



UNIVERSITÀ DEGLI STUDI FIRENZE

Relazione di progetto

CORSO DI INGEGNERIA DEL SOFTWARE
INGEGNERIA INFORMATICA

Kevin Madrigali, Andrea Innocenti

Anno accademico:
2023-2024

In breve: Si vogliono applicare le conoscenze acquisite durante il corso di Ingegneria del Software nella creazione di un'applicativo gestionale, scritto in linguaggio Java, in grado di consentire la prenotazione e gestione di prenotazioni all'interno di un fittizio circolo di campi da tennis e padel. Il progetto include l'uso di un database, un sistema di registrazione/login e personalizzazione degli utenti.

Nome di progetto: Court Reservation Manager

Indice

1	Introduzione	3
1.1	Obiettivo	3
1.2	Architettura	3
2	Analisi progettuale	4
2.1	Casi d'uso	4
2.2	Use Case Templates	5
2.3	Schema delle classi	7
2.4	Entity-Relationship Diagram	9
2.5	Pattern architettonico DAO	9
3	Implementazione	10
3.1	package ApplicationLayer	10
3.1.1	class GraphicInterfaceManager	10
3.2	package Context	12
3.2.1	classes Person, Client, SuperUser	12
3.2.2	class Reservation	12
3.2.3	class Court	13
3.2.4	class RentingKit	13
3.2.5	class TimeSlot	13
3.2.6	class Wallet	13
3.3	package Management	13
3.3.1	class ClientReservationManager, StandardClientReservationManager, PremiumClientReservationManager	13
3.3.2	class SuperUserReservationManager	14
3.3.3	class WalletManager	14
3.3.4	class CourtManager e TimeSlotManager	14
3.3.5	class MailManager e Utils	14
3.3.6	class GeneralLogging	15
3.4	package Database	15
3.4.1	class DatabaseManager	15
3.4.2	class ClientDaoImpl	16
3.4.3	class ReservationDaoImpl	16
3.4.4	class DatabaseLogging	17
4	Test	17
4.1	class ClientTest	17
4.2	class SuperUserTest	18
4.3	Risultati dei test	18
5	Confronto con le soluzioni architettoniche di Bing GPT	19
6	Librerie di terze parti	23

1. Introduzione

1.1. Obiettivo

Si vuole realizzare un software in grado di gestire le prenotazioni di campi da tennis o padel per un circolo sportivo. Gli utenti dovranno essere in grado di **creare un proprio account personale**, il quale include un **portafoglio digitale** che sarà utilizzato per le varie tipologie di transazioni. Gli **account degli utenti si suddividono in standard e premium**; per questi ultimi è previsto uno **sconto su ogni prenotazione** e un **sistema di punteggio**, il quale permette loro di usufruire di una prenotazione gratuita dopo un certo numero di prenotazioni effettuate. Il software dovrà anche tenere conto dei **giorni di festività**, in cui il circolo sarà chiuso, e vietare agli utenti di effettuare delle prenotazioni. Il circolo inoltre dà la possibilità di **noleggiare l'attrezzatura** per giocare ad un prezzo aggiuntivo al noleggio del campo (non incluso).

L'applicativo dovrà includere un **sistema di notifica agli utenti tramite mail** delle operazioni effettuate. Oltre agli utenti utilizzatori del servizio, il software dovrà anche essere in grado di **gestire uno o più utenti speciali** (superuser) in grado di **effettuare modifiche amministrative**, come visionare le prenotazioni ricevute, aggiungere o rimuovere campi, aggiungere o rimuovere slot orari. Nel caso in cui avvengano eliminazioni che incidano su delle prenotazioni ricevute, dovranno essere quindi annullate con emissione di relativo rimborso e invio di una mail di notifica verso l'utente.

1.2. Architettura

Di seguito è mostrato lo schema architetturale di progetto individuato e relative dipendenze.

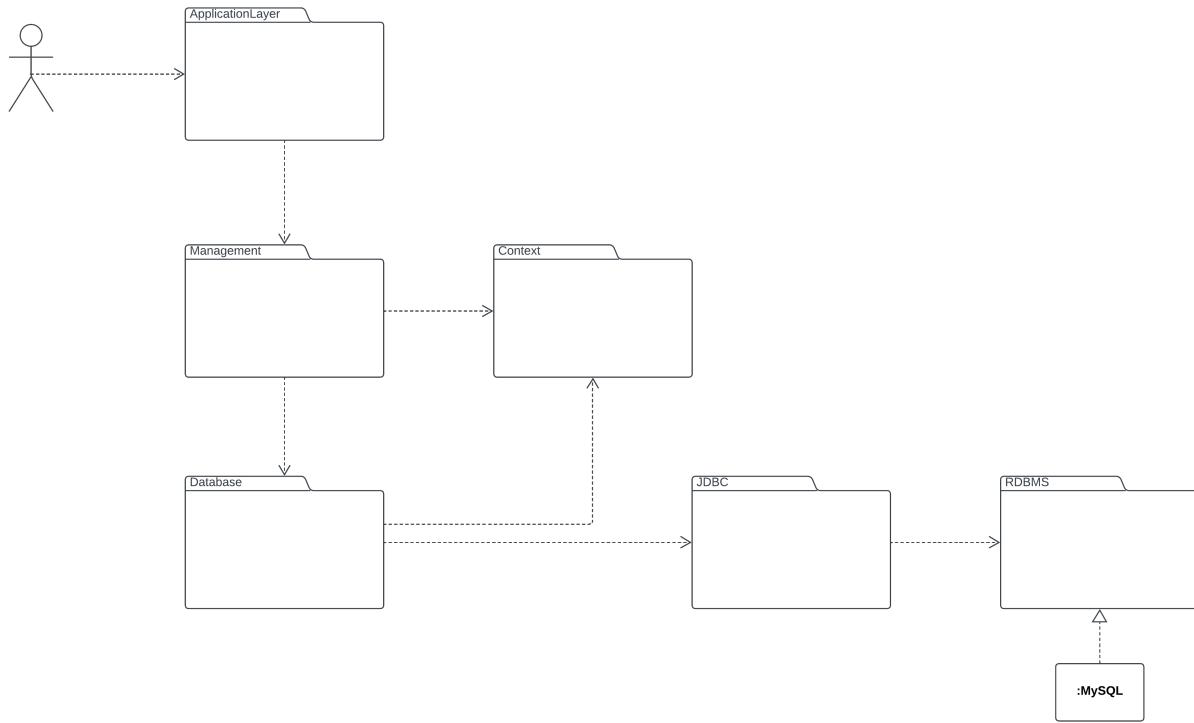


Figura 1: Diagramma delle dipendenze

Il software è stato realizzato in Java. Il package *Context* rappresenta il modello dei dati, mentre il package *Management* rappresenta la gestione delle operazioni. Per rendere i dati persistenti, il package *Database* è usato per dialogare con un database remoto MySQL, utilizzato JDBC (Java Database Connectivity). Il software include un package *ApplicationLayer* dedicato all'effettivo funzionamento del programma tramite un'interfaccia grafica su console (CLI). Il testing è stato effettuato tramite il framework JUnit.

I diagrammi delle classi e dei casi d'uso seguono lo standard UML (Unified Modeling Language) e sono stati realizzati con il software "Lucidchart". Per il database remoto è stato utilizzato il servizio di hosting "Fileless.io".

2. Analisi progettuale

2.1. Casi d'uso

La figura nella pagina successiva mostra il diagramma dei casi d'uso, con gli attori coinvolti nel sistema e i vari modi con cui essi possono interagirci.

I casi d'uso mostrati sono stati attualmente tutti implementati nel software.

Sono individuati **2 attori** coinvolti nel sistema:

- **Utenti**, utilizzatori dei servizi offerti dal circolo sportivo.
- **Amministratori**, i quali possono effettuare alcune operazioni di gestione e visualizzare le prenotazioni.

