentrarsi su operazioni specifiche senza appesantire le classi principali.
- **Riutilizzabilità**: le stesse utility possono essere utilizzate in diverse parti dell'applicazione, riducendo la duplicazione del codice e migliorando la manutenibilità.
- **Chiarezza del codice**: separando le operazioni comuni in classi dedicate, il codice principale risulta più leggibile e comprensibile, facilitando la collaborazione tra sviluppatori.
- **Testabilità**: le utility possono essere testate in modo indipendente, garantendo che funzionino correttamente senza dipendere da altre parti del sistema.

Questa organizzazione contribuisce a creare un'applicazione robusta, scalabile e facile da mantenere, rispettando le best practice di programmazione e design patterns.

5.3 SceneSwitcher

La classe `SceneSwitcher` è una utility fondamentale per la gestione della navigazione tra le diverse scene dell'applicazione JavaFX. Il suo obiettivo è centralizzare e semplificare il cambio pagina, la gestione delle dimensioni delle finestre, il passaggio di parametri tra controller e la visualizzazione di dialog personalizzati.

Ruolo e vantaggi

- Permette di passare da una scena all'altra in modo sicuro e modulare, evitando duplicazione di codice e errori di gestione delle finestre.
- Gestisce automaticamente le dimensioni minime, massime e predefinite delle finestre in base al tipo di scena (login, registrazione, main app, transazioni, dialog).
- Supporta la visualizzazione di dialog modali (es. conferma logout, calendario lezioni) centrando la finestra rispetto al parent.
- Consente il passaggio di parametri e controller tra le scene, facilitando la comunicazione tra le diverse componenti.
- Gestisce le differenze tra sistemi operativi (Windows, Linux, Mac) per massimizzazione e stile delle finestre.

Funzionalità principali

- `switchScene`: metodo generico per cambiare scena in base al tipo di pagina (login, register, main, payment, ecc.).
- `switchToLogin`, `switchToRegister`, `switchToMainApp`, `switchToTransaction`: metodi specifici per gestire le principali scene dell'applicazione.
- `showDialogCentered`: visualizza dialog modali centrati rispetto alla finestra principale.
- `showCalendarDialog`, `showLogoutDialog`: gestiscono la visualizzazione di dialog specializzati (calendario lezioni, conferma logout) con passaggio di parametri e controller.
- Gestione automatica delle dimensioni e massimizzazione delle finestre in base al contesto e al sistema operativo.

Esempio di utilizzo

```
// Cambio scena verso la homepage chef
SceneSwitcher.switchScene(stage, "/fxml/homepagechef.fxml", "UninaFoodLab - Homepage Chef");

// Visualizzazione dialog di logout
LogoutDialogBoundary dialog = SceneSwitcher.showLogoutDialog(stage);
if (dialog.isConfirmed()) {
    SceneSwitcher.switchToLogin(stage, "/fxml/loginpage.fxml", "UninaFoodLab - Login");
}
```

Integrazione e modularità SceneSwitcher è utilizzata in tutte le boundary e controller dell'applicazione per garantire una navigazione coerente, sicura e facilmente estendibile. La sua modularità consente di aggiungere nuove scene o dialog senza modificare la logica delle pagine principali, favorendo la manutenibilità e la scalabilità del progetto.

5.4 SuccessDialogUtils

La classe `SuccessDialogUtils` è una utility dedicata alla visualizzazione di dialog di conferma, successo o annullamento, offrendo feedback visivo e immediato all'utente in seguito a operazioni importanti (salvataggio dati, pagamento, annullamento modifiche, ecc.).

Ruolo e vantaggi

- Centralizza la gestione dei dialog di successo, evitando duplicazione di codice e garantendo coerenza grafica e funzionale.
- Permette di mostrare messaggi personalizzati in base al contesto (es. pagamento completato, dati salvati, modifiche annullate).
- Supporta la visualizzazione di informazioni aggiuntive (es. nome corso appena acquistato) e la personalizzazione dei titoli e dei messaggi.
- Gestisce la posizione e lo stile del dialog, centrando la finestra rispetto al parent e utilizzando uno stile grafico dedicato (`successdialog.fxml`).
- Facilita l'integrazione con boundary e controller, permettendo di mostrare feedback all'utente con una sola chiamata.

Funzionalità principali

- `showPaymentSuccessDialog`: mostra un dialog di conferma pagamento, con nome corso e messaggio personalizzato.
- `showSaveSuccessDialog`: mostra un dialog di conferma salvataggio dati.
- `showCancelSuccessDialog`: mostra un dialog di conferma annullamento modifiche.
- `showGenericSuccessDialog`: permette di mostrare dialog di successo generici, personalizzando titolo e messaggio.
- Gestione automatica della posizione, stile e chiusura del dialog.

Esempio di utilizzo

```
// Dopo il salvataggio dei dati account
SuccessDialogUtils.showSaveSuccessDialog(stage);

// Dopo il pagamento di un corso
SuccessDialogUtils.showPaymentSuccessDialog(stage, "Corso di Sushi");

// Dopo l'annullamento delle modifiche
SuccessDialogUtils.showCancelSuccessDialog(stage);
```

Integrazione e modularità SuccessDialogUtils è utilizzata in boundary e controller per offrire feedback immediato e coerente all'utente, migliorando l'esperienza d'uso e la chiarezza delle operazioni. La modularità della classe consente di estendere facilmente i dialog per nuovi contesti o messaggi, mantenendo la coerenza grafica e funzionale in tutta l'applicazione.

5.5 CardValidator

La classe `CardValidator` è una utility dedicata alla validazione delle carte di pagamento inserite dagli utenti. Il suo scopo è garantire che i dati relativi alle carte siano corretti, coerenti e conformi ai criteri di sicurezza richiesti per le transazioni online.

Ruolo e vantaggi

- Centralizza la logica di validazione delle carte, evitando duplicazione di codice e possibili errori nei controller.
- Permette di identificare il tipo di carta (Visa, Mastercard) in base al numero inserito, migliorando l'esperienza utente e la sicurezza.
- Verifica la validità del formato e della scadenza della carta, impedendo l'inserimento di dati errati o scaduti.
- Facilita l'integrazione con boundary e controller, permettendo di validare i dati con una sola chiamata.

Funzionalità principali

- `getCardType`: identifica il tipo di carta (Visa, Mastercard, Unknown) in base al prefisso del numero.
- `isValidCardType`: verifica se il numero inserito corrisponde a un tipo di carta supportato.
- `isValidExpiryDate`: controlla che la data di scadenza sia nel formato corretto (MM/YY o MM/YYYY) e che la carta non sia scaduta.

Criteri di validazione

- **Tipo carta**: accetta solo Visa (prefisso 4) e Mastercard (prefisso 5 o 2221-2720).
- **Formato scadenza**: accetta MM/YY o MM/YYYY, verifica che il mese sia tra 1 e 12 e che la data non sia passata.
- **Sicurezza**: impedisce l'inserimento di carte scadute o con formato non valido.

Esempio di utilizzo

```
// Validazione tipo carta
if (!CardValidator.isValidCardType(cardNumber)) {
    showError("Tipo di carta non supportato.");
}

// Validazione scadenza
```

```
if (!CardValidator.isValidExpiryDate(expiry)) {
    showError("Data di scadenza non valida o carta scaduta.");
}
```

Integrazione e modularità CardValidator è utilizzata in tutte le boundary e controller che gestiscono l'inserimento di carte di pagamento, garantendo coerenza, sicurezza e semplicità di utilizzo. La modularità della classe consente di estendere facilmente i criteri di validazione per supportare nuovi tipi di carte o regole di business.

5.6 ImageClipUtils

La classe `ImageClipUtils` è una utility dedicata alla manipolazione delle immagini profilo all'interno dell'applicazione. Il suo scopo principale è applicare un ritaglio circolare alle immagini, migliorando l'estetica della UI e garantendo uniformità nella visualizzazione delle foto utente e chef.

Ruolo e vantaggi

- Centralizza la logica di ritaglio delle immagini, evitando duplicazione di codice e garantendo coerenza grafica.
- Permette di applicare un clip circolare centrato a qualsiasi `ImageView`, con raggio personalizzabile o calcolato automaticamente.
- Migliora l'esperienza utente, offrendo una visualizzazione moderna e professionale delle immagini profilo.
- Facilita l'integrazione con boundary e controller, permettendo di applicare il ritaglio con una sola chiamata.

Funzionalità principali

- `setCircularClip`: applica un clip circolare centrato all'`ImageView` passato, con raggio specificato o calcolato in base alle dimensioni dell'immagine.

