

# **Software Gestionale per Palestre: Implementazione e Sviluppo**

Riccardo Fantechi - Marco Paglicci

Ottobre - Gennaio 2024/25



**UNIVERSITÀ  
DEGLI STUDI  
FIRENZE**

**Dipartimento di Ingegneria Dell'Informazione  
Ingegneria Del Software**



**Github Repository**

# Indice

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Contesto e Motivazioni . . . . .	3
1.2	Obiettivi dello sviluppo . . . . .	3
1.3	Valore Aggiunto e Vantaggi del Software . . . . .	3
<b>2</b>	<b>Requisiti di progetto e utilizzo</b>	<b>4</b>
2.1	Use case . . . . .	4
2.2	Template . . . . .	8
2.3	Mockups . . . . .	21
<b>3</b>	<b>Progettazione e Implementazione Java</b>	<b>25</b>
3.1	Class Diagram UML . . . . .	25
3.2	Classi ed Interfacce . . . . .	28
3.3	Database . . . . .	30
3.4	Design Patterns . . . . .	31
3.5	Evoluzione e miglioramento della progettazione . . . . .	33
<b>4</b>	<b>Test</b>	<b>39</b>
4.1	Test di inserimento utente . . . . .	42
4.2	Test Iscrizione ad un corso . . . . .	43
4.3	Test di visualizzazione dei clienti associati ad un Personal trainer . . . . .	44
4.4	Test invio messaggio . . . . .	45
4.5	Test modifica delle informazioni . . . . .	46

# 1 Introduzione

## 1.1 Contesto e Motivazioni

Negli ultimi anni, la gestione delle attività sportive e dei centri fitness ha subito una notevole evoluzione, grazie all'introduzione di soluzioni digitali per la pianificazione e l'organizzazione delle risorse. Tuttavia, molti centri sportivi e palestre continuano a utilizzare sistemi obsoleti o processi manuali che comportano inefficienze nella gestione delle iscrizioni, delle prenotazioni dei corsi e del monitoraggio degli utenti. La motivazione di questa ricerca risiede nella necessità di sviluppare un software per la gestione della palestra che sia intuitivo, flessibile e capace di integrarsi facilmente con le esigenze operative quotidiane. L'obiettivo è quello di creare una soluzione che migliori l'organizzazione interna e permetta una gestione ottimale delle attività, includendo funzionalità come la registrazione degli utenti, la prenotazione dei corsi, la gestione dei pagamenti e la comunicazione tra utenti e staff.

## 1.2 Obiettivi dello sviluppo

L'obiettivo principale di questo progetto è lo sviluppo di un software gestionale per palestre che consenta di ottimizzare la gestione delle attività e migliorare l'interazione tra utenti e struttura. In particolare, il software si propone di raggiungere i seguenti obiettivi specifici:

- Obiettivo 1: Creare un ambiente *user-friendly* che permetta al personale della palestra di gestire facilmente le iscrizioni, le prenotazioni e il monitoraggio degli abbonamenti dei vari clienti.
- Obiettivo 2: Implementare un sistema di gestione dei corsi offerti dalla struttura, in modo da consentire agli utenti di prenotare o cancellare la loro prenotazione in modo autonomo tramite un'interfaccia.
- Obiettivo 3: Offrire un servizio di comunicazione tra lo staff della palestra e gli utenti in modo da poter offrire allenamenti privati o notifiche riguardanti lo stato del proprio abbonamento.
- Obiettivo 4: Facilitare, per i personal trainer, l'organizzazione di corsi o di allenamenti privati in base al numero di iscritti e agli orari disponibili.

## 1.3 Valore Aggiunto e Vantaggi del Software

Il software sviluppato con questo progetto ha il potenziale di apportare notevoli benefici operativi e strategici ai centri fitness. Grazie alla sua capacità di ottimizzare i processi di gestione e migliorare l'interazione con gli utenti, il sistema può contribuire a ridurre i tempi di amministrazione e migliorare la soddisfazione dei clienti. Inoltre, l'integrazione di funzionalità di analisi dati offre un

valore aggiunto, permettendo al management di prendere decisioni basate su informazioni concrete, incrementando così la competitività delle palestre sul mercato.

## 2 Requisiti di progetto e utilizzo

### 2.1 Use case

In questa sezione presentiamo i casi d'uso del software per la gestione della palestra, evidenziando le principali interazioni tra gli utenti, lo staff e il sistema. Ogni caso d'uso descrive le azioni che gli utilizzatori possono intraprendere, fornendo un quadro chiaro delle operazioni gestite dal software.

#### Utente

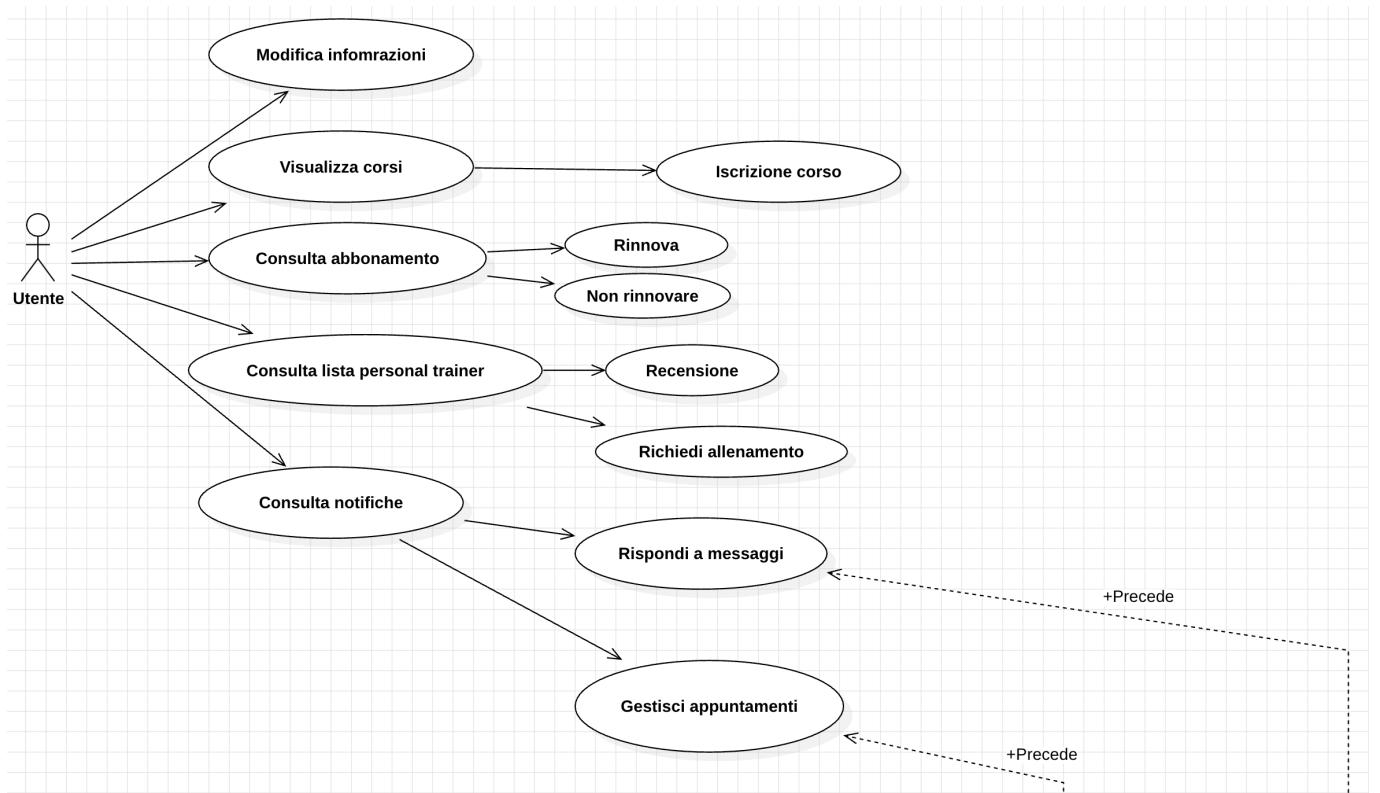


Figura 1: Use case Utente

Come possiamo notare, l'Utente può eseguire cinque azioni principali:

- **Modifica informazioni**
- **Visualizza corsi**
  - Iscrizione corso
- **Consulta abbonamento**

- Rinnova
- Non rinnovare

- **Consulta lista personal trainer**

- Recensione
- Richiedi allenamento

- **Consulta notifiche**

- Rispondi a messaggi
- Gestisci appuntamenti

Nell'opzione **"Modifica informazioni"**, stiamo dando la possibilità al cliente di modificare le informazioni anagrafiche o di pagamento.

Nella pagina **"Visualizza corsi"**, il cliente ha la possibilità di visualizzare una lista di corsi che la palestra mette a disposizione in diversi orari e stanze. L'utente può poi scegliere, selezionando un corso, se iscriversi ad esso.

La pagina **"Consulta abbonamento"** permette all'utente di visualizzare il suo piano di abbonamento alla palestra, insieme alle varie informazioni di scadenza e rinnovo. Per questo motivo, l'utente ha le opzioni di rinnovo o non rinnovo.

Nella pagina dei **"Personal trainer"**, l'utente può consultare una lista di coach che la palestra mette a disposizione, potendo valutarli e lasciare recensioni o richiedere un allenamento privato.

Nella **"Casella dei messaggi"**, l'utente può trovare le notifiche relative ai messaggi inviati dai personal trainer o notifiche di avvisi di novità. La pagina permette di rispondere ai messaggi e di gestire gli appuntamenti che i personal trainer propongono. Queste due opzioni hanno dei valori di *precedenza*, che si riferiscono a delle azioni che i personal trainer devono eseguire prima di poter accedere a questi utilizzi.