Un *membro dell'amministrazione* ha la possibilità di rimuovere o aggiungere campi, aggiungere o rimuovere slot orari prenotabili, di visionare tutte le prenotazioni ricevute, visionare i clienti e attuare modifiche. Un *utente* invece necessita la creazione di un account personale, con relativo portafoglio digitale abbinato, con i quali potrà usufruire dei servizi e pagarne gli importi. Il portafoglio digitale potrà essere ricaricato dal software. Gli utenti hanno la possibilità di eseguire l'upgrade da account standard a premium per usufruire di alcuni vantaggi.

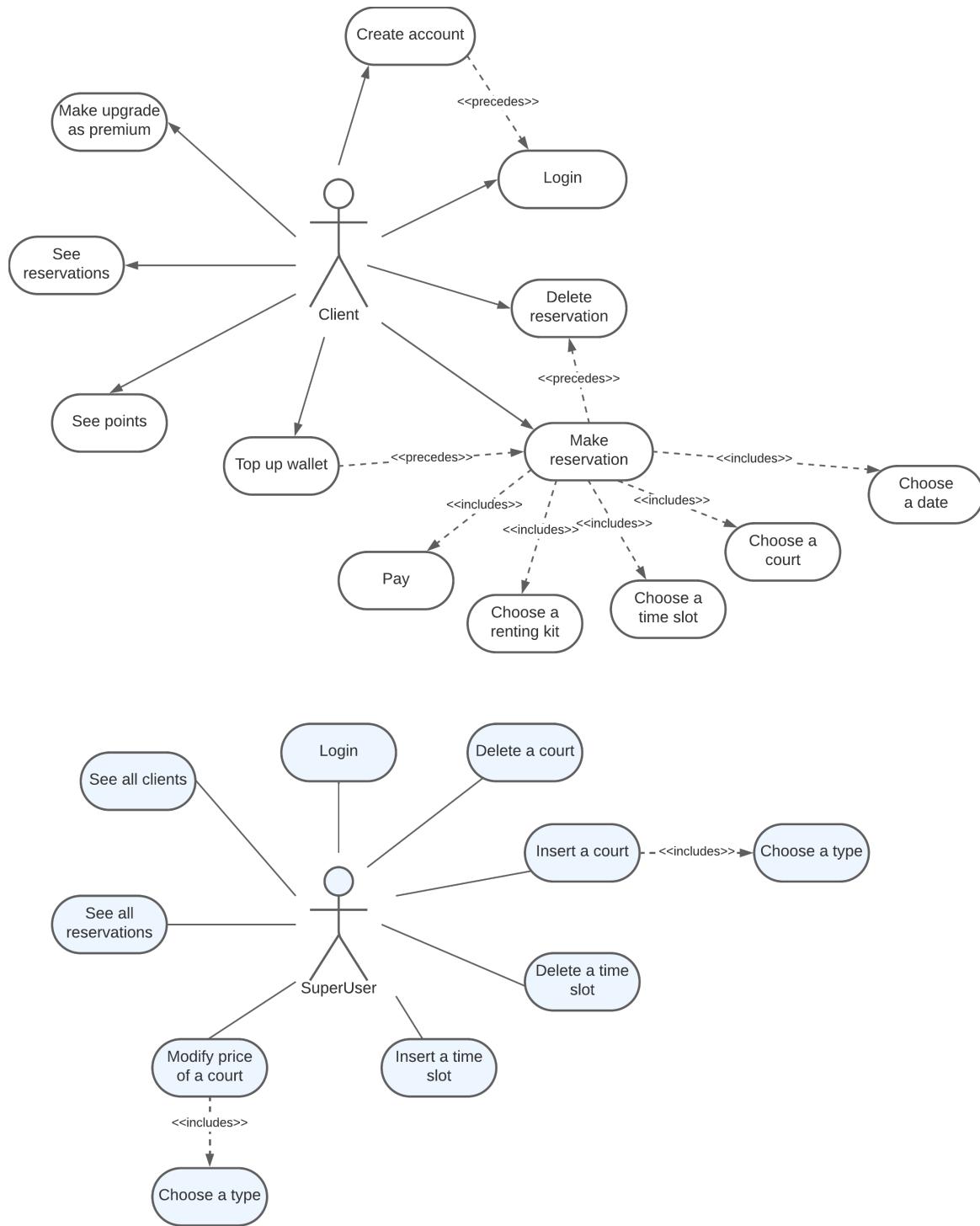


Figura 2: Diagramma dei casi d'uso

2.2. Use Case Templates

In seguito vengono riportati anche i template per alcuni dei casi d'uso implementati. Per ognuno di questi è mostrata una descrizione dettagliata, gli attori coinvolti ed il percorso usuale di funzionamento. Per alcuni casi d'uso sono inoltre riportati eventuali percorsi alternativi, oltre che eventuali pre-condizioni e post-condizioni.

Use case 1	Prenotazione campo
Description	Un cliente prenota un campo
Level	User goal
Actor	Client
Basic reservation	1. Il cliente sceglie una data 2. Viene controllato che la data sia valida e che non sia in un giorno festivo (italiano) 3. Il cliente sceglie un campo 4. Il cliente sceglie una finestra oraria (time slot) tra quelle disponibili 5. Il cliente (opzionale) prenota uno o più kits da gioco (renting kits) 6. Se il cliente è premium, viene controllato se il ha raggiunto i punti regalo (gift points) per usufruire di una prenotazione omaggio 7. Il cliente visualizza il prezzo totale, scontato se è premium 8. Viene controllato che il cliente abbia abbastanza denaro nel borsellino (wallet) per effettuare la prenotazione 9. Se il cliente è premium gli vengono accreditati dei punti prenotazione (reservation points) 10. Viene confermata la prenotazione con l'invio di una email
Alternative reservation 1	2. La data selezionata è errata o è in un giorno festivo, viene notificato un messaggio di errore
Alternative reservation 2	5. La finestra oraria selezionata non è disponibile, viene notificato un messaggio di errore
Alternative reservation 3	7. Il cliente ha raggiunto i punti regalo e sceglie di utilizzarli 8. Viene confermata la prenotazione con l'invio di una email
Alternative reservation 4	8. Il cliente non ha abbastanza denaro nel borsellino, la prenotazione viene annullata
Alternative reservation 5	10. C'è stato un errore di connessione con il server, l'operazione viene annullata e il cliente visualizza un messaggio di errore
Pre conditions	Deve esserci almeno un campo e il cliente deve disporre di un account
Post condition	Il campo prenotato risulta non disponibile agli altri clienti nella finestra oraria e nel giorno scelto

(a) Use Case Template 1

Use case 2	Annulloamento prenotazione
Description	Un cliente annulla una prenotazione ad un campo
Level	User goal
Actor	Client
Basic delete reservation	1. Il cliente visualizza tutte le sue prenotazioni future 2. Il cliente sceglie l'id di una prenotazione 3. Viene restituito il denaro usato per la prenotazione 4. Viene inviata una mail di cancellazione confermata
Alternative delete reservation 1	3. Il cliente aveva effettuato la prenotazione con i punti regalo, dunque gli vengono restituiti 4. Viene inviata una mail di cancellazione confermata
Alternative delete reservation 2	4. Il cliente aveva effettuato la prenotazione con lo status di premium, dunque gli vengono tolti i punti per quella prenotazione 5. Viene inviata una mail di cancellazione confermata
Alternative delete reservation 3	4. C'è stato un errore di connessione con il server, l'operazione viene annullata e il cliente visualizza un messaggio di errore
Pre condition	Il cliente deve aver fatto almeno una prenotazione