Esempio di utilizzo

```
// Applica ritaglio circolare all'immagine profilo utente
ImageClipUtils.setCircularClip(userProfileImage, 40);

// Applica ritaglio automatico (metà lato minore)
ImageClipUtils.setCircularClip(profileImageLarge, 0);
```

Integrazione e modularità ImageClipUtils è utilizzata in tutte le boundary che gestiscono immagini profilo (utente, chef), garantendo coerenza grafica e semplicità di utilizzo. La modularità della classe consente di estendere facilmente le funzionalità per nuovi tipi di ritaglio o effetti grafici.

5.7 FrequenzaSessioniProvider

La classe `FrequenzaSessioniProvider` è una utility pensata per gestire in modo centralizzato la fornitura delle opzioni di frequenza delle sessioni di corso. Il suo scopo è rendere disponibili, in modo efficiente e riutilizzabile, tutte le possibili frequenze (es. settimanale, bisettimanale, mensile) utilizzate nella creazione e modifica dei corsi.

Ruolo e vantaggi

- Centralizza la logica di recupero delle frequenze, evitando duplicazione di codice nei controller e boundary.
- Recupera le opzioni dal database tramite la DAO dedicata (`BarraDiRicercaDao`), garantendo che siano sempre aggiornate e coerenti con i dati reali.
- Utilizza una cache interna per evitare chiamate ripetute al database e migliorare le performance.
- Facilita l'integrazione con le form di creazione e modifica corso, permettendo di popolare le `ComboBox` con una sola chiamata.

Funzionalità principali

- `getFrequenze`: metodo statico che restituisce la lista delle frequenze disponibili, recuperandole dal database solo al primo utilizzo.

Esempio di utilizzo

```
// Popola la ComboBox delle frequenze nella form di creazione corso
frequenzaComboBox.getItems().addAll(FrequenzaSessioniProvider.getFrequenze());
```

Integrazione e modularità FrequenzaSessioniProvider è utilizzata in tutte le boundary e controller che gestiscono la creazione e modifica dei corsi, garantendo coerenza, efficienza e semplicità di utilizzo. La modularità della classe consente di estendere facilmente la logica per supportare nuove tipologie di frequenza o fonti dati.

5.8 UnifiedRecipeIngredientUI

La classe `UnifiedRecipeIngredientUI` è una utility avanzata dedicata alla generazione dinamica della UI per la gestione di ricette e ingredienti all'interno delle form di creazione e modifica corso. Il suo scopo è offrire un'interfaccia modulare, riutilizzabile e facilmente estendibile per la visualizzazione, aggiunta, modifica e rimozione di ricette e ingredienti.

Ruolo e vantaggi

- Centralizza la logica di costruzione della UI per ricette e ingredienti, garantendo coerenza grafica e funzionale tra le diverse scene.
- Permette di gestire ricette e ingredienti in modo dinamico, con supporto a modalità ibride e personalizzazione delle unità di misura.
- Facilita l'integrazione con boundary e controller, notificando automaticamente le modifiche tramite callback dedicate.
- Migliora l'esperienza utente, offrendo una UI intuitiva, ordinata e facilmente estendibile.

Funzionalità principali

- `createUnifiedRecipeBox`: genera un container grafico (VBox) per una ricetta, con campi per nome, lista ingredienti, pulsanti di aggiunta/rimozione e gestione callback.
- `createUnifiedIngredientBox`: genera una riga grafica (HBox) per un ingrediente, con campi per nome, quantità, unità di misura e pulsante di rimozione.
- Supporto a unità di misura personalizzabili e modalità ibride (es. corsi con sessioni di tipo diverso).
- Notifica automatica delle modifiche al controller tramite callback (Runnable).

Esempio di utilizzo

```
// Genera la UI per una ricetta e la aggiunge al container principale
VBox recipeBox = UnifiedRecipeIngredientUI.createUnifiedRecipeBox(ricetta, recipesList, contair
container.getChildren().add(recipeBox);
```

Integrazione e modularità UnifiedRecipeIngredientUI è utilizzata in tutte le boundary che gestiscono la creazione e modifica di corsi, garantendo coerenza, efficienza e semplicità di utilizzo. La modularità della classe consente di estendere facilmente la UI per nuove funzionalità (es. gestione allergeni, filtri avanzati, validazioni aggiuntive) senza modificare la logica principale delle form.

5.9 ErrorCarimentoPropic

La classe `ErrorCaricamentoPropic` è una utility dedicata alla gestione delle eccezioni durante il caricamento delle immagini profilo (propic) degli utenti e chef. Estende `RuntimeException` e fornisce un messaggio di errore specifico quando la propic non viene trovata o caricata correttamente.

Ruolo e vantaggi

- Centralizza la gestione degli errori relativi al caricamento delle immagini profilo, rendendo il codice più chiaro e facilmente manutenibile.
- Permette di distinguere rapidamente i problemi di caricamento propic da altre eccezioni, facilitando il debug e la gestione dei casi limite.
- Migliora la robustezza dell'applicazione, consentendo di gestire in modo controllato la mancanza di immagini profilo e di fornire feedback appropriato all'utente.

Funzionalità principali

- Costruttore senza parametri che imposta il messaggio di errore a `"Propic non trovata"`.
- Può essere lanciata da boundary, controller o utility che gestiscono il caricamento delle immagini profilo.

Esempio di utilizzo

```
// Nel caricamento dell'immagine profilo utente
if (immaginePropic == null) {
    throw new ErrorCaricamentoPropic();
}
```

Integrazione e modularità ErrorCaricamentoPropic è utilizzata in tutte le componenti che gestiscono il caricamento delle immagini profilo, in particolare nelle boundary e nelle utility grafiche. La sua modularità consente di estendere facilmente la gestione degli errori per altri tipi di immagini o risorse, mantenendo la coerenza e la chiarezza nella gestione delle eccezioni.

6 DTO e DAO

I pattern **DTO** (Data Transfer Object) e **DAO** (Data Access Object) sono fondamentali per la separazione tra logica applicativa e gestione dei dati all'interno di UninaFoodLab.

DTO Il DTO è una struttura dati semplice, spesso una classe con soli attributi e metodi getter/setter, utilizzata per trasferire informazioni tra i vari layer dell'applicazione (ad esempio tra DAO, controller e boundary). I DTO non contengono logica di business, ma solo dati, facilitando il passaggio di informazioni in modo sicuro.

DAO Il DAO è una classe o interfaccia che incapsula tutte le operazioni di accesso e manipolazione dei dati su una fonte persistente (come un database). Permette di astrarre dati, offrendo metodi chiari per leggere, scrivere, aggiornare ed eliminare dati, senza esporre i dettagli tecnici al resto dell'applicazione. Questo favorisce la manutenibilità, la testabilità e la scalabilità del sistema.

L'utilizzo combinato di DAO e DTO consente di:

- Separare la logica di accesso ai dati dalla logica di presentazione e business.
- Ridurre la duplicazione del codice e favorire la riusabilità.
- Migliorare la sicurezza e la coerenza nella gestione delle informazioni.
- Facilitare la manutenzione e l'estensione dell'applicazione.

6.1 Elenco dei DTO e DAO implementati

Nel progetto UninaFoodLab sono stati implementati diversi DAO e DTO per gestire le principali entità e operazioni dell'applicazione. Di seguito una breve lista dei più rilevanti:

DTO

- `Utente`: rappresentazione dati utente
- `UtenteVisitatore`: dati utente visitatore
- `Chef`: rappresentazione dati chef
- `Corso`: dati corso, sessioni, iscrizioni
- `Sessioni`: dati delle sessioni di corso
- `SessioneOnline`: dati sessione online
- `SessioneInPresenza`: dati sessione in presenza
- `Pagamento`: dati pagamento e transazione

- `CartaDiCredito`: dati carta di credito
- `Ricetta`: dati ricetta
- `Ingredienti`: dati ingredienti
- `GraficoChef`: dati statistici chef

DAO

- `UtenteDao`: gestione dati utente (registrazione, login, aggiornamento profilo)
- `UtenteVisitatoreDao`: operazioni su utenti visitatori
- `ChefDao`: gestione dati chef e corsi associati
- `CorsoDao`: operazioni su corsi, iscrizioni, sessioni
- `SessioniDao`: gestione delle sessioni di corso
- `SessioneOnlineDao`: gestione delle sessioni online
- `SessioneInPresenzaDao`: gestione delle sessioni in presenza
- `PagamentoDao`: gestione pagamenti e transazioni
- `CartaDiCreditoDao`: gestione delle carte di credito
- `RicettaDao`: gestione ricette
- `IngredientiDao`: gestione ingredienti
- `GraficoChefDao`: gestione dati statistici chef
- `BarraDiRicercaDao`: ricerca e filtri avanzati su corsi e chef

Questi DAO e DTO sono utilizzati per garantire una gestione strutturata, sicura e modulare dei dati tra i vari layer dell'applicazione.