## Personal trainer

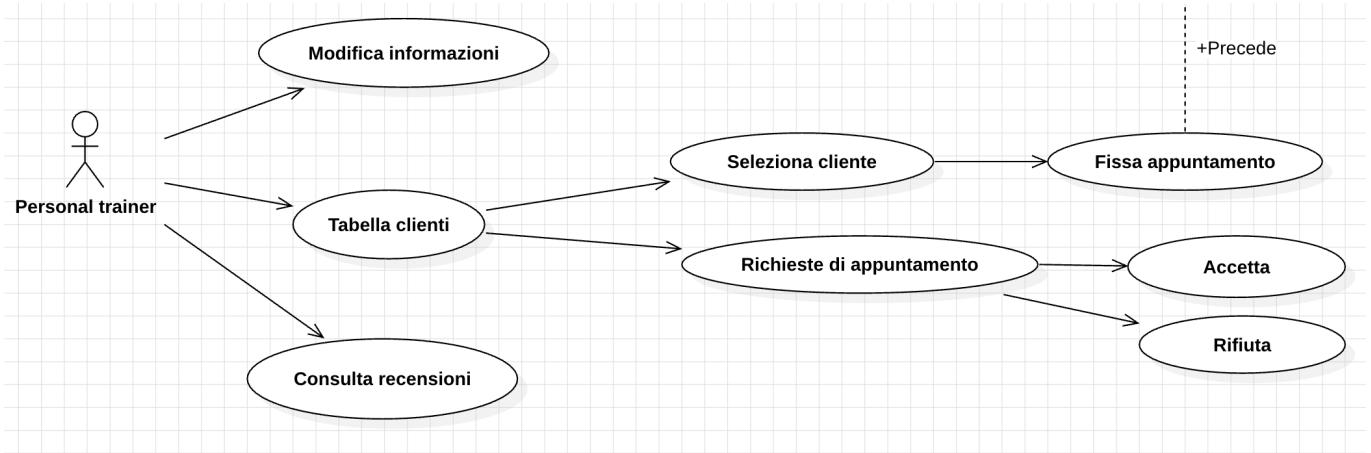


Figura 2: Use case Personal trainer

Il Personal trainer può eseguire tre azioni principali:

- **Modifica Informazioni**
- **Tabella clienti**
  - Seleziona cliente
  - Richieste appuntamenti
- **Consulta recensioni**

Oltre alle due possibili scelte che abbiamo già analizzato nel caso d'uso del cliente, notiamo un'opzione **Tabella clienti**. Da questa pagina il personal trainer può visualizzare tutti i clienti e può scegliere se fissare un appuntamento selezionando quest'ultimo. Nella stessa pagina è anche possibile visualizzare una casella di richieste di appuntamenti. Notiamo che l'attributo *precede* indica che un cliente può gestire un appuntamento solo se un personal trainer ne ha prima fissato uno.

## Personale

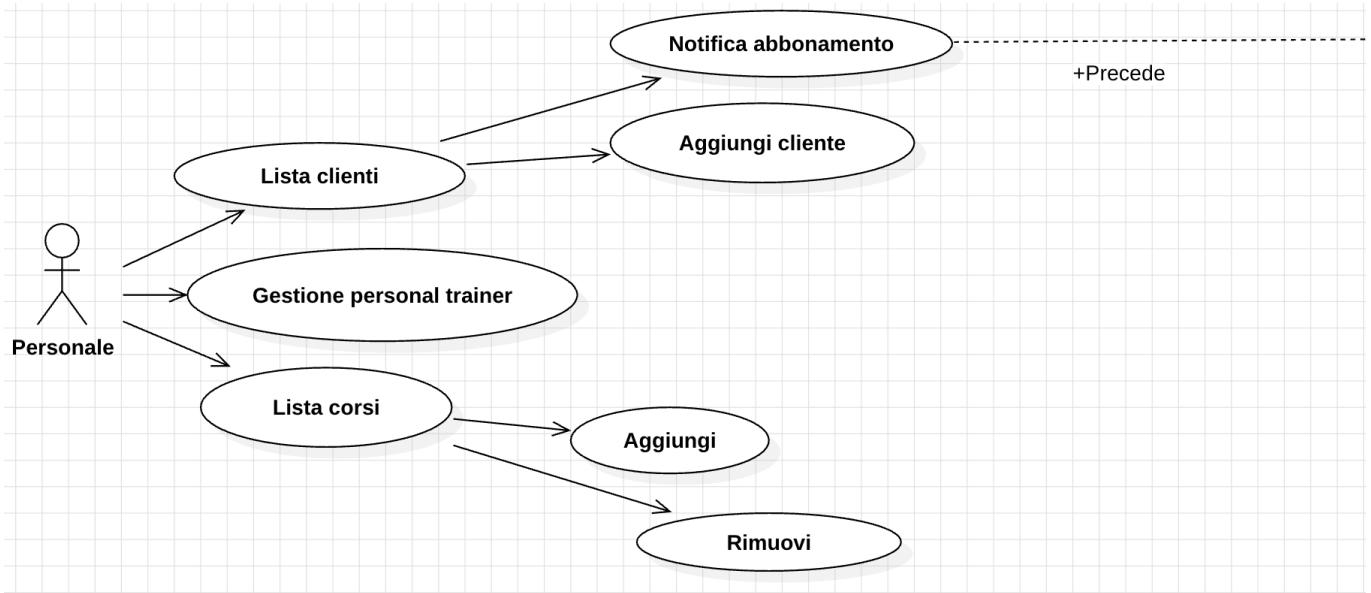


Figura 3: Use case Personale

Il Personale che lavora nella palestra ha tre possibili metodi per la gestione del sistema:

- **Lista clienti**
  - Notifica abbonamento
  - Aggiungi cliente
- **Gestione personal trainer**
- **Lista corsi**
  - Aggiungi
  - Rimuovi

Il personale può accedere alla pagina di tutti i **clienti iscritti** per aggiungerne di nuovi o per notificare dei messaggi ad un cliente specifico. Notiamo come l'attributo *precede* sta a indicare che un utente, prima di poter rispondere ad un messaggio, deve riceverne uno.

Nella pagina **Lista corsi** il personale può aggiungere o rimuovere dei corsi che la palestra offre, impostando orario e stanza del corso.

## 2.2 Template

Riportiamo di seguito i template costruiti sui vari casi d'uso appena discussi. Possiamo trovare tutte le informazioni necessarie per capire quali opzioni ogni ente ha a disposizione, con attributi di descrizione, pre e post condizioni. Ogni Template descrive più approfonditamente l'ambiente necessario per attuare una determinata azione, con un aggiunta di un classico svolgimento del metodo.

<b>Use Case 1</b>	Iscrizione a un corso
<b>Livello</b>	User Goal
<b>Descrizione</b>	L'utente decide di iscriversi ad un corso presente nella palestra
<b>Pre-condizioni</b>	L'utente deve avere l'accesso al software e le credenziali
<b>Attori</b>	Utente
<b>Post-condizioni</b>	Il corso viene aggiunto nella lista dei corsi frequentati dall'utente e l'utente viene registrato come partecipante a determinato corso
<b>Normale svolgimento</b>	L'utente accede all'applicazione e consulta la lista di corsi disponibili, ne sceglie uno con disponibilità e seleziona l'opzione "partecipa". Dopo l'accettazione si aggiornerà il database per far consultare all'utente l'orario del corso e per permettere al personale di visualizzare i partecipanti ad esso

Figura 4: Template Iscrizione corso

<b>Use Case 2</b>	Aggiungere un cliente
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il personale iscrive un nuovo cliente
<b>Pre-condizioni</b>	Nessuna
<b>Attori</b>	Personale
<b>Post-condizioni</b>	Il cliente è aggiunto al database e può accedere tramite le credenziali ai suoi servizi
<b>Normale svolgimento</b>	Il personale aggiunge le informazioni dell'utente necessarie e lo iscrive nella palestra, aggiunge le sue informazioni nel database ed attraverso esse dà la possibilità all'utente di sfruttare i servizi dell'applicazione
<b>Svolgimento alternativo</b>	Il personal trainer accede all'applicazione e visualizza la lista dei clienti che gli hanno inviato una richiesta, sceglie uno di essi e attraverso la funzione "aggiungi" aggiunge l'utente alla lista di clienti seguiti, successivamente esegue il normale svolgimento per l'appuntamento.

Figura 5: Template Aggiungere un cliente

<b>Use Case 3</b>	Fissare un appuntamento
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il Personal trainer fissa un appuntamento con un utente
<b>Pre-condizioni</b>	Il Personal trainer deve accettare la richiesta dell'utente per poterlo visualizzare nella lista dei clienti
<b>Attori</b>	Personal trainer
<b>Post-condizioni</b>	Viene aggiunto un elemento Appuntamento nel database con gli orari e le specifiche dell'appuntamento richiesto, l'utente riceverà una notifica nel suo spazio software
<b>Normale svolgimento</b>	Il personal trainer accede all'applicazione e visualizza la lista dei clienti che sta seguendo, sceglie uno di essi e attraverso la funzione "crea appuntamento" manda una notifica all'utente per l'appuntamento
<b>Svolgimento alternativo</b>	Il personal trainer accede all'applicazione e visualizza la lista dei clienti che gli hanno inviato una richiesta, sceglie uno di essi e attraverso la funzione "aggiungi" aggiunge l'utente alla lista di clienti seguiti, successivamente esegue il normale svolgimento per l'appuntamento.

Figura 6: Template Fissare appuntamento

<b>Use Case 4</b>	Modificare profilo
<b>Livello</b>	User Goal
<b>Descrizione</b>	L'utente modifica le sue informazioni personali
<b>Pre-condizioni</b>	L'utente deve essere registrato nel database
<b>Attori</b>	Utente
<b>Post-condizioni</b>	L'utente ha modificato le informazioni sul suo profilo
<b>Normale svolgimento</b>	L'utente accede alla sua area personale e sceglie se modificare alcuni dei suoi dati quali: data di nascita, nome, cognome, e metodo di pagamento; se si accede all'interfaccia di modifica dove può salvare i cambiamenti effettuati all'interno database.