(b) Use Case Template 2

Use case 3	Registrazione
Description	Un cliente effettua la registrazione
Level	User goal
Actor	Client
Basic registration	1. Il cliente inserisce nome, cognome, email, password, numero di telefono (opzionale) 2. Viene fatto un controllo sulla validità dell'email (formato parola@dominio) 3. Viene controllato se l'email è già presente nel database 4. Viene confermata la registrazione con l'invio di una email
Alternative registration 1	4. L'email è in un formato sbagliato, viene notificato un messaggio d'errore
Alternative registration 2	3. L'email è già presente nel database, viene notificato un messaggio d'errore
Alternative registration 3	4. C'è stato un errore di connessione con il server, l'operazione viene annullata e il cliente visualizza un messaggio di errore
Post condition	Il cliente ora può effettuare il login

(c) Use Case Template 3

Use case 4	Inserimento campo
Description	L'amministratore inserisce un campo
Level	User goal
Actor	Super User
Basic court insert	1. L'amministratore visualizza i campi disponibili 2. L'amministratore sceglie un id per il nuovo campo 3. Viene controllato se l'id è già presente nel database 4. L'amministratore sceglie un tipo di campo tra quelli possibili
Alternative court insert 1	4. l'id è già presente, viene notificato un messaggio d'errore
Alternative court insert 2	5. Il tipo di campo scelto non esiste, viene notificato un messaggio d'errore
Alternative court insert 3	5. C'è stato un errore di connessione con il server, l'operazione viene annullata e il cliente visualizza un messaggio di errore
Post condition	I clienti ora possono prenotare questo campo

(d) Use Case Template 4

Use case 5	Eliminazione fascia oraria
Description	L'amministratore elimina una fascia oraria
Level	User goal
Actor	Super User
Basic court insert	1. L'amministratore visualizza le fasce orarie disponibili 2. L'amministratore seleziona l'id della fascia oraria da eliminare 3. Viene controllato se l'id è presente nel database 4. Eliminata la fascia oraria viene controllato se ci sono prenotazioni a quella fascia 5. Se ci sono, vengono rimborsati e notificati i clienti, vengono lasciati loro i punti acquisiti (se premium) per il disagio
Alternative ts delete 1	4. l'id selezionato non esiste, viene notificato un messaggio di errore
Alternative ts delete 2	5. Uno o più clienti avevano effettuato una o più prenotazioni in quella finestra con i punti regalo, dunque vengono restituiti loro i punti utilizzati
Alternative ts delete 3	5. C'è stato un errore di connessione con il server, l'operazione viene annullata e il cliente visualizza un messaggio di errore
Post condition	I clienti ora non possono più prenotare in questa fascia oraria

(e) Use Case Template 5

I casi d'uso riportati sono di livello *User goal*, che rappresentano un'attività elementare che è possibile svolgere tramite il gestionale.

- Il **use case template 3a (1)** schematizza il percorso base da effettuare per inviare una *prenotazione* di un campo, con percorsi alternativi che riguardano casistiche per cui principalmente la prenotazione non può andare a buon fine.
- Il **use case template 3b (2)** riguarda l'eliminazione di una prenotazione effettuata da parte di un *Client*, applicabile solo per prenotazioni effettuate per almeno il giorno successivo.
- Il **use case template 3c (3)** riguarda il processo di registrazione di un account per un *Client*, permettendo come post-condizione la possibilità di effettuare il login.
- Il **use case template 3d (4)** riguarda l'aggiunta di un campo da gioco da parte di un amministratore (*SuperUser*). Questo permette quindi (come post-condizione) di permettere agli utenti di effettuare prenotazioni anche per questo nuovo campo inserito.
- Il **use case template 3e (5)** infine riguarda l'eliminazione di una fascia oraria da parte di un amministratore (*SuperUser*). Questo provoca, come post-condizione, la non possibilità per gli utenti di effettuare prenotazioni per qualsiasi campetto in quel preciso slot orario (ad esempio per variazioni di orari del circolo sportivo).

2.3. Schema delle classi

Il diagramma delle classi riportato nella figura della pagina successiva rappresenta le classi implementate e come queste interagiscono tra loro. Il diagramma mostra anche il design pattern DAO, spiegato dettagliatamente nella sezione 2.5.

Le classi implementate, sono suddivise secondo dei package, ognuno dei quali con uno scopo specifico:

- **ApplicationLayer**, contiene la classe **GraphicInterfaceManager**, la quale si occupa di fornire un'interfaccia grafica tramite *command line* all'utente e di seguire certi *percorsi logici* per l'esecuzione delle funzionalità fornite dal package **Management**, passando l'input processato dall'utilizzatore del software.
- **Management**, contiene tutte le classi addette alla gestione delle prenotazioni per gli utenti (**ClientReservationManager**, **StandardClientReservationManager**, **PremiumClientReservationManager**) e una classe specifica per gli amministratori che permetta la visualizzazione e gestione delle prenotazioni effettuate al circolo sportivo (**SuperUserReservationManager**) e altre classi per la gestione dei wallet, campi e timeslot. In sostanza sono esposte le operazioni effettuabili sui principali modelli del programma.
- **Context**, in cui si trovano le classi che rappresentano il *modello dei dati*. Queste classi rappresentano la modalità con la quale le sono stati rappresentati i dati e gli oggetti nell'applicativo.
- **Database**, il quale include le classi necessarie per l'attuazione del pattern DAO (Data Access Object) e una classe specifica (**DatabaseManager**) per la gestione e instaurazione della connessione al database. Le classi DAO ereditano da una superclasse (**GeneralDAOSettings**) così da condividere un singolo oggetto **DatabaseManager** tra tutti. Si ha una classe DAO per ogni modello dell'applicativo, il quale espone tutte le operazioni effettuabili su tale modello che necessiti un'operazione verso il database.