Nei paragrafi successivi verranno descritti i principali DAO e DTO implementati e utilizzati nel progetto, con esempi pratici e dettagli sulle loro responsabilità.

6.2 DTO

6.2.1 Utente

Il DTO `Utente` rappresenta il modello base per tutti gli utenti dell'applicazione UninaFoodLab. Definisce gli attributi comuni come nome, cognome, email, password, data di nascita, URL della propic e la lista dei corsi associati. Essendo una classe astratta, non viene mai istanziata direttamente, ma fornisce la struttura e i metodi fondamentali per la gestione dei dati utente, garantendo coerenza e riusabilità tra le diverse tipologie di utenti.

Principali attributi

- `nome`, `cognome`, `email`, `password`, `dataDiNascita`: dati anagrafici e di accesso.
- `Url_Propic`: URL dell'immagine profilo.
- `corsi`: lista dei corsi associati all'utente.

Ruolo La classe `Utente` funge da base per la gerarchia dei DTO utente, permettendo di estendere facilmente le funzionalità per utenti specifici come visitatori e chef.

6.2.2 UtenteVisitatore

Il DTO `UtenteVisitatore` estende `Utente` e rappresenta l'utente generico che può iscriversi ai corsi, gestire le proprie carte di credito e interagire con la piattaforma. Oltre agli attributi ereditati, include una lista di carte di credito, l'identificativo utente e il riferimento al proprio DAO per le operazioni di persistenza. La presenza della variabile statica `loggedUser` permette di gestire la sessione utente corrente.

Principali attributi

- `carte`: lista delle carte di credito associate.
- `id_UtenteVisitatore`: identificativo univoco dell'utente.
- `utenteVisitatoreDao`: riferimento al DAO per operazioni su database.

Ruolo `UtenteVisitatore` è il DTO utilizzato per la maggior parte delle operazioni utente, come iscrizione ai corsi, gestione pagamenti e aggiornamento profilo. Eredita tutti i metodi e attributi di `Utente`, aggiungendo funzionalità specifiche per la gestione delle carte e della persistenza.

6.2.3 Chef

Il DTO `Chef` estende `Utente` e rappresenta l'utente chef, ovvero colui che può creare e gestire corsi, visualizzare statistiche e modificare la propria descrizione. Oltre agli attributi comuni, include anni di esperienza, descrizione, identificativo chef, riferimento al proprio DAO e al grafico delle statistiche. Anche qui è presente la variabile statica `loggedUser` per la sessione chef corrente.

Principali attributi

- `anniDiEsperienza`: esperienza professionale dello chef.
- `id_Chef`: identificativo univoco dello chef.
- `Descrizione`: descrizione personale e professionale.
- `chefDao`: riferimento al DAO per operazioni su database.
- `grafico1`: dati statistici e grafici associati allo chef.

Ruolo `Chef` è il DTO dedicato agli utenti che gestiscono corsi e ricette sulla piattaforma. Eredita la struttura di `Utente`, aggiungendo attributi e metodi specifici per la gestione avanzata dei dati chef e delle statistiche.

Relazione tra i DTO Le classi `Utente`, `UtenteVisitatore` e `Chef` sono strettamente collegate: `Utente` fornisce la base comune, mentre `UtenteVisitatore` e `Chef` la estendono per coprire le esigenze delle due principali tipologie di utenti della piattaforma. Questa struttura favorisce la modularità, la riusabilità e la coerenza nella gestione dei dati utente, semplificando l'integrazione con i DAO e la logica applicativa.

6.2.4 Corso

Il DTO `Corso` rappresenta il modello dati per i corsi offerti sulla piattaforma UninaFoodLab. Questa classe incapsula tutte le informazioni fondamentali relative a un corso, come nome, descrizione, periodo di svolgimento, frequenza delle sessioni, numero massimo di partecipanti, prezzo, immagine di copertina, identificativo e dati del chef associato. Inoltre, gestisce la lista delle sessioni del corso e le tipologie di cucina trattate.

Principali attributi

- `nome`, `descrizione`: informazioni generali sul corso.
- `dataInizio`, `dataFine`: periodo di svolgimento.
- `FrequenzaDelleSessioni`: frequenza con cui si tengono le lezioni (es. settimanale, mensile).
- `MaxPersone`: numero massimo di partecipanti.
- `Prezzo`: costo di iscrizione al corso.
- `Url_Propic`: URL dell'immagine di copertina del corso.
- `id_Corso`: identificativo univoco del corso.
- `sessioni`: lista delle sessioni che compongono il corso.
- `TipiDiCucina`: lista delle tipologie di cucina trattate.
- `chefNome`, `chefCognome`, `chefEsperienza`: dati sintetici del chef associato.

Ruolo Il DTO `Corso` è centrale per la gestione e visualizzazione dei corsi all'interno della piattaforma. Viene utilizzato per trasferire i dati tra DAO, controller e boundary, garantendo che tutte le informazioni rilevanti siano disponibili per la presentazione, la modifica e la persistenza.

Relazione con altri DTO La classe `Corso` è strettamente collegata ai DTO `Sessione` (per la gestione delle lezioni), `Chef` (per i dati del docente) e `UtenteVisitatore` (per la gestione delle iscrizioni). Questa struttura favorisce la modularità e la coerenza nella gestione dei dati, permettendo di integrare facilmente nuove funzionalità come filtri, statistiche e personalizzazioni.

Esempio di utilizzo

```
Corso corso = new Corso("Corso Sushi", "Impara l'arte del sushi", LocalDate.of(2025, 9, 1), Loc
corso.setChefNome("Mario");
corso.setChefCognome("Rossi");
corso.setChefEsperienza(10);
corso.addTipoDiCucina("Giapponese");
corso.setSessioni(listaSessioni);
```

Questa organizzazione consente di gestire in modo strutturato e flessibile tutte le informazioni relative ai corsi, facilitando l'integrazione con la logica applicativa e la presentazione grafica.

6.2.5 Sessione

Il DTO `Sessione` rappresenta il modello astratto per una singola sessione di corso, sia essa online che in presenza. Definisce gli attributi comuni come data, orario, durata, giorno, identificativi e il riferimento al chef responsabile. Essendo una classe astratta, non viene mai istanziata direttamente, ma fornisce la struttura di base per le sessioni specializzate.

Principali attributi

- `giorno`, `data`, `orario`, `durata`: informazioni temporali della sessione.
- `id_Sessione`, `id_Corso`: identificativi della sessione e del corso associato.
- `chef`: riferimento al docente della sessione.

Ruolo La classe `Sessione` funge da base per la gerarchia delle sessioni, permettendo di estendere facilmente le funzionalità per sessioni online e in presenza, garantendo coerenza e riusabilità.

6.2.6 SessioneOnline

Il DTO `SessioneOnline` estende `Sessione` e rappresenta una sessione di corso svolta tramite piattaforme digitali. Oltre agli attributi ereditati, include informazioni specifiche come l'applicazione utilizzata (es. Zoom, Teams), il codice chiamata e una descrizione della sessione. Questo consente di gestire in modo strutturato le lezioni a distanza, facilitando la comunicazione e la partecipazione degli utenti.

Principali attributi

- `Applicazione`: piattaforma digitale utilizzata.
- `Codicechiamata`: codice per accedere alla sessione online.
- `Descrizione`: dettagli aggiuntivi sulla sessione.

Ruolo `SessioneOnline` è utilizzata per tutte le lezioni che si svolgono da remoto, integrando le funzionalità di `Sessione` con attributi specifici per la didattica online.

6.2.7 SessioniInPresenza

Il DTO `SessioniInPresenza` estende `Sessione` e rappresenta una sessione di corso svolta fisicamente in aula o laboratorio. Oltre agli attributi comuni, include informazioni sulla sede (città, via, cap), attrezzatura necessaria, descrizione, lista dei partecipanti e ricette trattate. Questo consente di gestire in modo dettagliato le lezioni in presenza, facilitando l'organizzazione logistica e la personalizzazione dell'esperienza.

Principali attributi

- `citta`, `via`, `cap`: dati sulla sede della sessione.
- `attrezzatura`: strumenti necessari per la lezione.
- `descrizione`: dettagli aggiuntivi sulla sessione.