Figura 7: Template Modificare profilo Utente

<b>Use Case 5</b>	Rinnovare – Non Rinnovare
<b>Livello</b>	User Goal
<b>Descrizione</b>	L'utente gestisce il suo abbonamento
<b>Pre-condizioni</b>	Il Personale ha inviato una notifica all'utente per la scadenza del suo abbonamento, l'utente ha visualizzato la notifica dalla sua area personale ed attraverso essa accede alla interfaccia di gestione abbonamento
<b>Attori</b>	Utente
<b>Post-condizioni</b>	L'abbonamento dell'utente viene rinnovato per un altro mese (o più a seconda delle scelte) o l'abbonamento viene terminato e l'utente viene rimosso dal database
<b>Normale svolgimento</b>	L'utente accede all'area gestione abbonamento dove può visualizzare informazioni su di esso e rinnovare o eliminare l'iscrizione con la palestra
<b>Svolgimento alternativo</b>	L'utente riceve una notifica sulla sua area notifiche, attraverso essa accede all'area gestione abbonamento dove può visualizzare informazioni su di esso e rinnovare o eliminare l'iscrizione con la palestra

Figura 8: Template Rinnovare-Non Rinnovare

<b>Use Case 6</b>	Recensione Personal trainer
<b>Livello</b>	User Goal
<b>Descrizione</b>	L'utente lascia una recensione al Personal Trainer
<b>Pre-condizioni</b>	Il Personal trainer deve essere registrato nel database, e l'utente deve essere un suo cliente
<b>Attori</b>	Utente
<b>Post-condizioni</b>	La recensione sarà una parte della media per la valutazione dei Personal Trainer
<b>Normale svolgimento</b>	L'utente accede alla lista dei Personal trainer, ne seleziona uno di cui è cliente e accede alla funzionalità "lascia recensione" dove fornirà un punteggio da 1 a 5 (rappresentato in stelline) che si unirà al resto delle recensioni per fare da media alla valutazione del Personal trainer

Figura 9: Template Recensione Personal Trainer

<b>Use Case 7</b>	Modificare profilo
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il Personal trainer modifica le sue informazioni personali
<b>Pre-condizioni</b>	Il Personal trainer deve essere registrato nel database
<b>Attori</b>	Personal trainer
<b>Post-condizioni</b>	Il Personal trainer ha modificato le informazioni
<b>Normale svolgimento</b>	Il Personal trainer accede alla sua area personale e sceglie se modificare alcuni dei suoi dati, se si accede all'interfaccia di modifica dove può salvare i cambiamenti effettuati nel database

Figura 10: Template Modificare profilo PT

<b>Use Case 8</b>	Consulta notifiche
<b>Livello</b>	User Goal
<b>Descrizione</b>	L'utente consulta le notifiche inviategli
<b>Pre-condizioni</b>	<u>L'Utente deve essere registrato nel database</u>
<b>Attori</b>	Utente
<b>Post-condizioni</b>	Possono variare a seconda dell'uso
<b>Normale svolgimento</b>	L'Utente accede alla sua area notifiche, qui possono essere presenti le notifiche degli appuntamenti fissati dai Personal trainer o una notifica per la scadenza dell'abbonamento tramite la quale l'utente accede alla sezione consulta abbonamento

Figura 11: Template Consulta notifiche

<b>Use Case 9</b>	Gestione richieste
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il Personal trainer gestisce le richieste inviategli
<b>Pre-condizioni</b>	Il Personal trainer deve essere registrato nel database e un cliente ha richiesto un allenamento al Personal trainer
<b>Attori</b>	Personal trainer
<b>Post-condizioni</b>	Il Personal trainer ha accettato o rifiutato la richiesta, se accettato l'utente viene aggiunto alla lista clienti
<b>Normale svolgimento</b>	Il Personal trainer accede alla sua area <u>personale</u> e attraverso di essa alle richieste inviategli dagli utenti che desiderano essere <u>seguiti</u> , il Personal trainer può accettare o rifiutare la richiesta di un cliente. Se viene accettato il cliente verrà aggiunto alla lista clienti del Personal trainer In ogni caso verrà mandata una notifica all'utente

Figura 12: Template Gestione richieste

<b>Use Case 10</b>	Consulta recensioni
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il Personal trainer visualizza le recensioni lasciate sul suo profilo
<b>Pre-condizioni</b>	Il Personal trainer deve essere registrato nel database
<b>Attori</b>	Personal trainer
<b>Post-condizioni</b>	Nessuna
<b>Normale svolgimento</b>	Il Personal trainer accede alla sua area personale e visualizza le recensioni lasciate dai clienti, vedendo il punteggio da esso assegnatogli e quante recensioni sono state lasciate, compreso la sua valutazione complessiva

Figura 13: Template Consulta recensioni

<b>Use Case 11</b>	Notifica abbonamento
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il personale invia una notifica
<b>Pre-condizioni</b>	L'utente deve essere registrato nel database, il suo abbonamento deve essere in scadenza
<b>Attori</b>	personale
<b>Post-condizioni</b>	L'utente riceverà una notifica
<b>Normale svolgimento</b>	Il personale accede alla lista di tutti gli utenti della palestra, da lì vede gli utenti con l'abbonamento in scadenza e gli invia una notifica per il rinnovo dell'iscrizione

Figura 14: Template Notifica abbonamento

<b>Use Case 12</b>	Gestione Personal-trainer
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il personale gestisce la lista dei Personal Trainer presenti
<b>Pre-condizioni</b>	nessuna
<b>Attori</b>	personale
<b>Post-condizioni</b>	un Personal Trainer viene aggiunto nel database o ne viene eliminato uno
<b>Normale svolgimento</b>	Il personale accede alla lista di tutti i personal trainer presenti in palestra, da qui si può decidere di aggiungere un nuovo personal trainer all'elenco o di rimuoverne uno già presente

Figura 15: Template Gestione Personal Trainer

<b>Use Case 13</b>	Gestione corsi
<b>Livello</b>	User Goal
<b>Descrizione</b>	Il personale gestisce la lista dei corsi presenti
<b>Pre-condizioni</b>	nessuna
<b>Attori</b>	personale
<b>Post-condizioni</b>	un corso viene aggiunto nel database o ne viene eliminato uno
<b>Normale svolgimento</b>	Il personale accede alla lista di tutti i corsi presenti in palestra, da qui si può decidere di aggiungere un nuovo corso all'elenco o di rimuoverne uno già presente

Figura 16: Template Gestione corsi

## 2.3 Mockups

Ti seguito sono riportati i mockups di una possibile implementazione del nostro software, visualizzabile tramite il sito web. Possiamo visualizzare le varie pagine in cui Clienti, Personal trainer e Impiegati possono navigare ed interagire tra di loro.

The mockup displays the main client interface. At the top right, there is a user profile picture of a man, a search bar with a magnifying glass icon, and a menu icon consisting of a checkmark and three horizontal lines. The top center features a welcome message: "Ciao Mario Rossi, Benvenuto!". On the left side, there is a sidebar with a back arrow icon. Below the welcome message, there are two sections: "Corsi Disponibili" (Available Courses) and "Personal Trainer".

**Corsi Disponibili**

Search bar: Cerca corsi...

**Yoga Basics**  
Schedule: Mon & Wed, 10-11 AM  
Instructor: Sarah Connor

**Mindful Meditation**  
Schedule: Tue & Thu, 2-3 PM  
Instructor: John Smith

**Zumba Fitness**  
Schedule: Fri, 1-3 PM  
Instructor: Emily Zhang

**Personal Trainer**

Search bar: Cerca personal trainer...

**John Doe**  
Specialty: Strength Training  
Rating: 4.8/5

**Jane Smith**  
Specialty: Yoga & Pilates  
Rating: 4.9/5

**Notifiche**

**New Class Alert**  
Join our new Yoga session starting tomorrow!  
2 hours ago

**Promotion Offer**  
Get 20% off on all Pilates classes this week.  
5 hours ago

**Class Cancellation**  
The Zumba class scheduled for tonight is canceled.  
1 day ago

Made with Visly

Figura 17: Pagina principale cliente

Questa è la opening page dell'utente, dove può visualizzare tutto in maniera semplice e veloce.

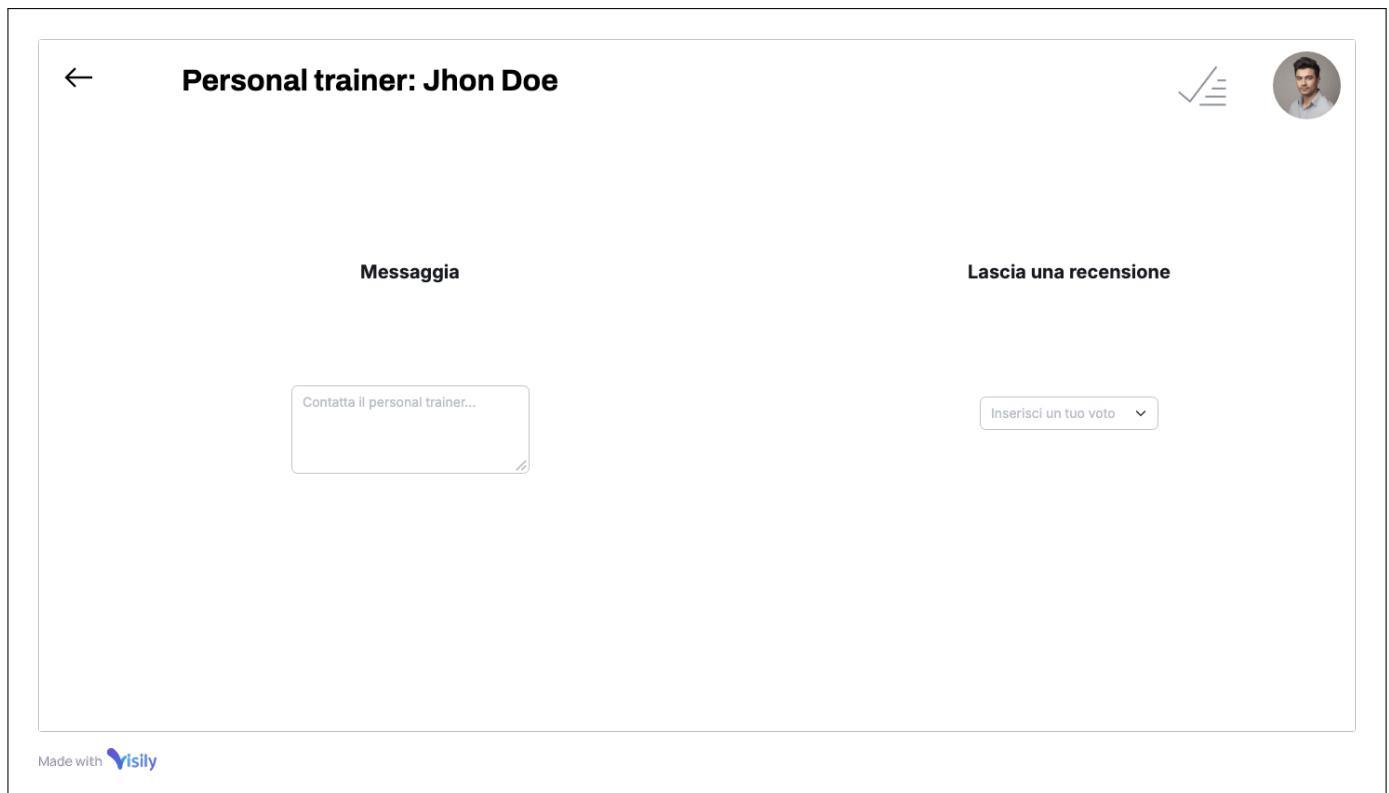


Figura 18: Selezione personal trainer

Troviamo la sezione dei personal trainer, una volta che l'utente seleziona un personal trainer, può comunicare con esso o lasciare una recensione.