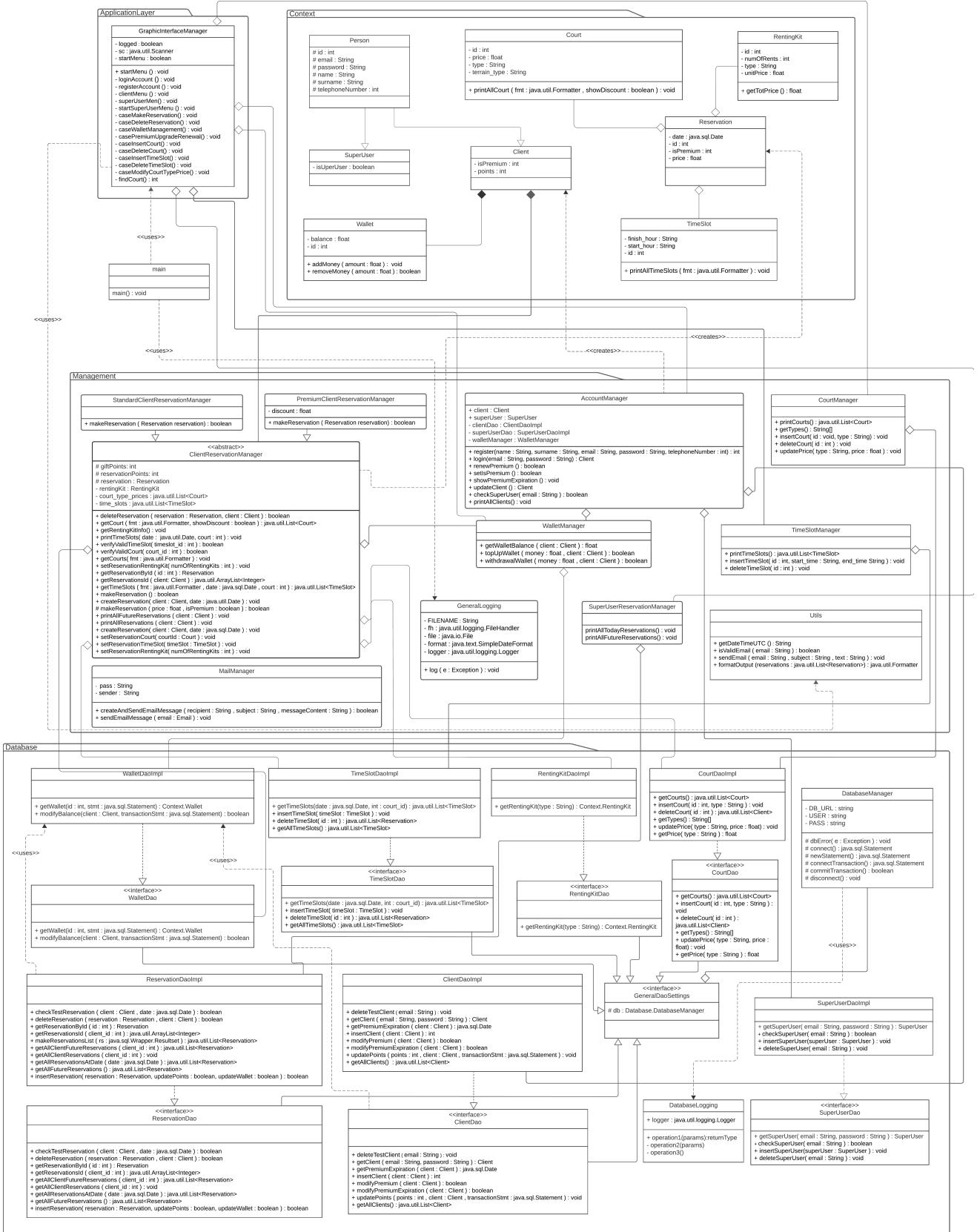


Figura 3: Diagramma delle classi

2.4. Entity-Relationship Diagram

Nella figura successiva è presentata la struttura del database utilizzato nell'applicativo.

Le *entità principali* coinvolte sono la generica persona utilizzatrice dell'applicativo (**Person**), che si suddivide in due entità specializzate cliente (**Client**) e amministratore (**SuperUser**), le prenotazioni (**Reservation**) e le entità relative alla prenotazione (**RentingKit**, **TimeSlot**, **Court**).

Una prenotazione deve includere un riferimento a un *campo scelto* ed il relativo *slot orario*, ma non necessariamente un riferimento a un *kit da gioco*, in quanto è opzionale. Per limitare la presenza di valori nulli, dato che un utente è in grado di noleggiare anche più kit, la relazione con la *quantità* di kit noleggiati è stata implementata nel database con l'inserimento di una tabella aggiuntiva che leggi la *prenotazione*, il *kit da gioco* (a seconda del campo scelto può essere di tennis o padel) e la *quantità* scelta.

Per quanto riguarda il cliente (**Client**), si ha una relazione uno-ad-uno con l'entità portafoglio (**Wallet**) in quanto ogni cliente deve avere un proprio wallet associato. Un cliente inoltre, può decidere di iscriversi ad un abbonamento premium annuale (**PremiumSubs**) per usufruire di alcuni vantaggi.

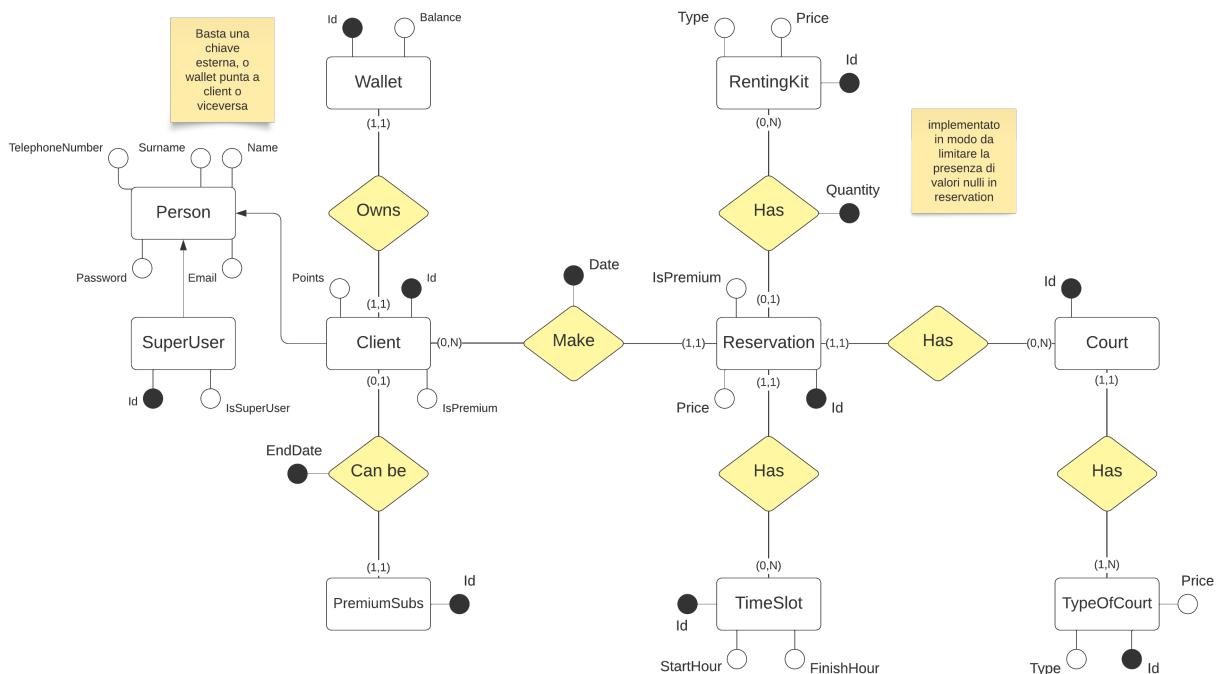


Figura 4: Diagramma Entità-Relazione

2.5. Pattern architetturale DAO

Il **DAO (Data Access Object)** è un pattern architetturale che fornisce un'interfaccia astratta per la *gestione della persistenza dei dati*, nel nostro caso utilizzando un database remoto MySQL.

Questo pattern permette di offrire alle classi le operazioni CRUD da effettuare sui dati, senza esporre in maniera diretta i dettagli implementativi del database e come avviene la persistenza, aderendo quindi al *principio di singola responsabilità*. Nel nostro caso, come si può vedere dalla Figura 3, è presente un'interfaccia generica DAO (**GeneralDaoSettings**) che contiene un oggetto **DatabaseManager** condiviso tra le varie classi DAO, le quali dovranno implementarla per gestire le operazioni al database tramite singole connessioni. Ad ogni entità è associato un DAO, si hanno quindi delle *interfacce specifiche* che estendono l'*interfaccia generale DAO* e hanno lo scopo di specificare i tipi effettivi sui cui le *classi DAO concrete* lavoreranno.

È importante notare che esistono delle *dipendenze transitive* tra i modelli, ad esempio un **Client** deve avere un riferimento al suo **Wallet**. La responsabilità di concatenare le operazioni di creazione degli oggetti e installare le dipendenze è contenuta nei DAO, perciò alcune classi DAO possono collaborare con altre classi DAO.

3. Implementazione

Il progetto è stato sviluppato in Java. Abbiamo deciso di utilizzare *Jetbrains IntelliJ* come strumento per la gestione delle librerie e l'organizzazione del progetto. Di seguito descriviamo i contenuti dei file di progetto.

3.1. package ApplicationLayer

Questo package contiene la classe addetta a fornire un'interfaccia grafica basata su linea di comando (CLI).

3.1.1. class GraphicInterfaceManager

Rappresenta la classe in carico di gestire l'interfaccia del software. Permette di seguire degli schemi logici, prendendo l'input dall'utente e chiamando le varie funzioni rese disponibili dal package **Management**. La classe implementa vari metodi in grado di gestire gli input dell'utente, il quale sceglie le operazioni da eseguire da due menu di visualizzazione. All'avvio del programma l'utente visualizza il primo menu (figura 5) che gli consente di effettuare il login o la registrazione. Una volta loggato, visualizzerà il secondo menu (figura 6).