- `corsoListPartecipanti`: lista dei partecipanti alla sessione.
- `ricette`: ricette trattate durante la sessione.

Ruolo `SessioniInPresenza` è utilizzata per tutte le lezioni che si svolgono fisicamente, integrando le funzionalità di `Sessione` con attributi specifici per la didattica in aula e la gestione dei partecipanti.

Relazione tra i DTO Le classi `Sessione`, `SessioneOnline` e `SessioniInPresenza` sono strettamente collegate: `Sessione` fornisce la base comune, mentre le altre la estendono per coprire le esigenze delle due principali modalità di svolgimento dei corsi. Questa struttura favorisce la modularità, la riusabilità e la coerenza nella gestione delle lezioni, semplificando l'integrazione con i DTO `Corso`, `Chef` e `UtenteVisitatore`.

6.2.8 Ricetta

Il DTO `Ricetta` rappresenta il modello dati per una ricetta culinaria all'interno della piattaforma UninaFoodLab. Questa classe incapsula il nome della ricetta, la lista degli ingredienti che la compongono e l'identificativo univoco. Permette di gestire in modo strutturato la creazione, modifica e visualizzazione delle ricette associate ai corsi e alle sessioni.

Principali attributi

- `nome`: nome della ricetta.
- `ingredientiRicetta`: lista degli ingredienti che compongono la ricetta.
- `id_Ricetta`: identificativo univoco della ricetta.

Ruolo Il DTO `Ricetta` è centrale per la gestione delle ricette nei corsi e nelle sessioni, consentendo di aggiungere, modificare o rimuovere ingredienti in modo dinamico. Facilita l'integrazione con la logica applicativa e la presentazione grafica, permettendo di visualizzare le ricette complete e dettagliate.

Relazione con altri DTO La classe `Ricetta` è strettamente collegata al DTO `Ingredienti`, che rappresenta i singoli ingredienti della ricetta. Questa struttura favorisce la modularità e la coerenza nella gestione dei dati culinari.

6.2.9 Ingredienti

Il DTO `Ingredienti` rappresenta il modello dati per un singolo ingrediente utilizzato nelle ricette della piattaforma. Definisce il nome, la quantità, l'unità di misura, la quantità totale e l'identificativo dell'ingrediente. Permette di gestire in modo dettagliato le informazioni nutrizionali e operative degli ingredienti, facilitando la creazione e la modifica delle ricette.

Principali attributi

- `nome`: nome dell'ingrediente.
- `quantita`: quantità utilizzata nella ricetta.
- `unitaMisura`: unità di misura della quantità.
- `quantitaTotale`: quantità totale disponibile o richiesta.
- `id_Ingrediente`: identificativo univoco dell'ingrediente.

Ruolo Il DTO `Ingredienti` è utilizzato per rappresentare in modo preciso e flessibile i componenti delle ricette, consentendo di gestire le quantità, le unità di misura e le informazioni aggiuntive. Facilita la personalizzazione delle ricette e la gestione delle scorte.

Relazione con altri DTO La classe `Ingredienti` è strettamente collegata al DTO `Ricetta`, di cui rappresenta i singoli componenti. Questa organizzazione favorisce la modularità e la coerenza nella gestione dei dati culinari e delle operazioni di cucina.

6.2.10 GraficoChef

Il DTO `GraficoChef` rappresenta il modello dati per le statistiche e i grafici associati all'attività di uno chef sulla piattaforma UninaFoodLab. Questa classe incapsula informazioni come il numero massimo e minimo di partecipanti ai corsi, la media, il numero totale di corsi, il numero di sessioni in presenza e il numero di sessioni telematiche. Permette di visualizzare e analizzare le performance dello chef in modo strutturato e dettagliato.

Principali attributi

- `numeroMassimo`: massimo numero di partecipanti ai corsi.
- `NumeroMinimo`: minimo numero di partecipanti ai corsi.
- `Media`: media dei partecipanti o delle valutazioni.
- `NumeriCorsi`: numero totale di corsi gestiti dallo chef.
- `numeroSessioniInPresenza`: numero di sessioni svolte in presenza.
- `numerosessionitelematiche`: numero di sessioni svolte online.

Ruolo Il DTO `GraficoChef` è utilizzato per raccogliere e presentare dati statistici relativi all'attività dello chef, facilitando la visualizzazione delle performance e l'analisi dei risultati. È integrato con il DTO `Chef` per offrire una panoramica completa delle attività svolte.

Relazione con altri DTO La classe `GraficoChef` è strettamente collegata al DTO `Chef`, di cui rappresenta le statistiche e i dati aggregati. Questa organizzazione favorisce la modularità e la coerenza nella gestione delle informazioni di performance.

6.2.11 CartaDiCredito

Il DTO `CartaDiCredito` rappresenta il modello dati per una carta di credito associata agli utenti della piattaforma. Definisce l'intestatario, la data di scadenza, le ultime quattro cifre, il circuito e l'identificativo della carta. Permette di gestire in modo sicuro e strutturato le informazioni di pagamento, facilitando le operazioni di acquisto e iscrizione ai corsi.

Principali attributi

- `Intestatario`: nome del titolare della carta.
- `DataScadenza`: data di scadenza della carta.
- `UltimeQuattroCifre`: ultime quattro cifre della carta.
- `Circuito`: circuito di pagamento (es. Visa, Mastercard).
- `IdCarta`: identificativo univoco della carta.

Ruolo Il DTO `CartaDiCredito` è utilizzato per gestire le informazioni di pagamento degli utenti, garantendo sicurezza e flessibilità nelle transazioni. È integrato con il DTO `UtenteVisitatore` per la gestione delle carte associate a ciascun utente.

Relazione con altri DTO La classe `CartaDiCredito` è strettamente collegata al DTO `UtenteVisitatore`, di cui rappresenta i metodi di pagamento. Questa organizzazione favorisce la modularità e la coerenza nella gestione delle informazioni finanziarie.

6.3 DAO

6.3.1 UtenteDao

La classe `UtenteDao` è una classe astratta che definisce le operazioni fondamentali per la gestione degli utenti sulla piattaforma UninaFoodLab. Fornisce metodi per il login, la determinazione del tipo di account, il recupero dei corsi filtrati per categoria e frequenza, e le operazioni di base su profilo e corsi. Le query SQL utilizzate permettono di verificare l'esistenza dell'utente, distinguere tra chef e partecipante, e recuperare i dati dei corsi con join tra le tabelle correlate.

Principali metodi e query

- `LoginUtente`: verifica l'esistenza dell'utente tramite una query che controlla sia la tabella Partecipante che Chef.
- `TipoDiAccount`: determina se l'utente è chef (`c`), partecipante (`v`) o non esistente (`n`) tramite query dedicate.
- `recuperaCorsi`: recupera i corsi filtrati per categoria e frequenza, utilizzando join tra Corso, Chef, TipoDiCucina e altre tabelle correlate.
- Metodi astratti: `caricaPropic`, `AssegnaCorso`, `recuperaDatiUtente`, `RegistrazioneUtente`, `ModificaUtente`, `RecuperaCorsi`.

Ruolo `UtenteDao` funge da base per i DAO specializzati (`UtenteVisitatoreDao`, `ChefDao`), garantendo coerenza e riusabilità nella gestione delle operazioni comuni sugli utenti.

Esempi UtenteDao

```
// Metodo LoginUtente
public boolean LoginUtente(Utente utente) {
    String query = "SELECT EXISTS (SELECT 1 FROM Partecipante WHERE Email = ? AND Password = ?)
    // ...
    ps = conn.prepareStatement(query);
    ps.setString(1, utente.getEmail());
    ps.setString(2, utente.getPassword());
    ps.setString(3, utente.getEmail());
    ps.setString(4, utente.getPassword());
    rs = ps.executeQuery();
    if (rs.next()) {
        boolean esiste = rs.getBoolean("Esistenza");
        return esiste;
    }
    // ...
    return false;
}

// Metodo TipoDiAccount
public String TipoDiAccount(Utente utente) {
    String queryChef = "SELECT 1 FROM Chef WHERE Email = ? AND Password = ?";
    String queryPartecipante = "SELECT 1 FROM Partecipante WHERE Email = ? AND Password = ?";
    // ...
    ps = conn.prepareStatement(queryChef);
    ps.setString(1, utente.getEmail());
    ps.setString(2, utente.getPassword());
    rs = ps.executeQuery();
    if (rs.next()) {
        return "c";
    }
    // ...
    ps = conn.prepareStatement(queryPartecipante);
    ps.setString(1, utente.getEmail());
    ps.setString(2, utente.getPassword());
    rs = ps.executeQuery();
    if (rs.next()) {
        return "v";
    }
    // ...
    return "n";
}
```

6.3.2 UtenteVisitatoreDao

La classe `UtenteVisitatoreDao` estende `UtenteDao` e implementa le operazioni specifiche per la gestione degli utenti partecipanti. Permette la registrazione, il recupero e la modifica dei dati utente, l'assegnazione dei corsi, la gestione delle carte di credito e la partecipazione alle sessioni in presenza. Le query SQL sono ottimizzate per garantire la coerenza dei dati e la sicurezza delle operazioni.