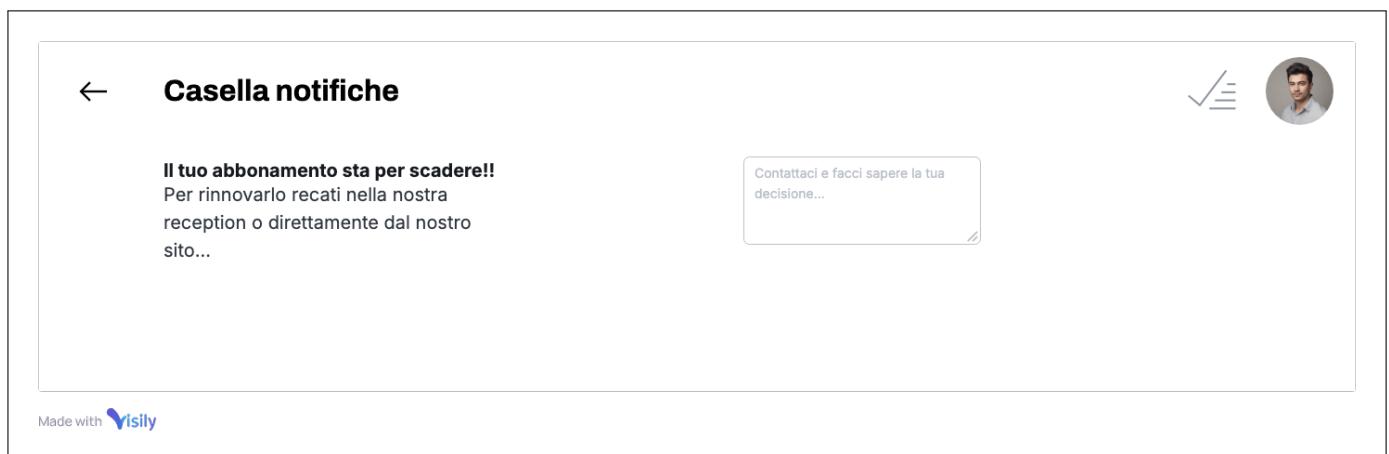


Figura 19: Casella notifiche del cliente

Questa è la casella delle notifiche dove ogni utente può ricevere aggiornamenti importanti, ricevendo al possibilità di rispondere a tali messaggi.

## ← Appuntamenti

**Corso di Zumba lunedì alle 16:30**  
Hai un appuntamento prenotato per lunedì alle 16:30 con Jhon Doe per il tuo corso di Zumba!!

Made with

Figura 20: Appuntamenti del cliente

## ← Informazioni personali

∅

Nome:  ∅

Cognome:  ∅

Tipo di abbonamento:  ∅

Metodo di pagamento:  ∅

Made with

Figura 21: Visualizzazione delle informazioni del cliente

## Impiegato: Mario Rossi

**Personal Trainer**

Scheda: Jhon Doe	

+ Aggiungi personal trainer

**Corsi**

Corso di Zumba	<span style="color: #ccc;">-</span>
Corso di Yoga	<span style="color: #ccc;">-</span>
Corso di Mindfulness	<span style="color: #ccc;">-</span>
Corso di Box	<span style="color: #ccc;">-</span>

+ Aggiungi corso

**Clienti**

Mario Rossi	<span style="color: #ccc;">Contatta</span>

+ Aggiungi cliente

Made with

Figura 22: Gestione software impiegato

Questa è la pagina che il lavoratore vede attraverso il software, da questa pagina può gestire i personal trainer, i clienti e i corsi, rimuovendone o aggiungendone di nuovi.

**Personal trainer: Mario Rossi**

**Recensioni**

- A Luigi Verdi ★★★★½

**Clienti**

- Mario Rossi Contatta
- Mario Rossi Contatta
- Mario Rossi Contatta
- Mario Rossi Contatta

**Conferma appuntamenti**

- Mario Rossi: Lunedì 16:30
- Mario Rossi: Giovedì 18:00
- Mario Rossi: Sabato 11:00

Made with 

Figura 23: Gestione software personal trainer

**Cliente: Mario Rossi**

Classes Attended: 15

Upcoming Classes: 5

Selezione la data ▼

Selezione l'orario ▼

Tipo di appuntamento... ▼

Invia 

Made with 

Figura 24: Fissaggio appuntamento da parte del personal trainer

Questo è ciò che il personal trainer visualizza tramite il software. Può consultare le recensioni ricevute, contattare dei clienti, per fissare appuntamenti, o gestire le richieste di appuntamenti dai suoi clienti. Successivamente vediamo come il personal trainer può mandare una richiesta di appuntamento ad un cliente selezionato, tramite una notifica.

### 3 Progettazione e Implementazione Java

Per la realizzazione del nostro progetto in linguaggio Java e per eseguire i test, abbiamo scelto di utilizzare le IDE IntelliJ IDEA e Visual Studio Code. Per facilitare la collaborazione tra i membri del gruppo, abbiamo adottato GitHub come strumento per il versionamento del codice. La fase di progettazione è stata supportata dalla consultazione di testi accademici e delle risorse offerte nel corso di Ingegneria del Software, permettendoci di identificare le strategie più appropriate per i nostri obiettivi. Successivamente, ci siamo riuniti per sviluppare il diagramma UML utilizzando StarUML, cercando di comprendere e modellare in modo efficace tutte le funzionalità previste dal sistema.

#### 3.1 Class Diagram UML

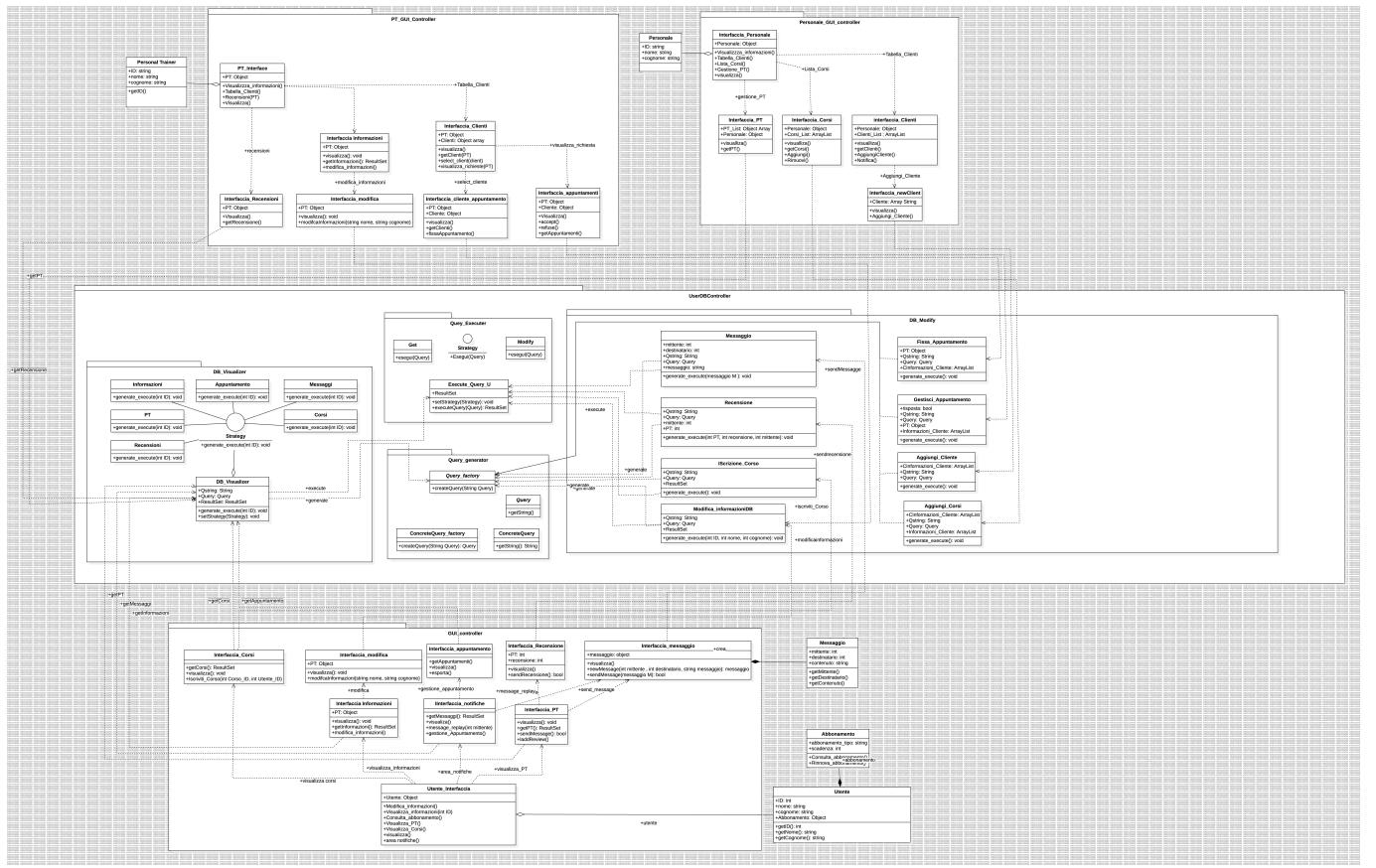


Figura 25: UML dell'intero progetto

Il progetto è diviso in package che rappresentano sia suddivisioni di responsabilità sia di logiche di comportamento e di accesso al sistema, i package sono di due tipi principali:

- DB Controller
- GUI Controller

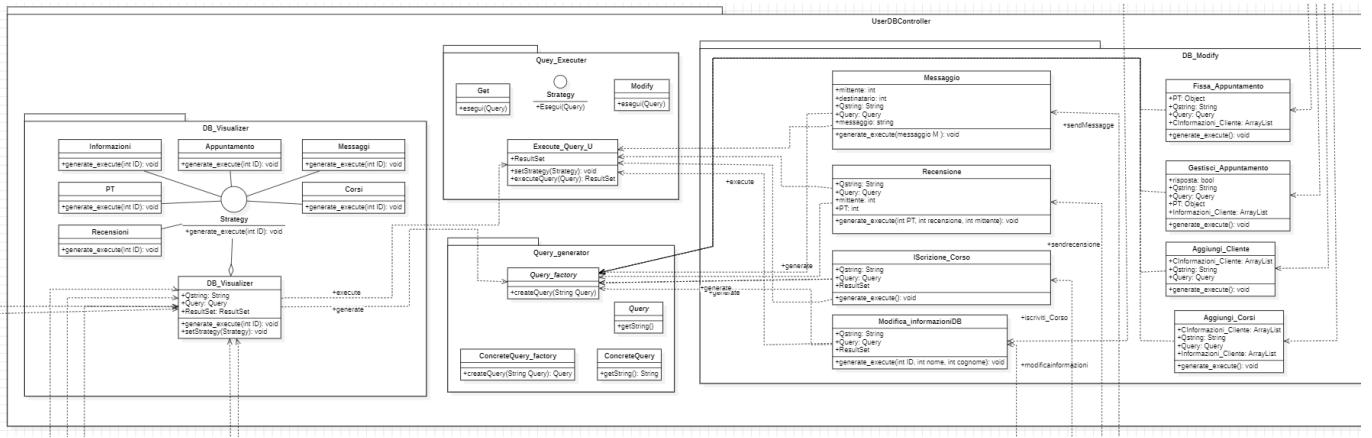


Figura 26: Database controller

Il package DB controller ha il compito di gestire l’accesso al Database, che rimane separato dalle altre classi e package, e offrire servizi di visualizzazione e modifica sul DB. Questi due ruoli sono divisi, all’interno del DB controller ci sono infatti 4 package che suddividono il ruolo di ogni operazione in compartimenti, la visualizzazione e la modifica dei contenuti è svolta in classi separate: questo perché all’interno dell’applicazione l’accesso alle informazioni, e quindi la loro visualizzazione, è eseguita costantemente in ogni interfaccia di uso dell’utente, mentre la modifica delle informazioni sul Database è un’operazione più complessa che viene eseguita separatamente.

Per comodità abbiamo anche separato il processo di creazione e di esecuzione di una Query SQL.

La *Query generator* viene gestita con la creazione di Query dedicate, wrapper che vengono generati in base alla necessità del cliente, utilizziamo questa pratica per incapsulare le informazioni di una query all’interno di un oggetto che può essere facilmente creato attraverso un factory. E successivamente inviati e eseguiti dall’*Execute Query*, l’unico vero responsabile di qualsiasi azione sul DB, che restituisce informazioni all’interfaccia che ha richiesto i servizi.

I controller GUI sono i package responsabili della gestione dell’applicazione dal lato utente. Si occupano di fornire interazione sia per i servizi che per il comparto grafico, visualizzando le informazioni e eseguendo le varie funzionalità a disposizione.

Ogni package rappresenta un’area dell’interfaccia diversa con cui gli utilizzatori del software interagiscono con il sistema. Ogni interfaccia ha funzionalità specifiche e permette di gestire le richieste separatamente. Questo approccio consente di trattare le varie interfacce utente come singole applicazioni che interagiscono con il Database Controller.

Abbiamo deciso di creare package di interfacce diverse in quanto ci siamo immaginati che l’applicazione abbia funzionalità diverse quindi necessità grafiche e pratiche diverse a seconda di quale entità la stia utilizzando

Ogni pagina ha delle funzionalità legate alla grafica e alla navigazione all'interno dell'applicazione, gestite dalla funzione visualizza(), oltre a una serie di funzionalità per la gestione e la modifica dei dati nel database. Le principali responsabilità di ciascun package GUI sono quelle di rendere l'utilizzo dei servizi intuitivo e user-friendly.