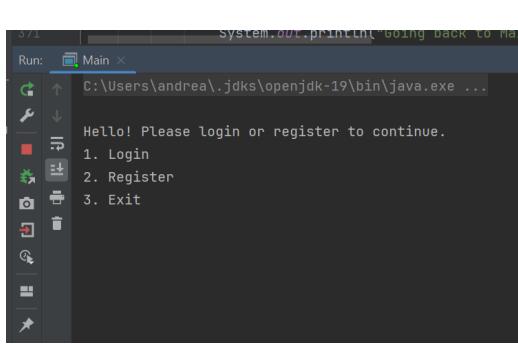


Figura 5: Menu all'avvio

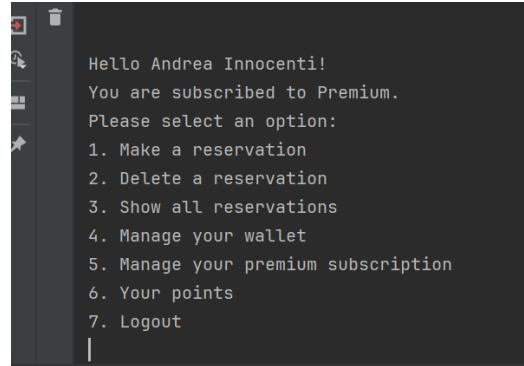


Figura 6: Menu del client

La classe implementa 2 metodi per gestire i 2 menu del client, `startMenu()` e `clientMenu()`. In entrambi i casi una clausola `switch/case`, elabora la scelta dell'utente; nel caso di `startMenu()` vengono chiamate rispettivamente `registerAccount()` per la registrazione e `loginAccount()` per il login, funzioni che delegano il lavoro ad `AccountManager` (listing 1).

Per quanto riguarda il menu utente è interessante analizzare la creazione della prenotazione, funzionalità chiave del software e di maggior interesse applicativo. Come possiamo notare dal (listing 2, vengono effettuati i controlli sulla validità della data (sul fatto che sia festiva o meno tramite la funzione `holidayManager.isHoliday()`, della libreria de.jollyday e che la data selezionata sia futura), dell'id del campo e dell'id dello slot orario. La creazione dell'oggetto `reservation` è delegata al `ReservationManager` del client in questione, che sarà `StandardReservationManager` se il client è standard, `PremiumReservationManager` se è premium. Man mano che gli input vengono processati l'oggetto `reservation` viene popolato, tramite il `ReservationManager`, con data, id del campo, slot orario e numero di kit da gioco. Se la prenotazione è andata a buon fine (riga 20) viene inviata una mail di conferma al client.

Listing 1: `startMenu()`

```
1 public void startMenu() {
2     //... stampa menu di visualizzazione ...
3     int choice;
4     // ... codice per ricevere l'input ...
5     switch (choice) {
6         case 1 -> {
```

```

7         startMenu = false;
8         loginAccount();
9     }
10    case 2 -> {
11        startMenu = false;
12        registerAccount();
13    }
14    // ...
15 }

```

Listing 2: caseMakeReservaion()

```

1 private void caseMakeReservation() {
2 //... altro codice ...
3     date = Date.valueOf(sc.nextLine()); // ricezione data
4 // ... controllo sul fatto che la data sia futura e non sia festiva ...
5     accountManager.client.getReservationManager().creteReservation(accountManager.
6         client, date); // creazione oggetto reservation
7     //... visualizzazione dei campi ...
8     court = sc.nextInt();
9     //... controllo che l'id del campo sia valido ...
10    accountManager.client.getReservationManager().setReservationCourt(court); //
11        aggiunta campo a reservation
12    //... visualizzazione degli slot orari ...
13    slot = sc.nextInt(); // ricezione id slot
14    // ... controllo sull'input ...
15    accountManager.client.getReservationManager().setReservationTimeSlot(slot);
16
17    // ... visualizzazione info su i kit da gioco ...
18    int numOfRent = sc.nextInt(); // scelta del numero di kit
19    //... controllo della validita' dell'input ...
20    accountManager.client.getReservationManager().setReservationRentingKit(numOfRent);
21        // aggiunta del numero di kit da gioco a reservation
22    //...
23    if(accountManager.client.getReservationManager().makeReservation()) // se la
24        prenotazione e' andata a buon fine viene inviata una mail di conferma
25    { System.out.println("ReservationSuccessful.");
      Utils.sendEmail( ... );
      // ...
    }
}

```

Se l'utente loggato è invece un **SuperUser** (amministratore), la cui registrazione avviene manualmente direttamente dal pannello di controllo del database, verrà visualizzato un menu differente (figura 7). Per esempio, per quanto riguarda l'inserimento di un nuovo slot orario, la funzione `caseInsertTimeSlot()`, provvederà a mostrare gli slot già disponibili, a ricevere un id per il nuovo slot, un orario di inizio e uno di fine, a controllare che siano validi e a delegare la sua creazione a `timeSlotManager` (listing 3)

```

Hello andrea innocent!
You are super user.
Please select an option:
1. Insert a court
2. Delete a court
3. Modify price of a court
4. Insert a time slot
5. Delete a time slot
6. See all reservations
7. See all future reservations
8. See all clients
9. Logout

```

Figura 7: Menu dell'amministratore

Listing 3: startMenu()

```

1 private void caseInsertTimeSlot() {
2     List<TimeSlot> timeSlots = timeSlotManager.printTimeSlots(); // stampa time
3         slot già disponibili
4     id = sc.nextInt();
5     // ... controllo se l'id scelto è già presente in timeSlots ...
6     start_hour = sc.nextInt();
7     // ... controllo se l'ora di inizio è già presente in timeSlots ...
8     end_hour = sc.nextInt();
9     // ... controllo se l'ora di inizio è già presente in timeSlots ...
10    timeSlotManager.insertTimeSlot(id, String.valueOf(start_hour), String.valueOf(
11        end_hour)); // delega a timeSlotManager l'inserimento del timeSlot
12 }

```

3.2. package Context

Questo package contiene le classi che rappresentano il modello dei dati.

3.2.1. classes Person, Client, SuperUser

Le classi **Client** e **SuperUser** derivano entrambe dalla classe **Person**. Quest'ultima definisce le informazioni base che rappresentano una persona nel nostro applicativo.

Le due estensioni, invece, aggiungono a quest'ultima i dati specifici del cliente o dell'amministratore.

- **Client**, il quale aggiunge attributi necessari per l'utente, quali un riferimento a un *Wallet*, contatore dei *punti* accumulati, un riferimento a *ClientReservationManager* e un attributo booleano *isPremium* per indicare se il cliente ha sottoscritto un abbonamento premium.
- **SuperUser**, il quale necessita in aggiunta di un booleano *isSuperUser* per certificare il suo stato.

3.2.2. class Reservation

Questa classe contiene tutti i dati che caratterizzano una prenotazione: un identificativo, campo da gioco, data, cliente di appartenenza, slot orario scelto, prezzo e un attributo booleano *isPremium* per indicare se la prenotazione è stata effettuata dal cliente come utente premium (così in caso di eliminazione della prenotazione come utente standard, gli saranno comunque tolti i punti accumulati con la prenotazione).

3.2.3. class Court

Questa classe contiene i dati che caratterizzano un campo da gioco, ovvero un identificativo, il tipo (tennis o padel), il tipo di terreno (varia per i campi da tennis) ed il prezzo.

3.2.4. class RentingKit

La classe caratterizza le proprietà di un kit da gioco, dunque un identificativo, un costo unitario, una quantità di noleggi ed il tipo (tennis o padel).

3.2.5. class TimeSlot

Rappresenta lo slot orario, tramite un identificativo a cui è quindi associato un orario di inizio e di fine.

3.2.6. class Wallet

Questa classe rappresenta il portafoglio digitale associato ad un cliente, con un identificativo e il saldo disponibile.

3.3. package Management

Questo package contiene le classi che rappresentano le operazioni logiche effettuabili sul modello dei dati e sul database, tramite l'utilizzo delle varie classi concrete DAO fornite dal package Database.