Principali metodi e query

- `RegistrazioneUtente`: inserisce un nuovo partecipante nella tabella Partecipante.
- `recuperaDatiUtente`: recupera i dati utente tramite una query sulla tabella Partecipante.
- `AssegnaCorso`: registra il pagamento e l'iscrizione al corso tramite la tabella RICHIESTAPAGAMENTO.
- `haGiaConfermatoPresenza`: verifica la conferma di presenza tramite la tabella ADESIONE_SESSIONEPRES
- `partecipaAllaSessioneDalVivo`: registra la partecipazione a una sessione in presenza.
- `isUtenteIscrittoAlCorso`: verifica l'iscrizione tramite la tabella RICHIESTAPAGAMENTO.
- `ModificaUtente`: aggiorna i dati utente tramite una query di update su Partecipante.
- `RecuperaCorsiNonIscritto`: recupera i corsi disponibili tramite join e subquery.
- `RecuperaCorsi`: recupera i corsi a cui l'utente è iscritto tramite join tra RICHIESTAPAGAMENTO e CORSO.
- `EliminaCarta`, `aggiungiCartaAPossiede`: gestione delle carte di credito associate all'utente.
- `caricaPropic`: recupera l'immagine profilo dalla tabella Partecipante.

Ruolo `UtenteVisitatoreDao` gestisce tutte le operazioni di persistenza e aggiornamento per gli utenti partecipanti, integrando la logica di business con la sicurezza e la coerenza dei dati.

Esempi UtenteVisitatoreDao

```
// Metodo RegistrazioneUtente
public void RegistrazioneUtente(Utente utenteVisitatore) {
    String query = "INSERT INTO Partecipante (Nome, Cognome, Email, Password,  DataDiNascita) V
    // ...
    ps = conn.prepareStatement(query);
    java.sql.Date sqlData = java.sql.Date.valueOf(utenteVisitatore.getDataDiNascita());
    ps.setString(1, ((UtenteVisitatore) utenteVisitatore).getNome());
    ps.setString(2, ((UtenteVisitatore) utenteVisitatore).getCognome());
    ps.setString(3, ((UtenteVisitatore) utenteVisitatore).getEmail());
    ps.setString(4, ((UtenteVisitatore) utenteVisitatore).getPassword());
    ps.setDate(5, sqlData);
    ps.execute();
    // ...
}
```

```
// Metodo AssegnaCorso
public void AssegnaCorso(Corso corso, Utente utente1) {
    String query = "INSERT INTO RICHIESTAPAGAMENTO (idPartecipante, idCorso, DataRichiesta, Imp
    LocalDate DataRichiesta = LocalDate.now();
    // ...
    ps = conn.prepareStatement(query);
    java.sql.Date sqlData = java.sql.Date.valueOf(DataRichiesta);
    int idUtente = ((UtenteVisitatore) utente1).getId\_UtenteVisitatore();
    int idCorso = corso.getId\_Corso();
    ps.setInt(1, idUtente);
    ps.setInt(2, idCorso);
    ps.setDate(3, sqlData);
    ps.setDouble(4, corso.getPrezzo());
    ps.execute();
    // ...
}
```

6.3.3 ChefDao

La classe `ChefDao` estende `UtenteDao` e implementa le operazioni specifiche per la gestione degli utenti chef. Permette la registrazione, il recupero e la modifica dei dati chef, l'assegnazione e l'eliminazione dei corsi, e il recupero dei corsi gestiti. Le query SQL sono ottimizzate per gestire le relazioni tra chef e corsi, e per garantire la corretta associazione dei dati.

Principali metodi e query

- `RegistrazioneUtente`: inserisce un nuovo chef nella tabella Chef.
- `recuperaDatiUtente`: recupera i dati chef tramite una query sulla tabella Chef.
- `AssegnaCorso`: assegna un corso allo chef tramite update sulla tabella Corso.
- `RecuperaCorsi`: recupera i corsi gestiti dallo chef tramite join tra Corso e Chef.
- `eliminaCorso`: elimina un corso associato allo chef tramite query di delete.
- `ModificaUtente`: aggiorna i dati chef tramite una query di update su Chef.
- `caricaPropic`: recupera l'immagine profilo dalla tabella Chef.

Ruolo `ChefDao` gestisce tutte le operazioni di persistenza e aggiornamento per gli utenti chef, integrando la logica di business con la sicurezza e la coerenza dei dati, e facilitando la gestione avanzata dei corsi e delle statistiche.

Esempi ChefDao

```
// Metodo RegistrazioneUtente
public void RegistrazioneUtente(Utente chef1) {
    String query = "INSERT INTO Chef (Nome, Cognome, Email, Password, DataDiNascita, AnniDiEspe
    // ...
```

```
    ps = conn.prepareStatement(query);
    java.sql.Date sqlData = java.sql.Date.valueOf(chef1.getDataDiNascita());
    ps.setString(1, chef1.getNome());
    ps.setString(2, chef1.getCognome());
    ps.setString(3, chef1.getEmail());
    ps.setString(4, chef1.getPassword());
    ps.setDate(5, sqlData);
    ps.setInt(6, ((Chef)chef1).getAnniDiEsperienza());
    ps.execute();
    // ...
}

// Metodo AssegnaCorso
public void AssegnaCorso(Corso corso, Utente chef1) {
    String query = "UPDATE Corso SET IdChef = ? WHERE IdCorso = ?";
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, ((Chef) chef1).getId\_Chef());
    ps.setInt(2, corso.getId\_Corso());
    ps.executeUpdate();
    // ...
}
```

6.3.4 CorsoDao

La classe `CorsoDao` gestisce tutte le operazioni di persistenza relative ai corsi della piattaforma UninaFoodLab. Permette la creazione, il recupero, l'aggiornamento e la gestione delle relazioni tra corsi, chef, tipologie di cucina e sessioni (sia in presenza che online). Le query SQL sono ottimizzate per gestire le relazioni tra le varie entità e per garantire la coerenza dei dati.

Principali metodi e query

- `memorizzaCorsoERicavaId`: inserisce un nuovo corso nella tabella Corso e ne recupera l'id generato.
- `getCorsoById`: recupera i dati di un corso tramite una query sulla tabella Corso.
- `recuperaCorsi`: recupera tutti i corsi, includendo i dati dello chef associato tramite join tra Corso e Chef.
- `aggiornaCorso`: aggiorna i dati di un corso tramite una query di update.
- `recuperaTipoCucinaCorsi`: recupera le tipologie di cucina associate a un corso tramite join tra TIPODICUCINA e TIPODICUCINA_CORSO.
- `recuperoSessioniPerCorso`: recupera tutte le sessioni (in presenza e online) associate a un corso.
- `recuperoSessioniCorsoOnline`: recupera le sessioni online associate a un corso.
- `recuperoSessionCorso`: recupera le sessioni in presenza associate a un corso.

Ruolo `CorsoDao` centralizza la logica di accesso e manipolazione dei dati relativi ai corsi, facilitando la gestione delle relazioni con chef, sessioni e tipologie di cucina. Garantisce la coerenza e l'integrità dei dati, integrando la logica di business con le operazioni di persistenza.