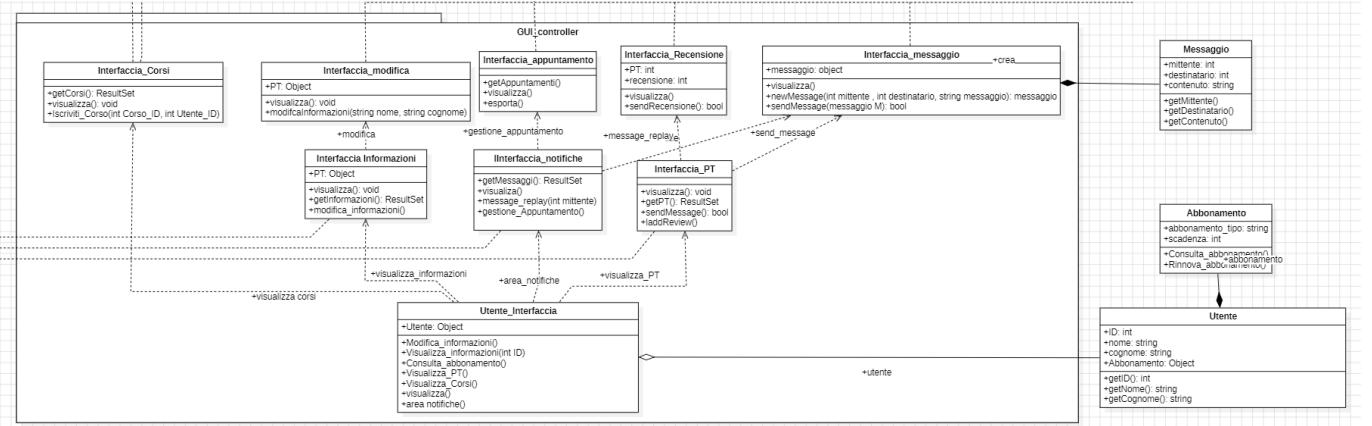


Figura 27: User GUI Controller

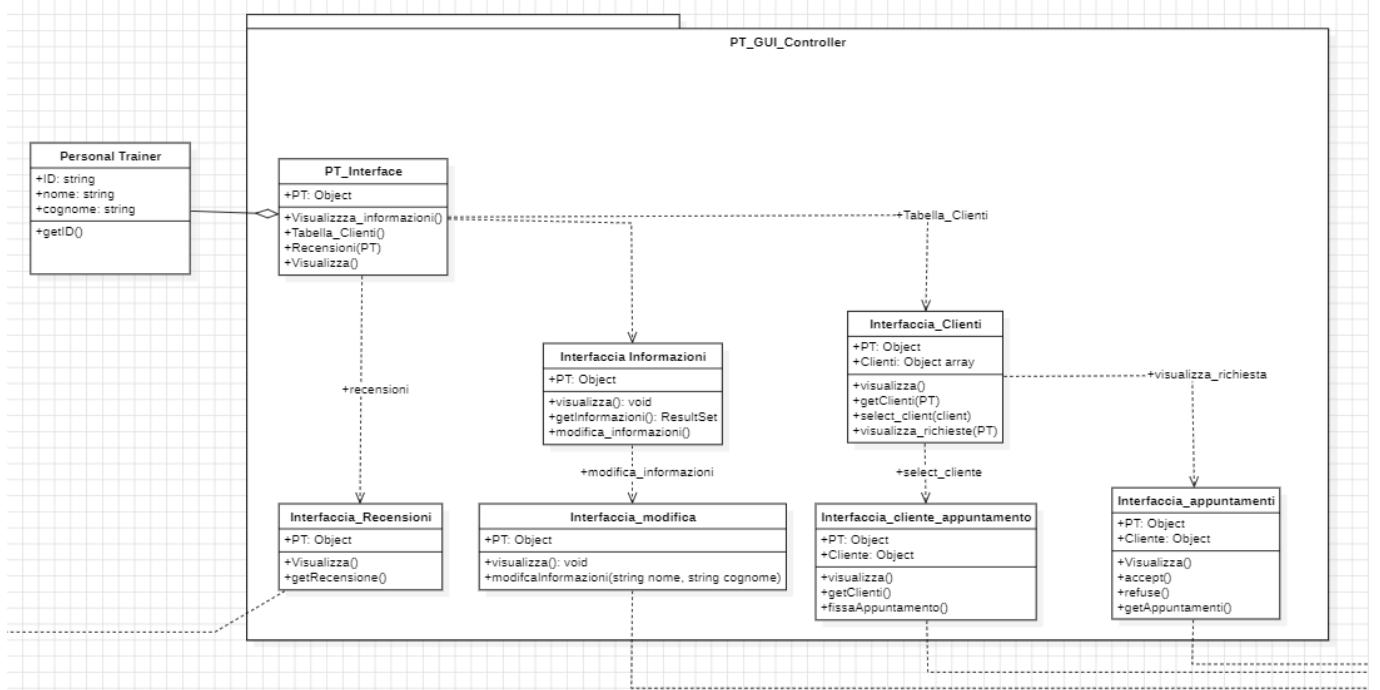


Figura 28: Personal Trainer GUI Controller

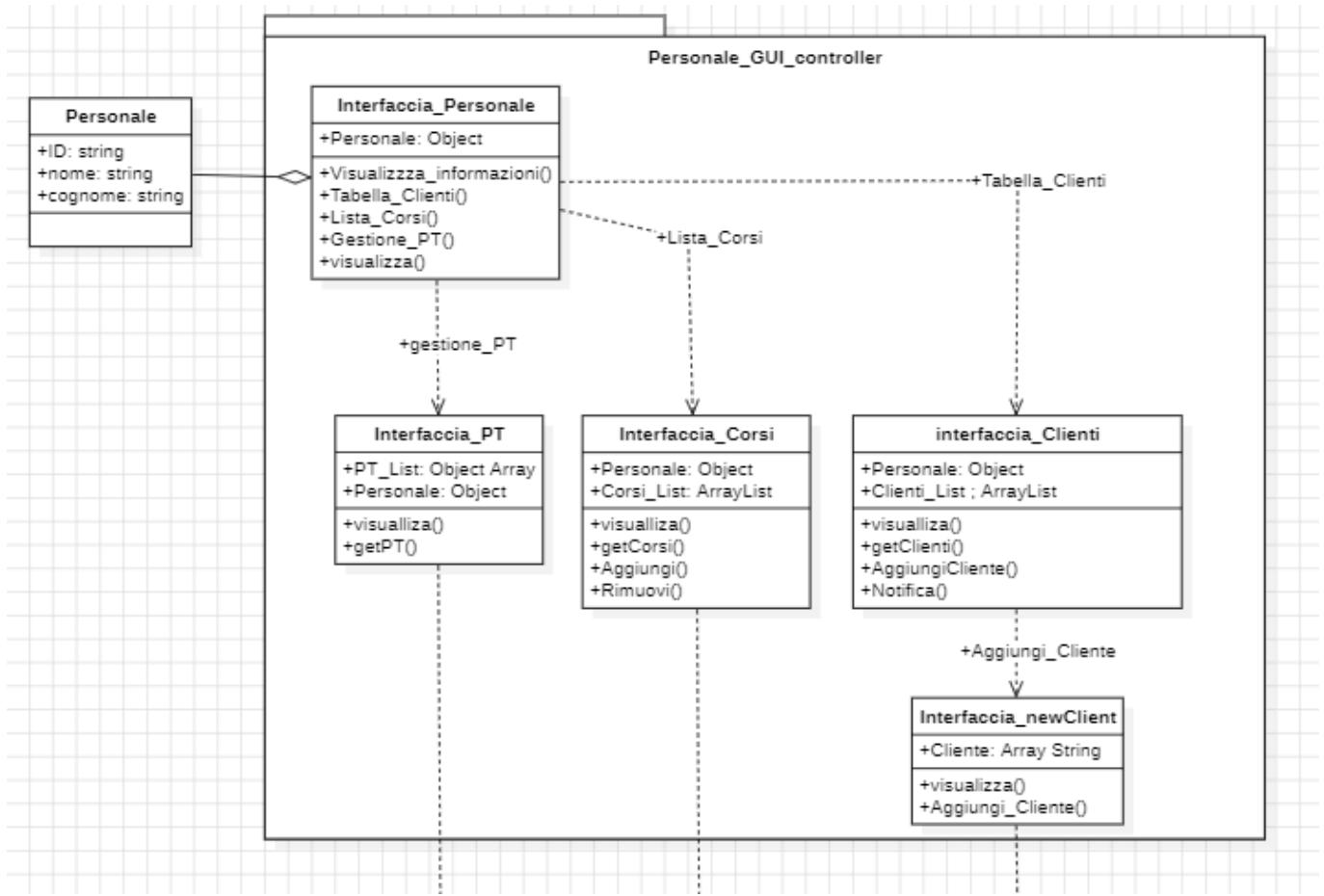


Figura 29: Personale GUI Controller

### 3.2 Classi ed Interfacce

Per lo sviluppo di questa applicazione ci siamo concentrati su un utilizzo di poche classi che definissero correttamente gli oggetti che avremmo trattato, e come gestire la connessione logica tra essi anche in visione di una implementazione in SQL.

Le classi principali definiscono gli utenti dell'applicazione e le loro informazioni per essere utilizzate nelle interfacce, come per il caso di Utente e Personal Trainer, altre classi definiscono modelli utili per lo scambio di informazioni, come la classe Messaggio e Abbonamento che definiscono due oggetti che incapsulano l'informazione e un'associazione con gli utenti della applicazione.

A seconda del package le classi si comportano in maniera diversa: le funzionalità principali del GUI Controller sono quelle di elaborare e visualizzare informazioni in maniera ordinata.

Il GUI Controller funziona attraverso una serie di interfacce, ognuna delle quali rappresenta logicamente una *pagina* dell'applicazione, dove quindi sono visualizzate informazioni e vengono messe a disposizione una serie di informazioni. Per ogni interfaccia abbiamo creato una classe in quanto sono strutture abbastanza complesse, responsabili non solo di rendere interattivo il software ma di intera-

gire con i servizi offerti dal Package del DB. In questo modo il cambio "pagina" rappresenta una vero e proprio cambio di responsabilità , e passa il controllo alla prossima interfaccia della applicazione. Prendiamo l'interfaccia di partenza della applicazione *Utente Interfaccia*, questa avrà al suo interno il codice necessario per visualizzare le informazioni, cioè la componente grafica e della GUI con il codice per ottenere le informazioni necessarie attraverso una l'utilizzo dell'apposito servizio di estrazione dati del Database. Inoltre ha una serie di funzioni che vengono richiamate da comandi del cliente per accedere ad altre , come una serie di pulsanti che ci permettono di passare da un interfaccia all'altra , o di modificare le informazioni *pagine* dell'applicazione.

Nel caso di *Utente Interfaccia* avremmo una window di visualizzazione che mostrerà il nome utente e alcune informazioni di base inoltre mette a disposizione una serie di pulsanti per navigare all'interno delle varie aree di gestione del software. Ogni funzione di *Utente Interfaccia* quindi si occupa della navigazione nelle varie pagine e fornisce il punto di partenza per le interazioni.

*Interfaccia Informazioni* oltre a contenere il codice che gestisce la propria GUI (funzione *visualizza()*) contiene una funzione per estrarre le informazioni necessarie attraverso *getInformazioni()* che va a richiamare i servizi offerti dal package DBController per estrarre le informazioni dell'Utente dal Database.

Generalmente questo è il funzionamento di tutte le interfacce di gestione della GUI nel package *GUI controller*: hanno una funzione *visualizza* che permette la gestione della grafica e una o più funzioni che interagiscono con il database e richiamano il DB Controller.

Questo avviene con due modalità: estrazione dei dati già presenti nel Database e visualizzazione nell'interfaccia o per apportare modifiche al database in base alle modifiche fatte dall'Utente

L'*Interfaccia Corsi* ha la funzione *visualizza()* che si occupa della parte grafica, la funzione *getCorsi()*, che viene gestita dal DB controller, si occupa di restituire le informazioni richieste (in questo caso ottenere la lista dei corsi disponibili) e la funzione *iscriviti()* va ad associare l'utente al corso che ha selezionato, invocando il *DB Controller* che modificherà il Database.

L'esecuzione di questa Iscrizione quindi è responsabilità del *DB Controller* più precisamente del *DB Modify* insieme alla classe *Iscrizione Corso*.

Alla classe *Iscrizione Corso* vengono passate le informazioni sull'utente e sul corso dall'interfaccia che lo invoca: *Iscrizione Corso* esegue generate *execute()* dopo aver invocato la Query Factory per creare una Query di suo interesse relativa alle necessità dell'interfaccia. Dato che tutte le operazioni di iscrizione ad un corso, dal punto di vista del DB, possono essere gestite da Query simili, *Iscrizione Corso* deve inserire le informazioni che ha ottenuto in un prompt comune, per poi generare una Query in cui inserire il prompt e passare la Query al *Query Executer* che si occuperà di eseguirla e,

nel caso ve ne sia uno, di restituire il risultato.

Proprio perché le operazioni sul DB vengono eseguite con lo scopo di ottenere informazioni o di modificarle, queste due richieste vengono gestite diversamente dall'*Executer*, che salverà il risultato delle operazioni all'interno dell'attributo *ResultSet* solo quando questo è richiesto (se si tratta di modiche, restituirà un valore booleano).

I ResulSet sono degli oggetti forniti dall'API JDBC, di cui parleremo successivamente, che consentono di lavorare con i risultati di una query SQL eseguita su un database. Un ResultSet rappresenta una tabella di dati che può essere navigata riga per riga, con dei metodi di navigazione per l'estrazione dei dati. Per accedere ai dati utilizziamo funzione come *getString(colonna)*, *getInt(colonna)*, dove colonna può essere sia il nome della tabella sia l'indice della colonna che vogliamo estrarre. Infatti, per estrarre i dati utili dal DB, utilizzeremo questi metodi, in modo da poter rendere utili i dati all'utente che gli ha richiesti.

### 3.3 Database

Nel progetto abbiamo utilizzato un database con l'obbiettivo di mantenere tutti i dati e risorse necessarie per la corretta esecuzione del software. Per separare le responsabilità e facilitare la manutenzione del codice abbiamo utilizzato il design pattern **DAO (Data Acces Object)** che consiste nell'isolare la logica di accesso ai dati da tutto il resto dell'applicazione. Lo scopo del DAO è fare da tramite tra le richieste dell'interfacce e l'esecuzione della query , trasformando i risultati di tipo ResultSet in oggetti coerenti con le entità del nostro progetto , che essi siano oggetti singoli o liste di oggetti. Con questo approccio centralizziamo la logica necessaria per l'accesso dei diversi dati e riduciamo la complessità del codice per le altre classi. Le interfacce infatti non dovranno scomporre o interpretare il risultato di una Query per ottenere le informazioni necessarie, ma riceveranno invece direttamente oggetti che contengo le informazioni delle entità , facilmente trattabili e che oscurano alle interfacce il funzionamento del database. Abbiamo utilizzato il **JDBC (Java Database Connectivity)** che ci permette di interagire con il database da un'applicazione Java. JDBC ci permette di stabilire la connessione il database per eseguire le operazione necessarie. Abbiamo potuto gestire le operazioni di accesso ai dati direttamente, garantendo l'efficienza delle query SQL e il totale controllo sulle operazioni sul database.

Vediamo ora la struttura del database tramite lo schema Entity Relationship:

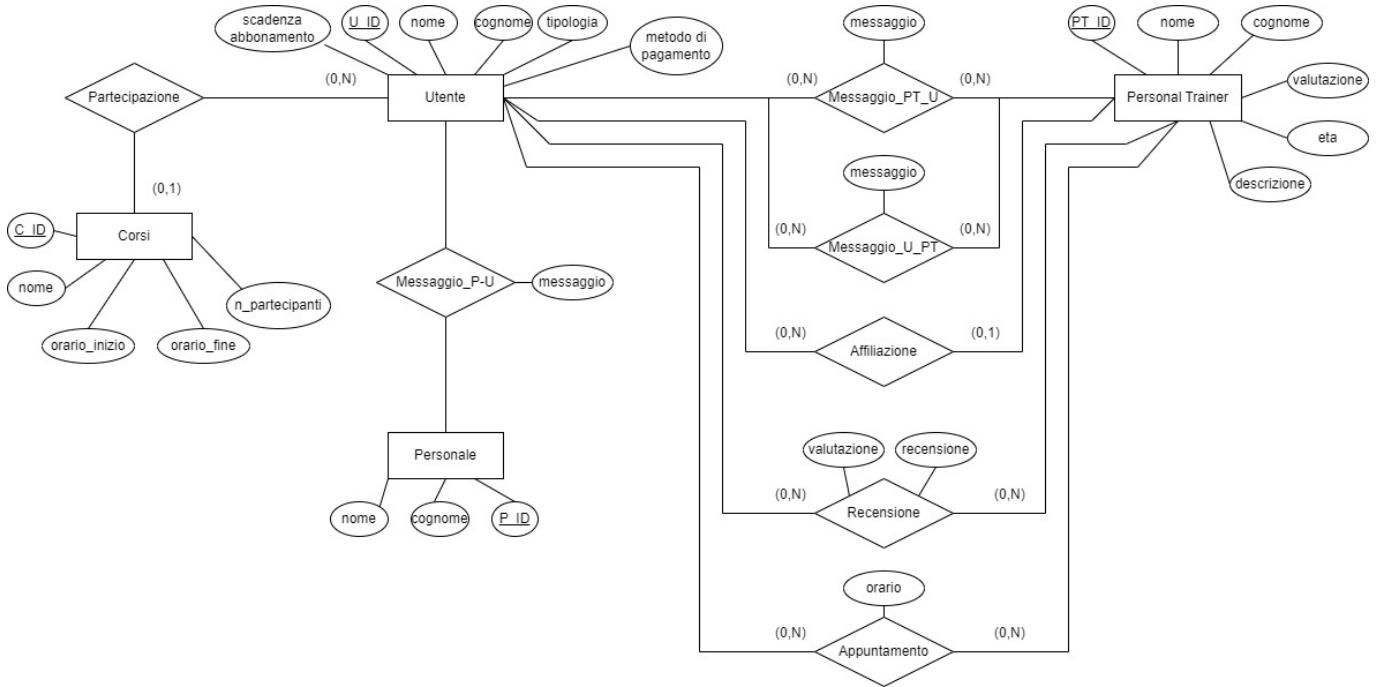


Figura 30: Modello ER del database implementato

Come possiamo notare il database gestisce le informazioni relative agli utenti, ai corsi, ai personal trainer, alle recensioni, agli appuntamenti, e a molte altre entità necessarie per il funzionamento della palestra Figura 30. Ogni tabella è progettata per riflettere le entità del sistema, con relazioni ben definite tra di loro. Per esempio, un utente può partecipare a più corsi, un personal trainer può essere associato a più utenti, e le recensioni sono legate sia a utenti che personal trainer. Notiamo ancora meglio l'utilizzo del design pattern DAO che ci permette di separare la logica di accesso ai dati dalla logica di business, consentendo un'interazione diretta con il database solo attraverso gli oggetti DAO per l'estrazione dei dati.

### 3.4 Design Patterns

Durante la progettazione abbiamo cercato di utilizzare Design Patterns che potessero migliorare e ottimizzare il codice, con anche l'obiettivo di sperimentare le varie soluzioni proposte da Design Patterns differenti.

Una volta definiti i casi d'uso e studiata una possibile implementazione del codice , abbiamo svolto una ricerca sui Design più efficienti e comuni , utilizzabile nel nostro caso , abbiamo individuato molti design pattern interessanti , alcuni però erano superflui e non necessari per la realizzazione , abbiamo quindi ridotto l'utilizzo ai design che permettevano un effettivo miglioramento della struttura del software.

La prima scelta è stata quella di separare le responsabilità attraverso i package e di usarli come compartimenti stagni che scambiano informazioni tra di loro.

Abbiamo scelto di implementare il Design Pattern **Strategy** per gestire il prelievo di dati dal database da parte di diverse classi. Questa scelta ci ha permesso di ottenere un alto grado di modularità e flessibilità, consentendoci di selezionare dinamicamente l'algoritmo di prelievo più adatto in base alle esigenze specifiche.

L'utilizzo dello Strategy ci ha permesso di evitare la proliferazione di funzioni specializzate che gestiscono vari algoritmi di prelievo a seconda dell'interfaccia o della classe richiedente. Invece, è sufficiente che la classe client imposti la strategia appropriata tramite un semplice metodo di *set*, delegando l'esecuzione dell'algoritmo alla strategia scelta in fase di runtime.

Questo pattern si è rivelato particolarmente utile nel nostro caso, poiché ci consente di mantenere il comportamento variabile, separato dalla logica di interesse della classe principale, così facendo, abbiamo migliorato la manutenibilità e la leggibilità del codice, evitando la creazione di numerosi blocchi condizionali (*if-else o switch*) che avrebbero complicato la gestione delle svariate condizioni. Un'altra implementazione dei Design Patterns è stata quella di una **Query Factory**.

Per automatizzare la modifica delle informazioni nel database da parte delle classi, abbiamo scelto di trattare le Query non come semplici stringhe, ma come oggetti che le incapsulano pronti per essere eseguiti. Questi oggetti vengono creati dinamicamente dalle classi in base alle necessità specifiche e poi passati alla parte esecutiva che si occupa dell'interazione con il database.

Per migliorare questo processo di creazione delle Query, soprattutto per quelle classi che si occupano di modificare i dati nel database, abbiamo implementato una *Query Factory*. Questa Factory consente di generare Query personalizzate in base a condizioni specifiche, adattandosi alle esigenze dei dati e alle operazioni da eseguire tramite il package *DB MODIFY* (il modulo dedicato alla modifica dei dati nel database).

Riconosciamo che l'introduzione della Factory aumenta il numero di classi e la complessità del sistema, tuttavia, il suo utilizzo semplifica la creazione degli oggetti Query, spostando la responsabilità della loro generazione fuori dalle singole classi. Ciò consente una maggiore modularità, manutenibilità e flessibilità nel progetto.

L'introduzione di uno Strategy per l'esecuzione delle Query potrebbe sembrare ridondante, dato che ci sono solo due strategie distinte, una per l'esecuzione di Query di modifica e una per l'esecuzione di Query di visualizzazione. Tuttavia, abbiamo scelto di implementarlo ugualmente per diverse ragioni, innanzitutto per una questione di coerenza logica e chiarezza nella separazione delle responsabilità e in secondo luogo perché le modalità di esecuzione delle due tipologie di Query differiscono

significativamente l'una dall'altra.

Le Query di modifica del database, ad esempio, non restituiscono alcun valore ma richiedono una conferma dell'esito dell'operazione. Viceversa, le Query di visualizzazione si aspettano un *ResultSet* contenente i dati risultanti dall'operazione *SELECT* sul database, che devono essere successivamente rielaborati e gestiti. L'adozione dello Strategy in questo contesto ci permette di gestire in modo più modulare e flessibile le differenze operative.

Un design importantissimo e necessario è stato quello della struttura DAO per l'incapsulamento delle informazioni, che ha aggiunto robustezza al codice e separazione delle responsabilità. Inoltre durante lo sviluppo del software abbiamo implementato il design di try-with-resources .Il costrutto try-with-resources in Java è un'implementazione del design pattern RAII (Resource Acquisition Is Initialization), che si basa sull'idea di acquisire e rilasciare risorse automaticamente nel loro ciclo di vita. È una soluzione pratica al problema della gestione delle risorse , nel nostro caso per l'accesso al Database.

### 3.5 Evoluzione e miglioria della progettazione

Durante la progettazione iniziale del software, abbiamo sviluppato una serie di idee basate sulle specifiche e sulla comprensione dei requisiti della palestra e ai suoi vari casi d'uso. Ciò ci ha portato a creare un modello UML iniziale che rappresentava le principali entità, i loro attributi e le interazioni tra di esse. Tuttavia, durante lo sviluppo effettivo, abbiamo incontrato situazioni in cui alcune scelte fatte durante la progettazione iniziale si sono rivelate migliorabili o necessitavano di modifiche per rendere il codice più leggibile o meglio organizzato. Di seguito riportiamo gli UML dei package e delle classi che hanno subito le maggiori modifiche:

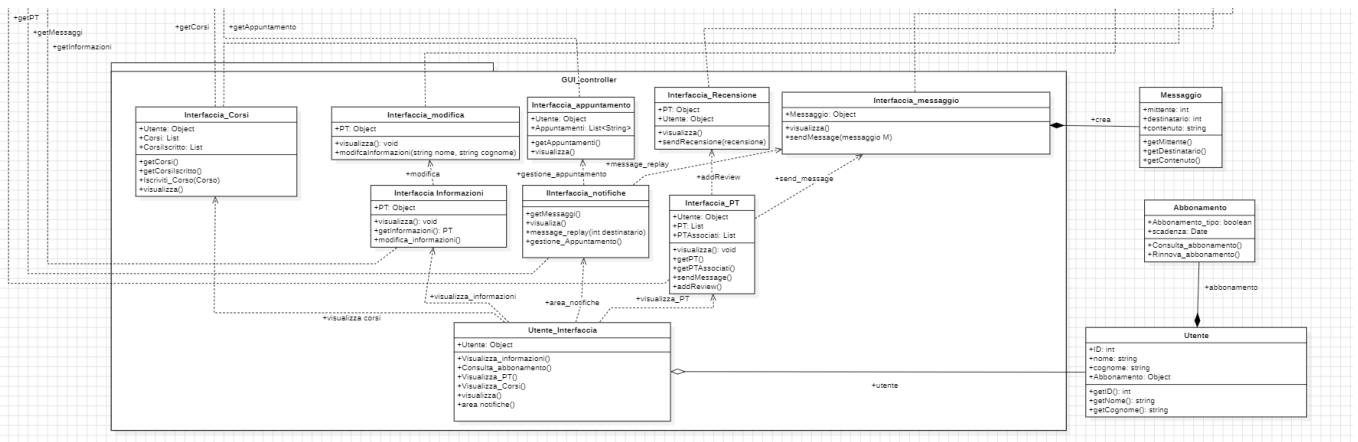


Figura 31: Client GUI controller modificato

Come possiamo notare le modifiche apportate al GUI controller non sono molte, principalmente

abbiamo sistemato tutti i tipi di ritorno delle varie funzioni e sistemato i parametri presi in ingresso. Notiamo che non sono presenti *ResultSet*, infatti abbiamo deciso di implementare un Design pattern DAO, in modo da dividere la logica dell'estrazione dati dall'utilizzo di quest'ultimi.