3.3.1. class ClientReservationManager, StandardClientReservationManager, PremiumClientReservationManager

Come descritto nel diagramma di classe UML, ClientReservationManager gestisce le operazioni relative a stampa, inserimento, modifica e cancellazione della prenotazione da parte del client, compone un oggetto Reservation che viene da lei stessa creato e che poi tramite la funzione MakeReservation(), preparato per essere inserito nel database tramite un ReservationDaoImpl. Le classi StandardClientReservationManager e PremiumReservationManager, che estendono ClientReservationManager, gestiscono le differenze al momento della prenotazione fra cliente premium e standard. In particolare, come si può notare dal listing 4, il metodo makeReservation() viene sovrascritto in PremiumClientReservationManager per gestire l'assegnazione ed in caso l'utilizzo dei punti regalo. Il codice a comune è condiviso dalle due classi in un altro metodo MakeReservation() sovraccaricato nella super classe, in cui viene anche controllato che vi siano sufficienti fondi per effettuare l'operazione.

Listing 4: MakeReservationPremium()

```
1  @Override
2  public boolean makeReservation() {
3      // ... altro codice ...
4      if (reservation.getClient().getPoints() >= giftPoints) {
5          // ... gestisce il caso in cui l'utente ha raggiunto i punti regalo ...
6          reservationDao.makeReservation(reservation, true, false); // effettua la
7              // prenotazione nel caso in questione
8      }
9      // ... altro codice ...
10     price = reservation.getPrice() * discount; // calcola il prezzo scontato
11     // ... altro codice ...
```

```

12     reservation.getClient().setPoints(reservation.getClient().getPoints() +
13         reservationPoints); // assegna i punti al client
14     return super.makeReservation(price, isPremium); // delega alla super classe
15     la creazione della prenotazione
}
}

```

3.3.2. class SuperUserReservationManager

Questa classe fornisce i metodi al **SuperUser** per visualizzare tutte le prenotazioni future del circolo, o in generale l'archivio di tutte le prenotazioni, sia passate che future.

3.3.3. class WalletManager

Questa classe fornisce i metodi per modificare il bilancio del **Wallet**, compone un **WalletDaoImpl** a questo scopo.

3.3.4. class CourtManager e TimeSlotManager

Queste due classi svolgono funzioni del tutto analoghe di inserimento, cancellazione e modifica dei rispettivi oggetti gestiti. Di particolare interesse è la funzione di cancellazione, che è mostrata nel caso specifico di **CourtManager** nel listing 5, ma del tutto analoga in **TimeSlotManager**, che prevede, a seguito della cancellazione dell'oggetto dal database, la notifica di tutti i client a cui è stata cancellata la prenotazione a causa di questa modifica.

Listing 5: deleteCourt()

```

1 public void deleteCourt(int id) {
2     List<Reservation> reservations = courtDao.deleteCourt(id);
3     if (reservations != null)
4         for (Reservation r : reservations) {
5             if (r.getPrice() == 0)
6                 // notifica gli utenti che hanno effettuato la prenotazione coi punti
7                 regalo
8                 Utils.sendEmail(r.getClient().getEmail(), "//_message");
9             else
10                 Utils.sendEmail(r.getClient().getEmail(), "//_message");
11         }
}

```

3.3.5. class MailManager e Utils

MailManager gestisce l'invio delle mail da parte del software, con indirizzo email "notifier.courtreservation@gmail.com". La classe **Utils** invece si interfaccia direttamente con gli altri manager, si occupa di verificare la validità della mail durante la registrazione, di fare forwarding a **MailManager** per l'invio delle mail e fornisce un metodo per restituire una data.

3.3.6. class GeneralLogging

Questa classe salva in file .log i messaggi di errore lanciati dal programma.

3.4. package Database

Come spiegato in 2.5, la gestione del database mySql è stata effettuata tramite il pattern Dao, le cui classi sono state implementate in questo package. Visto il notevole numero di classi, limitiamo la trattazione alcuni casi d'interesse. In questi casi è stato necessario utilizzare delle **transazioni** sul database, le quali assicurano che in caso di errore non venga eseguita nessuna delle operazioni all'interno della transazione stessa.

3.4.1. class DatabaseManager

Inanzitutto è doveroso descrivere la classe **DatabaseManager**. Ogni Dao ha un riferimento a questa classe, che si occupa di avviare la connessione con il database tramite il metodo **connect()** oppure in caso di transazione con metodo **connectTransaction()**, descritte nel listing 6. Nel caso debba essere effettuata una transazione, come vediamo nel listing, sarà poi necessario fare un **commit** manuale, tramite la funzione **commitTransaction()**. Per effettuare la disconnessione viene chiamato il metodo **disconnect()** (listing ?? ??).

Listing 6: DatabaseManagement

```
1 // ritorna un java.sql.Statement
2 Statement connect() {
3     try {
4         Class.forName("com.mysql.cj.jdbc.Driver");
5         conn = DriverManager.getConnection(DB_URL, USER, PASS);
6         return conn.createStatement();
7     // ... altro codice ...
8 }
9
10    Statement connectTransaction() {
11        try {
12            Class.forName("com.mysql.cj.jdbc.Driver");
13            conn = DriverManager.getConnection(DB_URL, USER, PASS);
14            conn.setAutoCommit(false); // blocco dell'auto commit
15            conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
16            // imposta il livello di isolamento
17            return conn.createStatement();
18        // ... altro codice ...
19    }
20
21    void disconnect() {
22        try {
23            if (conn != null) {
24                conn.close();
25            }
26        } catch (SQLException e) {
27            dbError(e);
28            System.err.println("Error during disconnection.");
29        }
30    }
```

3.4.2. class ClientDaoImpl

Per fornire un esempio di transazione, analizziamo l'inserimento di un client nel database, tramite la funzione `insertClient()`. Nell'inserimento di un client è necessario anche creare un `Wallet`. Le due operazioni non possono avvenire l'una senza l'altra, dunque vengono inserite all'interno di una transazione, come descritto nel listing 7.

Listing 7: ClientDaoImplInsert

```
1 @Override
2     public int insertClient(Client client) {
3         Statement stmt = db.connectTransaction();
4         // ... altro codice ...
5         stmt.executeUpdate("INSERT INTO client(name, surname, email, password, telephone_number, points, is_premium) VALUES ('" + client.getName() +
6             "','" + client.getSurname() + "', '" + client.getEmail() + "', '" +
7             client.getPassword() + "', '" + client.getTelephoneNumber() + "', '" +
8             client.getPoints() + "', '" + client.getIsPremium() + "')"); // inserimento client
9         // ... select per prelevare l'id del client che e' autoincrementale ...
10        stmt.executeUpdate("INSERT INTO wallet(id, balance, client) VALUES ('" +
11            client_id + "', '0', '" + client_id + "')"); // inserimento wallet
12        stmt.executeUpdate("update client set wallet = '" + client_id + "' where id = '" +
13            client_id + "'");
14        // aggiornamento del client con riferimento al wallet appena creato
15        rs.close();
16        db.commitTransaction();
17        // ... altro codice ...
18    }
```

3.4.3. class ReservationDaoImpl

Un altro esempio interessante è il caso dell'eliminazione di una prenotazione. In questo caso la classe `ReservationDaoImpl`, deve gestire, con il metodo `deleteReservation()`, anche la restituzione dei soldi al cliente e l'aggiornamento dei punti se la prenotazione è stata effettuata con lo status di premium. Il metodo dunque, all'interno di una transazione, delega ai rispettivi Dao l'aggiornamento delle tabelle corrispondenti, poi elimina la prenotazione (listing 8).