Esempi CorsoDao

```
// Metodo memorizzaCorsoERicavaId
public void memorizzaCorsoERicavaId(Corso corso) {
    String query = "INSERT INTO Corso (Nome, Descrizione, DataInizio, DataFine, FrequenzaDelleS
    // ...
    ps = conn.prepareStatement(query, PreparedStatement.RETURN\_GENERATED\_KEYS);
    ps.setString(1, corso.getNome());
    ps.setString(2, corso.getDescrizione());
    ps.setDate(3, java.sql.Date.valueOf(corso.getDataInizio()));
    ps.setDate(4, java.sql.Date.valueOf(corso.getDataFine()));
    ps.setString(5, corso.getFrequenzaDelleSessioni());
    ps.setInt(6, corso.getMaxPersone());
    ps.setFloat(7, corso.getPrezzo());
    ps.setString(8, corso.getUrl\_Propic());
    ps.executeUpdate();
    // ...
    ResultSet generatedKeys = ps.getGeneratedKeys();
    if (generatedKeys.next()) {
        int id = generatedKeys.getInt(1);
        corso.setId\_Corso(id);
    }
    // ...
}

// Metodo recuperaCorsi
public ArrayList<Corso> recuperaCorsi() {
    String query = "SELECT c.*, ch.Nome AS chef\_nome, ch.Cognome AS chef\_cognome, ch.AnniDiEs
    // ...
    ps = conn.prepareStatement(query);
    rs = ps.executeQuery();
    while (rs.next()) {
        Corso corso = new Corso(
            rs.getString("Nome"),
            rs.getString("Descrizione"),
            rs.getDate("DataInizio").toLocalDate(),
            rs.getDate("DataFine").toLocalDate(),
            rs.getString("FrequenzaDelleSessioni"),
            rs.getInt("MaxPersone"),
            rs.getFloat("Prezzo"),
            rs.getString("Propic")
        );
        corso.setId\_Corso(rs.getInt("idCorso"));
        corso.setChefNome(rs.getString("chef\_nome"));
```

```
        corso.setChefCognome(rs.getString("chef\_cognome"));
        corso.setChefEsperienza(rs.getInt("chef\_esperienza"));
        // ...
    }
    // ...
}

// Metodo recuperaTipoCucinaCorsi
public void recuperaTipoCucinaCorsi(Corso corso) {
    String query = "SELECT T.Nome FROM TIPODICUCINA\_CORSO TC NATURAL JOIN TIPODICUCINA T WHERE
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, corso.getId\_Corso());
    rs = ps.executeQuery();
    while (rs.next()) {
        corso.addTipoDiCucina(rs.getString("Nome"));
    }
    // ...
}

// Metodo recuperoSessioniCorsoOnline
public ArrayList<SessioneOnline> recuperoSessioniCorsoOnline(Corso corso) {
    String query = "SELECT * FROM SESSIONE\_TELEMATICA WHERE idcorso = ?";
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, corso.getId\_Corso());
    rs = ps.executeQuery();
    while (rs.next()) {
        SessioneOnline sessione = new SessioneOnline(
            rs.getString("giorno"),
            rs.getDate("data").toLocalDate(),
            rs.getTime("orario").toLocalTime(),
            rs.getTime("durata").toLocalTime(),
            rs.getString("Applicazione"),
            rs.getString("Codicechiamata"),
            rs.getString("Descrizione"),
            rs.getInt("idsessionetelematica")
        );
        // ...
    }
    // ...
}
```

6.3.5 SessioniDao

La classe astratta `SessioniDao` definisce le operazioni comuni per la gestione delle sessioni (sia in presenza che online) sulla piattaforma UninaFoodLab. Fornisce metodi per il controllo delle sovrapposizioni di sessioni per uno chef e dichiara il metodo astratto per la memorizzazione di una

sessione. Viene estesa dalle classi specializzate `SessioneOnlineDao` e `SessioneInPresenzaDao`.

Principali metodi e query

- `ControllosSessioniAttiveLoStessoPeriodo`: verifica se uno chef ha già una sessione attiva nello stesso periodo tramite una query sulle tabelle SESSIONE_PRESENZA_CHEF e SESSIONE.
- `MemorizzaSessione`: metodo astratto per la memorizzazione di una sessione, implementato dalle sottoclassi.

Ruolo `SessioniDao` centralizza la logica di controllo e gestione delle sessioni, garantendo coerenza e riusabilità per le operazioni comuni tra sessioni in presenza e online.

Esempi SessioniDao

```
// Metodo ControllosSessioniAttiveLoStessoPeriodo
public boolean ControllosSessioniAttiveLoStessoPeriodo(Sessione sessione, Chef chef) {
    String query = "Select count(*) as NumSessioni from SESSIONE\_PRESENZA\_CHEF natural join S
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, chef.getId\_Chef());
    ps.setString(2, sessione.getGiorno());
    ps.setDate(3, java.sql.Date.valueOf(sessione.getData()));
    ps.setTime(4, java.sql.Time.valueOf(sessione.getOrario()));
    ps.setTime(5, java.sql.Time.valueOf(sessione.getDurata()));
    rs = ps.executeQuery();
    return rs.next();
    // ...
}
```

6.3.6 SessioneOnlineDao

La classe `SessioneOnlineDao` estende `SessioniDao` e implementa le operazioni specifiche per la gestione delle sessioni telematiche (online). Permette la creazione, l'aggiornamento e il recupero delle sessioni online associate ai corsi, gestendo i dati relativi all'applicazione, codice chiamata, orario, durata, giorno e descrizione.

Principali metodi e query

- `MemorizzaSessione`: inserisce una nuova sessione telematica nella tabella SESSIONE_TELEMATICA.
- `aggiornaSessione`: aggiorna i dati di una sessione telematica esistente.
- `getSessioniByCorso`: recupera tutte le sessioni telematiche associate a un corso tramite query sulla tabella SESSIONE_TELEMATICA.

Ruolo `SessioneOnlineDao` gestisce tutte le operazioni di persistenza e aggiornamento per le sessioni online, integrando la logica di business con la sicurezza e la coerenza dei dati.

Esempi SessioneOnlineDao

```
// Metodo MemorizzaSessione
public void MemorizzaSessione(Sessione sessione) {
    String query = "INSERT INTO SESSIONE\_TELEMATICA (Applicazione, CodiceChiamata, Data, Orari
    // ...
    ps = conn.prepareStatement(query);
    ps.setString(1, ((SessioneOnline) sessione).getApplicazione());
    ps.setString(2, ((SessioneOnline) sessione).getCodicechiamata());
    ps.setDate(3, java.sql.Date.valueOf(sessione.getData()));
    ps.setTime(4, java.sql.Time.valueOf(sessione.getOrario()));
    // Calcola la durata in ore intere (1-8) e passa come interval
    LocalTime durata = sessione.getDurata();
    int durataOre = Math.max(1, Math.min(8, durata.getHour()));
    PGInterval interval = new PGInterval(0, 0, 0, durataOre, 0, 0);
    ps.setObject(5, interval);
    ps.setString(6, sessione.getGiorno());
    String descrizione = ((SessioneOnline) sessione).getDescrizione();
    ps.setString(7, descrizione != null ? descrizione : "");
    ps.setInt(8, ((SessioneOnline) sessione).getId\_Corso());
    ps.setInt(9, sessione.getChef().getId\_Chef());
    ps.executeUpdate();
    // ...
}

// Metodo getSessioniByCorso
public static ArrayList<SessioneOnline> getSessioniByCorso(int idCorso) {
    String query = "SELECT * FROM sessione\_telematica WHERE idcorso = ? ORDER BY data, orario'
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, idCorso);
    rs = ps.executeQuery();
    while (rs.next()) {
        SessioneOnline sessione = new SessioneOnline();
        sessione.setId\_Sessione(rs.getInt("idsessionetelematica"));
        // ...
    }
    // ...
}
```

6.3.7 SessioneInPresenzaDao

La classe `SessioneInPresenzaDao` estende `SessioniDao` e implementa le operazioni specifiche per la gestione delle sessioni in presenza. Permette la creazione, l'aggiornamento e il recupero delle sessioni in presenza associate ai corsi, gestendo i dati relativi a luogo, orario, durata, ricette associate e partecipanti.

Principali metodi e query

- `MemorizzaSessione`: inserisce una nuova sessione in presenza nella tabella SESSIONE_PRESENZA.
- `aggiornaSessione`: aggiorna i dati di una sessione in presenza esistente.
- `getSessioniByCorso`: recupera tutte le sessioni in presenza associate a un corso tramite query sulla tabella SESSIONE_PRESENZA.
- `associaRicettaASessione`: associa una ricetta a una sessione in presenza.
- `recuperaRicetteSessione`: recupera le ricette associate a una sessione in presenza tramite join tra RICETTA e SESSIONE_PRESENZA_RICETTA.
- `recuperaPartecipantiSessione`: recupera i partecipanti di una sessione in presenza tramite join tra PARTECIPANTE e ADESIONE_SESSIONEPRESENZA.

Ruolo `SessioneInPresenzaDao` gestisce tutte le operazioni di persistenza e aggiornamento per le sessioni in presenza, integrando la logica di business con la sicurezza e la coerenza dei dati, e facilitando la gestione delle ricette e dei partecipanti associati.

Esempi SessioneInPresenzaDao

```
// Metodo MemorizzaSessione
public void MemorizzaSessione(Sessione sessione) {
    String query = "INSERT INTO SESSIONE\_PRESENZA (giorno, Data, Orario, Durata, citta, via, c
    // ...
    ps = conn.prepareStatement(query, PreparedStatement.RETURN\_GENERATED\_KEYS);
    ps.setString(1, sessione.getGiorno());
    ps.setDate(2, java.sql.Date.valueOf(sessione.getData()));
    ps.setTime(3, java.sql.Time.valueOf(sessione.getOrario()));
    LocalTime durata = sessione.getDurata();
    int durataOre = Math.max(1, Math.min(8, durata.getHour()));
    PGInterval interval = new PGInterval(0, 0, 0, durataOre, 0, 0);
    ps.setObject(4, interval);
    ps.setString(5, ((SessioniInPresenza) sessione).getCitta());
    ps.setString(6, ((SessioniInPresenza) sessione).getVia());
    ps.setString(7, ((SessioniInPresenza) sessione).getCap());
    String descrizione = ((SessioniInPresenza) sessione).getDescrizione();
    ps.setString(8, descrizione != null ? descrizione : "");
    ps.setInt(9, ((SessioniInPresenza) sessione).getId\_Corso());
    ps.setInt(10, sessione.getChef().getId\_Chef());
    ps.executeUpdate();
    // ...
}

// Metodo associaRicettaASessione
public void associaRicettaASessione(SessioniInPresenza sessione, Ricetta ricetta) {
    String query = "INSERT INTO sessione\_presenza\_ricetta (idsessionepresenza, idricetta) VAI
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, sessione.getId\_Sessione());
```

```
    ps.setInt(2, ricetta.getId\_Ricetta());
    ps.executeUpdate();
    // ...
}

// Metodo recuperaPartecipantiSessione
public ArrayList<UtenteVisitatore> recuperaPartecipantiSessione(SessioniInPresenza sessione) {
    String query = "SELECT p.* FROM partecipante p JOIN adesione\_sessionepresenza a ON p.idpar
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, sessione.getId\_Sessione());
    rs = ps.executeQuery();
    while (rs.next()) {
        UtenteVisitatore utente = new UtenteVisitatore(
            rs.getString("nome"),
            rs.getString("cognome"),
            rs.getString("email"),
            rs.getString("password"),
            rs.getDate("datadinascita").toLocalDate()
        );
        // ...
    }
    // ...
}
```

6.3.8 RicettaDao

La classe `ricettaDao` gestisce tutte le operazioni di persistenza relative alle ricette. Permette la creazione, il recupero, la modifica e la cancellazione delle ricette, nonché la gestione delle associazioni tra ricette e ingredienti. Le query SQL sono ottimizzate per garantire la coerenza tra le tabelle RICETTA, INGREDIENTE e PREPARAZIONEINGREDIENTE.

Principali metodi e query

- `memorizzaRicetta`: inserisce una nuova ricetta nella tabella RICETTA e ne recupera l'id generato.
- `getIngredientiRicetta`: recupera tutti gli ingredienti associati a una ricetta tramite join tra INGREDIENTE e PREPARAZIONEINGREDIENTE.
- `associaIngredientiARicetta`: associa un ingrediente a una ricetta tramite inserimento nella tabella PREPARAZIONEINGREDIENTE.
- `rimuoviAssociazioneIngrediente`: rimuove l'associazione tra una ricetta e un ingrediente.
- `cancellaricetta`: elimina una ricetta dalla tabella RICETTA.

Ruolo `ricettaDao` centralizza la logica di accesso e manipolazione dei dati relativi alle ricette, facilitando la gestione delle relazioni con gli ingredienti e garantendo la coerenza dei dati.

Esempi ricettaDao

```
// Metodo memorizzaRicetta
public void memorizzaRicetta(Ricetta ricetta) {
    String query = "Insert into Ricetta (Nome) values (?)";
    // ...
    ps = conn.prepareStatement(query, PreparedStatement.RETURN\_GENERATED\_KEYS);
    ps.setString(1, ricetta.getNome());
    ps.executeUpdate();
    generatedKeys = ps.getGeneratedKeys();
    if (generatedKeys.next()) {
        ricetta.setId\_Ricetta(generatedKeys.getInt(1));
    }
    // ...
}

// Metodo getIngredientiRicetta
public ArrayList<Ingredienti> getIngredientiRicetta(Ricetta ricetta) {
    String query = "Select I.Nome, I.UnitaDiMisura, I.IdIngrediente, P.QuanititaUnitaria from I
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, ricetta.getId\_Ricetta());
    rs = ps.executeQuery();
    while (rs.next()) {
        Ingredienti ingrediente = new Ingredienti(rs.getString("Nome"), rs.getFloat("QuanititaU
        ingrediente.setIdIngrediente(rs.getInt("IdIngrediente"));
        // ...
    }
    // ...
}
```

6.3.9 IngredientiDao

La classe `IngredientiDao` gestisce tutte le operazioni di persistenza relative agli ingredienti. Permette la creazione, il recupero, la modifica e la cancellazione degli ingredienti, nonché la verifica dell'utilizzo di un ingrediente in altre ricette. Le query SQL sono ottimizzate per garantire la coerenza tra le tabelle INGREDIENTE e PREPARAZIONEINGREDIENTE.

Principali metodi e query

- `memorizzaIngredienti`: inserisce un nuovo ingrediente nella tabella INGREDIENTE e ne recupera l'id generato.
- `modificaIngredienti`: aggiorna i dati di un ingrediente tramite una query di update.
- `cancellaingrediente/eliminaIngrediente`: elimina un ingrediente dalla tabella INGREDIENTE.
- `isIngredienteUsatoAltrove`: verifica se un ingrediente è utilizzato in altre ricette tramite query sulla tabella PREPARAZIONEINGREDIENTE.

- `recuperaQuantitaTotale`: recupera la quantità totale di un ingrediente per una ricetta tramite la tabella QuantitaPerSessione.

Ruolo `IngredientiDao` centralizza la logica di accesso e manipolazione dei dati relativi agli ingredienti, facilitando la gestione delle relazioni con le ricette e garantendo la coerenza dei dati.

Esempi IngredientiDao

```
// Metodo memorizzaIngredienti
public void memorizzaIngredienti(Ingredienti ingredienti) {
    String query = "INSERT INTO ingrediente (Nome, UnitaDiMisura) VALUES (?, ?::unitadimisura)"
    // ...
    ps = conn.prepareStatement(query, PreparedStatement.RETURN\_GENERATED\_KEYS);
    ps.setString(1, ingredienti.getNome());
    ps.setString(2, ingredienti.getUnitaMisura());
    ps.executeUpdate();
    generatedKeys = ps.getGeneratedKeys();
    if (generatedKeys.next()) {
        ingredienti.setIdIngrediente(generatedKeys.getInt(1));
    }
    // ...
}

// Metodo isIngredienteUsatoAltrove
public boolean isIngredienteUsatoAltrove(Ingredienti ingrediente) {
    String query = "SELECT COUNT(*) as cnt FROM PREPARAZIONEINGREDIENTE WHERE IdIngrediente = ?
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, ingrediente.getId\_Ingrediente());
    rs = ps.executeQuery();
    if (rs.next()) {
        int count = rs.getInt("cnt");
        return count > 0;
    }
    // ...
    return false;
}
```

6.3.10 GraficoChefDao

La classe `GraficoChefDao` gestisce il recupero delle statistiche mensili relative all'attività di uno chef, utilizzando la vista `vista_statistiche_mensili_chef`. Permette di ottenere valori aggregati come minimo, massimo, media di ricette per sessione, numero di corsi, sessioni in presenza e telematiche, e guadagno totale. Questi dati sono utilizzati per la generazione di grafici e report statistici.

Principali metodi e query

- `RicavaMinimo`, `RicavaMassimo`, `RicavaMedia`: recuperano rispettivamente il valore minimo, massimo e medio di ricette per sessione per uno chef in un dato mese e anno.
- `RicavaNumeroCorsi`: recupera il numero di corsi gestiti dallo chef nel periodo.
- `RicavaNumeroSessioniInPresenza`, `RicavaNumeroSesssioniTelematiche`: recuperano il numero di sessioni in presenza e telematiche svolte dallo chef.
- `ricavaGuadagno`: recupera il guadagno totale dello chef nel periodo.
- `stampaVistaCompleta`: stampa tutte le colonne della vista per uno chef, mese e anno (debug).