Listing 8: ReservationDaoImpl

```
1 @Override
2     public boolean deleteReservation(Reservation reservation, Client client) {
3         try {
4             Statement stmt = db.connectTransaction();
5             assert stmt != null;
6             ClientDaoImpl clientDao = new ClientDaoImpl();
7
8             clientDao.updatePoints(client.getPoints(), client, stmt); // aggiorna i punti
9             client.getWallet().addMoney(reservation.getPrice());
10            WalletDaoImpl walletDao = new WalletDaoImpl();
11            walletDao.modifyBalance(client, stmt); // aggiorna il bilancio
```

```

12     stmt.executeUpdate("DELETE FROM reservation WHERE id = " + reservation.
13         getId() + ""); // elimina la prenotazione
14     db.commitTransaction();
15     // ... altro codice ...
16 }

```

3.4.4. class DatabaseLogging

Questa classe si occupa di salvare su un file .log gli i messaggi di errori lanciati dalle classi del package.

4. Test

Con lo scopo di controllare la correttezza delle funzionalità implementate, sono stati realizzati dei **test di unità** (unit test), con l'aiuto della libreria java **JUnit**. In particolare, essendo l'applicativo *input based*, è stato necessario realizzare delle funzioni di test che simulassero l'input dell'utente, sia per quanto riguarda le operazioni ad appannaggio di un **client**, sia per quelle relative ad un **SuperUser**. Vengono presentati in seguito degli esempi di funzioni di test per entrambi i soggetti.

4.1. class ClientTest

Qui vengono testate tutte le operazioni che può effettuare un **Client**, dunque registrazione, login, ricarica del borsellino, prenotazione, cancellazione di prenotazione e upgrade a cliente premium. Di seguito, nel listing 9, il test per la prenotazione senza denaro sufficiente nel borsellino. La funzione **setUp()** si occupa di racchiudere il codice a comune dei test; in particolare a riga 8 il codice sovrascrive temporaneamente il flusso di input standard (System.in) con un nuovo flusso di input creato da **simulatedUserInput**. Questo serve per simulare l'input da parte del client durante il test. **simulatedUserInput** è una stringa che viene convertita in un array di byte e quindi utilizzata come flusso di input per la classe **gim**. In seguito viene chiamato **startMenu()**, dunque l'applicazione viene avviata e riceve l'input simulato.

Listing 9: ClientTest

```

1 @Test
2 public void TestEMakeReservationWithoutMoney() {
3     simulatedUserInput = "1\ntest1@email\npassword\n1\n2025-01-02\n1\n2\n3\n0\n7\
4         n3\n"; // input simulato per fare una prenotazione; il client non ha
5         soldi nel borsellino
6     setUp(); // funzione che contiene il codice a comune tra tutti i test e
7         lancia l'asserzione
8 }
9
10
11 public void setUp() {
12     System.setIn(new ByteArrayInputStream(simulatedUserInput.getBytes()));
13     gim = new GraphicInterfaceManager();
14     assertDoesNotThrow(() -> gim.startMenu());
15 }

```

A precedere la classe vi sono due annotazioni, **@DisableIfTestFails** e **@TestMethodOrder(MethodOrderer.MethodName.class)**; la prima fa sì che, appena un test fallisce, i test successivi non vengano neanche eseguiti; la seconda invece fa sì che i test vengano eseguiti in ordine alfabetico, condizione che ha permesso la corretta

esecuzione di test in successione, seguendo una logica per cui per far sì che alcuni test funzionino correttamente, è necessario eseguirne prima altri a loro funzionali. Per fare un esempio, per prima cosa viene eseguito il test di registrazione che crea un client che sarà il soggetto interessato alle operazioni fatte nei test successivi, senza il quale non potrebbero funzionare. Il metodo `tearDown()`, preceduto dall'annotazione `@AfterAll`, si occupa di rilasciare le risorse e quindi di eliminare il client creato per i test. (listing 10).

Listing 10: Test annotations

```

1  @DisableIfTestFails
2  @TestMethodOrder(MethodOrderer.MethodName.class)
3  public class ClientTest {
4
5      // ... test ...
6
7      @AfterAll
8      static void tearDown() {
9          ClientDaoImpl clientDao = new ClientDaoImpl();
10         clientDao.deleteTestClient("test1@email");
11     }
12 }
```

4.2. class SuperUserTest

Per quanto riguarda le operazioni eseguibili dal `SuperUser`, la classe fornisce dei test del tutto simili nelle modalità di implementazione nel caso del `Client`, valgono dunque tutte le considerazioni fatte pocanzi. Per fare un esempio pratico, il listing 11, mostra il test di modifica del prezzo di un tipo di campo. Notiamo che l'operazione viene fatta due volte, la prima per modificare il prezzo, seconda per reimpostarlo a quello originario.

Listing 11: testModifyPrice

```

1  @Test
2  public void TestDModifyPrice() {
3      String type = "clay";
4      float price = courtDao.getPrice(type);
5      simulatedUserInput = "1\na@b\nandre\nn3\nclay\nn25\nn9\nn3\nn";
6      assertion(); // fa l'asserzione e prepara l'input
7      simulatedUserInput = "1\na@b\nandre\nn3\nclay\nn" + price + "\n9\nn3\nn";
8      assertion();
9  }
```

4.3. Risultati dei test

Di seguito i risultati dei test effettuati con JUnit (figura 8).

Test in SWE_court_prenotation: 21 total, 21 passed

2 m 10 s

[Collapse](#) | [Expand](#)

ClientTest		1 m 47 s
TestARegistration()	passed	5.05 s
TestBRegistrationWithAlreadyUsedEmail()	passed	2.32 s
TestCRegistrationInvalidEmail()	passed	
TestDLogin()	passed	4.36 s
TestEMakeReservationWithoutMoney()	passed	9.76 s
TestFAddMoney()	passed	5.73 s
TestGMakeReservationStandard()	passed	7.30 s
TestHMakeReservationStandardRentingKit()	passed	8.75 s
TestIUpgradePremium()	passed	5.48 s
TestJMakeReservationPremium()	passed	7.50 s
TestKMakeReservationAlreadyBookedSlot()	passed	7.45 s
TestLMakeReservationPremiumRentingKit()	passed	10.63 s
TestMMakeReservationOnHoliday()	passed	2.62 s
TestNDeleteReservation()	passed	22.44 s
TestORenewPremium()	passed	7.61 s
SuperUserTest		23.40 s
TestALogin()	passed	987 ms
TestBInsertCourt()	passed	3.34 s
TestCDeleteCourt()	passed	2.83 s
TestDModifyPrice()	passed	7.36 s
TestEInsertTimeSlot()	passed	4.21 s
TestFDeleteTimeSlot()	passed	4.68 s

Generated by IntelliJ IDEA on 02/11/23, 17:26

Figura 8: Risultati dei test

5. Confronto con le soluzioni architetturali di Bing GPT

In questa sezione è presentato un esperimento che vede il confronto tra il nostro codice e quello generato da Bing GPT, a cui è stato chiesto di generare le classi del Domain Model e quelle necessarie ad implementare il pattern DAO. Di seguito le nostre domande ed il codice fornito dall'AI.

Utente

Potresti realizzare, in Java, le classi del DomainModel necessarie a realizzare un software in grado di gestire le prenotazioni di campi da tennis (in erba, terra, o cemento) o padel per un circolo sportivo? Gli utenti dovranno essere in grado di creare un proprio account personale, il quale include un portafoglio digitale che sarà utilizzato per le varie tipologie di transazioni. Gli account degli utenti si suddividono in standard e premium; per questi ultimi è previsto uno sconto su ogni prenotazione e un sistema di punteggio, il quale permette loro di usufruire di una prenotazione gratuita dopo un certo numero di prenotazioni effettuate. Il software dovrà anche tenere conto dei giorni di festività, in cui il circolo sarà chiuso, e vietare agli utenti di effettuare delle prenotazioni. Il circolo inoltre dà la possibilità di noleggiare 0 o più kit da gioco ad un prezzo aggiuntivo al noleggio del campo (non incluso). L'applicativo dovrà includere un sistema di notifica agli utenti tramite mail delle operazioni effettuate. Oltre agli utenti utilizzatori del servizio, il software dovrà anche essere in grado di gestire uno o più utenti speciali (superuser) in grado di effettuare modifiche amministrative, come visionare le prenotazioni ricevute, aggiungere o rimuovere campi, aggiungere o rimuovere slot orari. Nel caso in cui avvengano eliminazioni che incidano su delle prenotazioni ricevute, dovranno essere quindi annullate con emissione di relativo rimborso e invio di una mail di notifica verso l'utente.

Bing

Certainly! Here are the classes of the Domain Model that you can use to develop a software for managing tennis or padel court reservations for a sports club:

```
public class User {  
    private int id;  
    private String name;  
    private String email;  
    private String password;  
    private String accountType;  
    private double walletBalance;  
    private List<Reservation> bookingHistory;  
    // getters and setters  
}  
  
public class Court {  
    private int id;  
    private String courtType;  
    private boolean isAvailable;  
    private double pricePerHour;  
    private boolean equipmentAvailable;  
    // getters and setters  
}  
  
public class Reservation {  
    private int id;  
    private int userId;  
    private int courtId;  
    private LocalDateTime startTime;  
    private LocalDateTime endTime;  
    private boolean isCancelled;  
    private double refundAmount;  
    private List<String> equipmentList;
```

```

    // getters and setters
}

public class Holiday {
    private int id;
    private LocalDate holidayDate;
    private String description;
    // getters and setters
}

public class Superuser {
    private int id;
    private String name;
    private String email;
    private String password;
    // getters and setters
}

```

The User class represents the users of the system. It has attributes such as `id`, `name`, `email`, `password`, `accountType`, `walletBalance`, and `bookingHistory`. The `accountType` attribute can be either `standard` or `premium`. Premium users get a discount on every reservation and can earn points to avail a free reservation after a certain number of bookings.

The Court class represents the tennis or padel courts available for reservation. It has attributes such as `id`, `courtType`, `isAvailable`, `pricePerHour`, and `equipmentAvailable`. The `courtType` attribute can be either `tennis`, `padel`, `tennis in erba`, `tennis in terra`, or `tennis in cemento`. The `isAvailable` attribute is a boolean value that indicates whether the court is available for reservation or not. The `equipmentAvailable` attribute is a boolean value that indicates whether the equipment is available for rent or not.

The Reservation class represents a reservation made by a user. It has attributes such as `id`, `userId`, `courtId`, `startTime`, `endTime`, `isCancelled`, `refundAmount`, and `equipmentList`. The `startTime` and `endTime` attributes represent the start and end times of the reservation. The `isCancelled` attribute is a boolean value that indicates whether the reservation has been cancelled or not. The `refundAmount` attribute represents the amount to be refunded in case of cancellation. The `equipmentList` attribute is a list of equipment rented along with the reservation. If no equipment is rented, the list will be empty.

The Holiday class represents the holidays on which the club is closed. It has attributes such as `id`, `holidayDate`, and `description`.

The Superuser class represents the superusers of the system. It has attributes such as `id`, `name`, `email`, and `password`. Superusers have administrative privileges such as viewing reservations, adding or removing courts, and adding or removing time slots.

The system should also include a notification system that sends emails to users for various operations such as reservation confirmation, cancellation, and refund. The system should also prevent users from making reservations on holidays when the club is closed. In case of cancellations that affect received reservations, they should be cancelled with a refund and a notification email sent to the user.

Utente

Puoi ora mostrarmi le classi necessarie ad implementare il pattern Database access object con questo domain model ?

Bing

Certainly! Here are the classes of the Domain Model that you can use to develop a software for managing tennis or padel court reservations for a sports club with the DAO pattern:

```

public interface UserDao {
    public User getUserById(int id);
    public void addUser(User user);
    public void updateUser(User user);
    public void deleteUser(User user);
}

public interface CourtDao {
    public Court getCourtById(int id);
    public void addCourt(Court court);
}

```

```

    public void updateCourt(Court court);
    public void deleteCourt(Court court);
}

public interface ReservationDao {
    public Reservation getReservationById(int id);
    public void addReservation(Reservation reservation);
    public void updateReservation(Reservation reservation);
    public void deleteReservation(Reservation reservation);
}

public interface HolidayDao {
    public Holiday getHolidayById(int id);
    public void addHoliday(Holiday holiday);
    public void updateHoliday(Holiday holiday);
    public void deleteHoliday(Holiday holiday);
}

public interface SuperuserDao {
    public Superuser getSuperuserById(int id);
    public void addSuperuser(Superuser superuser);
    public void updateSuperuser(Superuser superuser);
    public void deleteSuperuser(Superuser superuser);
}

```

The UserDao interface provides methods for CRUD (Create, Read, Update, Delete) operations on the User class.

The CourtDao interface provides methods for CRUD operations on the Court class.

The ReservationDao interface provides methods for CRUD operations on the Reservation class.

The HolidayDao interface provides methods for CRUD operations on the Holiday class.

The SuperuserDao interface provides methods for CRUD operations on the Superuser class.

You can implement these interfaces using any database technology of your choice such as JDBC, Hibernate, or JPA.

I hope this helps! Let me know if you have any other questions.

Utente

Secondo te, conviene utilizzare una classe base Dao?

Bing

La creazione di una classe base Dao dipende dalle esigenze del progetto. Se si prevede di utilizzare lo stesso codice per la gestione delle operazioni CRUD su più classi, allora può essere utile creare una classe base Dao che implementi i metodi comuni a tutte le classi. In questo modo, si evita la duplicazione del codice e si semplifica la manutenzione del software. Tuttavia, se le classi hanno requisiti di gestione dei dati molto diversi tra loro, allora potrebbe essere meglio creare classi Dao separate per ciascuna classe. In ogni caso, è importante progettare il software in modo che sia facile da mantenere e scalare in futuro.

Spero di esserti stato utile! Fammi sapere se hai altre domande.

Dalla conversazione con Bing GPT sono sorte differenze nell'architettura, sia nel Domain Model che nel Dao.

1. Non ha ritenuto necessario utilizzare una classe base Person che racchiudesse attributi, getter e setter comuni a Client (User) ed a SuperUser. Inoltre non ha gestito la possibilità che un cliente sia premium, problema risolvibile con un attributo booleano `isPremium`.
2. Non è stata fatta la classe RentingKit, necessaria però a memorizzare prezzo e numero di attrezzature noleggiate per la prenotazione. Al contrario è stato aggiunto un attributo booleano `equipmentAvailable` in `Court`, che sembra segnalare la disponibilità o meno di un kit per un certo campo, funzione però non richiesta nella domanda. Inoltre è stato aggiunto a `Reservation` l'attributo `equipmentList`, una lista di stringhe.
3. Non è stata implementata la classe `Wallet`, visto invece come un attributo di `Client`.
4. E' stata aggiunta la classe `Holiday` per la gestione dei giorni festivi, mentre noi abbiamo utilizzato la libreria de.jollyday. Non ha pensato di consigliarmi nessuna libreria.
5. Non è stata implementata la classe `TimeSlot`, visto invece come un attributo di `Reservation`, utile però nell'ottica Dao.
6. Per quanto riguarda il Dao, non sapendo la complessità delle interazioni con il database, ha giustamente fornito i metodi basilari del pattern, anche se, per coerenza con ciò che ha generato, sarebbe stata utile una classe base Dao generica, che noi invece non abbiamo usato data la diversità delle operazioni tra i Dao concreti.

6. Librerie di terze parti

Di seguito è riportata la lista delle librerie di terze parti utilizzate.

JUnit - per gli unit test.

<https://junit.org/junit5/>

JavaMail - per l'invio delle mail.

<https://docs.oracle.com/javaee/7/api/javax/mail/package-summary.html>

JDBC - per il collegamento al database mySql.

<https://www.oracle.com/it/database/technologies/appdev/jdbc.html>

Jollyday - per il controllo dei giorni festivi.

<https://jollyday.sourceforge.net/>

PlantUML - per la generazione di diagramma di classe UML, poi adattato manualmente.

<https://plantuml.com/>