Ruolo `GraficoChefDao` centralizza la logica di accesso ai dati statistici aggregati per chef, facilitando la generazione di grafici, report e analisi sull'attività mensile degli chef.

Esempi GraficoChefDao

```
// Metodo RicavaMinimo
public int RicavaMinimo(Chef chef1, int mese, int anno){
    String quarry="SELECT min\_ricette\_in\_sessione FROM vista\_statistiche\_mensili\_chef WHE
    // ...
    ps = conn.prepareStatement(quarry);
    ps.setInt(1, chef1.getId\_Chef());
    ps.setInt(2, mese);
    ps.setInt(3, anno);
    rs = ps.executeQuery();
    if (rs.next()) {
        ValoreMinimo = rs.getInt(1);
    }
    // ...
}

// Metodo RicavaMedia
public float RicavaMedia(Chef chef1, int mese, int anno){
    String quarry="SELECT media\_ricette\_in\_sessione FROM vista\_statistiche\_mensili\_chef W
    // ...
    ps = conn.prepareStatement(quarry);
    // ...
}
```

6.3.11 CartaDiCreditoDao

La classe `CartaDiCreditoDao` gestisce tutte le operazioni di persistenza relative alle carte di credito associate agli utenti. Permette la creazione, il recupero, la cancellazione e la gestione delle relazioni tra utente e carta tramite la tabella `POSSIEDE`. Le query SQL sono ottimizzate per garantire la coerenza tra le tabelle `Carta` e `POSSIEDE`.

Principali metodi e query

- `getCarteByUtenteId`: recupera tutte le carte di credito associate a un utente tramite join tra POSSIEDE e CARTA.
- `memorizzaCarta`: inserisce una nuova carta di credito e la associa all'utente, oppure associa una carta già esistente.
- `cancellaCarta`: elimina la relazione tra utente e carta, e cancella la carta se non più associata ad altri utenti.

Ruolo `CartaDiCreditoDao` centralizza la logica di accesso e manipolazione dei dati relativi alle carte di credito, garantendo la sicurezza e la coerenza delle associazioni tra utenti e carte.

Esempi CartaDiCreditoDao

```
// Metodo getCarteByUtenteId
public List<CartaDiCredito> getCarteByUtenteId(int idUtente) {
    String query = "SELECT c.* FROM POSSIEDE p JOIN Carta c ON p.IdCarta = c.IdCarta WHERE p.Id
    // ...
    ps = conn.prepareStatement(query);
    ps.setInt(1, idUtente);
    rs = ps.executeQuery();
    while (rs.next()) {
        CartaDiCredito carta = new CartaDiCredito();
        carta.setIdCarta(rs.getString("IdCarta"));
        // ...
    }
    // ...
}

// Metodo memorizzaCarta
public void memorizzaCarta(CartaDiCredito carta, int idUtente) {
    String selectId = "SELECT IdCarta FROM Carta WHERE Intestatario = ? AND DataScadenza = ? AN
    // ...
    psSelect = conn.prepareStatement(selectId);
    // ...
}
```

6.3.12 BarraDiRicercaDao

La classe `BarraDiRicercaDao` gestisce le operazioni di ricerca e recupero dei valori enumerativi e delle categorie dal database, utili per la compilazione dinamica delle barre di ricerca e dei filtri nell'interfaccia utente. Permette di ottenere i valori degli enum definiti nel database (come frequenza delle sessioni, unità di misura, giorni della settimana) e le categorie di cucina disponibili.

Principali metodi e query

- `CeraEnumFrequenza`: recupera i valori dell'enum FrequenzaDelleSessioni (FDS) dal database tramite la funzione `enum_range`.

- `Categorie`: recupera le categorie di cucina disponibili tramite query sulla tabella TIPODICUCINA.
- `GrandezzeDiMisura`: recupera i valori dell'enum UnitaDiMisura dal database.
- `GiorniSettimanaEnum`: recupera i valori dell'enum Giorno dal database.

Ruolo `BarraDiRicercaDao` centralizza la logica di accesso ai valori enumerativi e alle categorie, facilitando la generazione dinamica di filtri e opzioni di ricerca nell'applicazione.

Esempi BarraDiRicercaDao

```
// Metodo CeraEnumFrequenza
public ArrayList<String> CeraEnumFrequenza() {
    String query = "SELECT unnest(enum\_range(NULL::FDS )) AS valore";
    // ...
    ps = conn.prepareStatement(query);
    rs = ps.executeQuery();
    while (rs.next()) {
        Enum.add(rs.getString("valore"));
    }
    // ...
}

// Metodo Categorie
public ArrayList<String> Categorie() {
    String query = "SELECT DISTINCT Nome as valore FROM TIPODICUCINA";
    // ...
    ps = conn.prepareStatement(query);
    rs = ps.executeQuery();
    while (rs.next()) {
        Categorie.add(rs.getString("valore"));
    }
    // ...
}

// Metodo GrandezzeDiMisura
public ArrayList<String> GrandezzeDiMisura() {
    String query = "SELECT unnest(enum\_range(NULL::UnitaDiMisura )) AS valore";
    // ...
    ps = conn.prepareStatement(query);
    rs = ps.executeQuery();
    while (rs.next()) {
        UnitaDiMisura.add(rs.getString("valore"));
    }
    // ...
}

// Metodo GiorniSettimanaEnum
```

```
public ArrayList<String> GiorniSettimanaEnum() {
    String query = "SELECT unnest(enum\_range(NULL::Giorno )) AS valore";
    // ...
    ps = conn.prepareStatement(query);
    rs = ps.executeQuery();
    while (rs.next()) {
        giorni.add(rs.getString("valore"));
    }
    // ...
}
```

6.4 JDBC

La piattaforma UninaFoodLab utilizza JDBC (Java Database Connectivity) per gestire la connessione e le operazioni con il database PostgreSQL. JDBC è uno standard Java che permette di interagire con database relazionali tramite driver specifici, garantendo portabilità e flessibilità.

6.4.1 Connessione al database

La connessione al database è gestita tramite la classe `ConnectionJavaDb`, che centralizza la logica di accesso e configurazione. I parametri di connessione (URL, utente, password) sono letti dal file di configurazione `db.properties`, che permette di gestire profili multipli e di cambiare facilmente ambiente senza modificare il codice sorgente.

6.4.2 Gestione delle risorse

La classe `SupportDb` fornisce metodi di utilità per la chiusura sicura di oggetti JDBC come `Connection`, `Statement` e `ResultSet`, riducendo il rischio di memory leak e semplificando la gestione delle risorse.

6.4.3 Dipendenze e configurazione Maven

Il file `pom.xml` include la dipendenza per il driver JDBC PostgreSQL (`org.postgresql:postgresql`), necessaria per la comunicazione tra l'applicazione Java e il database. Inoltre, sono configurate le versioni di Java e i plugin Maven per la compilazione e l'esecuzione del progetto.

6.4.4 Descrizione dei file principali

- `ConnectionJavaDb.java`: gestisce la lettura dei parametri di connessione dal file `db.properties`, carica il driver PostgreSQL e fornisce il metodo statico `getConnection()` per ottenere una connessione attiva al database. Supporta la selezione di profili multipli tramite la proprietà `db.profile`.
- `SupportDb.java`: fornisce metodi per la chiusura sicura di `Connection`, `Statement` e `ResultSet`, semplificando la gestione delle risorse JDBC e riducendo il rischio di errori.
- `pom.xml`: include la dipendenza per il driver JDBC PostgreSQL e configura la compilazione del progetto. La presenza della dipendenza `org.postgresql:postgresql` è fondamentale per la connessione al database.

Esempio di utilizzo

```
// Ottenere una connessione al database
Connection conn = ConnectionJavaDb.getConnection();
// Eseguire una query
PreparedStatement ps = conn.prepareStatement("SELECT * FROM tabella");
ResultSet rs = ps.executeQuery();
// ...
// Chiusura delle risorse
new SupportDb().closeAll(conn, ps, rs);
```

Questa architettura garantisce una connessione sicura, configurabile e facilmente estendibile tra l'applicazione Java e il database PostgreSQL.

6.4.5 Best practice e vantaggi

Questa architettura garantisce una connessione sicura, configurabile e facilmente estendibile tra l'applicazione Java e il database PostgreSQL. I vantaggi principali sono:

- Centralizzazione della logica di accesso ai dati e gestione delle risorse.
- Facilità di manutenzione e aggiornamento delle dipendenze.
- Robustezza contro errori di connessione e memory leak.
- Portabilità e flessibilità grazie all'uso di JDBC e Maven.
- Codice più pulito e conforme alle best practice Java.