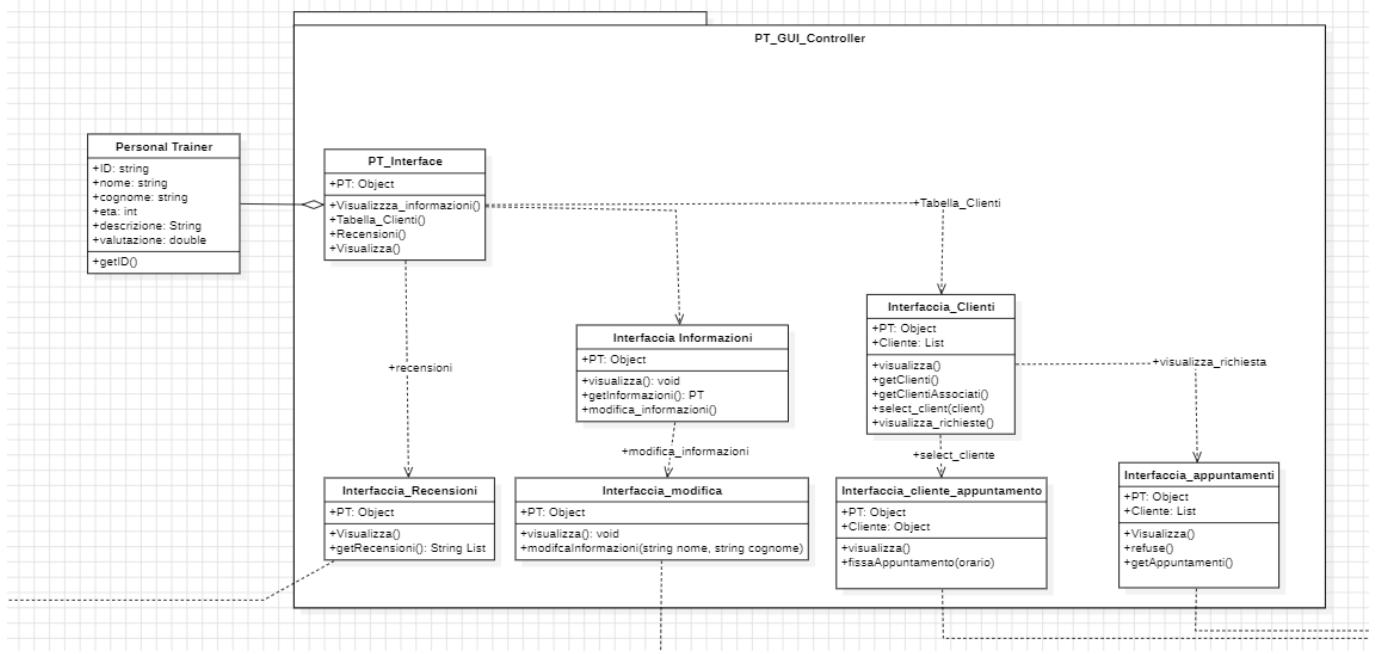


Figura 32: PT GUI controller modificato

Stesso cosa possiamo notarla per il PT\_GUI controller, non abbiamo dovuto modificare eccessivamente la classi o le loro funzionalità , ma abbiamo aggiunti alcuni attributi all'entità personal trainer in modo da avere una classe più completa. Oltre che sistemare l'input e il return di varie funzioni.

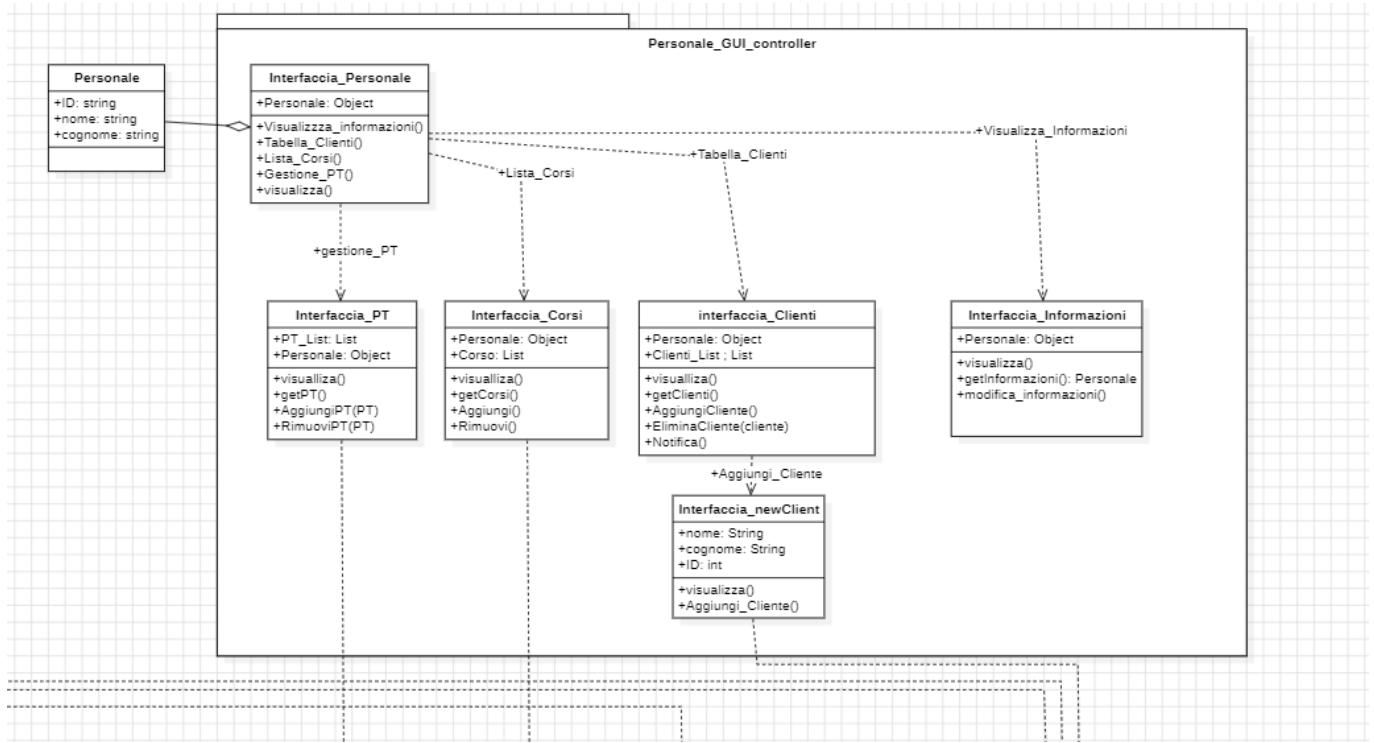


Figura 33: Personale GUI controller modificato

Per quanto riguarda il package dell’impiegato, oltre alle solite modifiche di cui abbiamo già parlato, abbiamo aggiunto una nuova classe *Interfaccia\_Informazioni*, in modo che anche un impiegato possa visualizzare e modificare le sue informazioni. Non abbiamo inserito una classe, modifica informazioni, per non appesantire il codice, visto che non è necessario che l’impiegato visualizzi una pagina diversa.

Vista l’implementazione del DAO, in ogni funzione getter che comunica con il database troviamo come tipo di ritorno una lista di oggetti di utile interesse. Abbiamo utilizzato la struttura dati lista perché è la più rapida ed efficiente, ci permetteva di restituire una lista di oggetti e poi effettuare un cast sulla classe del determinato oggetto, anche nel caso di iterazione su di essa ed accesso ai singoli elementi che contiene.

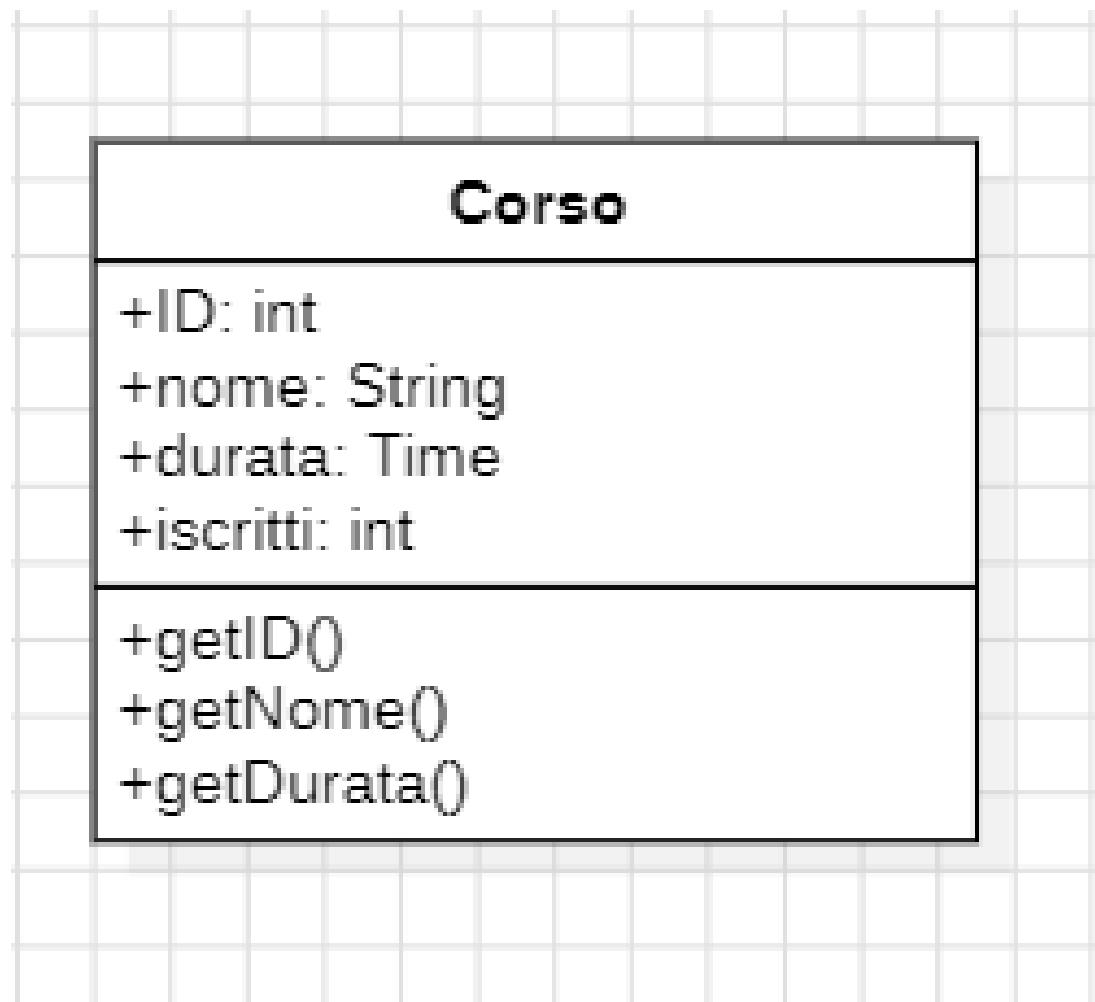


Figura 34: Nuova entità corso

Per facilitare alcune operazioni e per rendere il codice più comprensibile, abbiamo deciso di introdurre la classe corso, in modo da poter gestire delle liste di questo oggetto, quando il cliente ne richiede la visualizzazione.

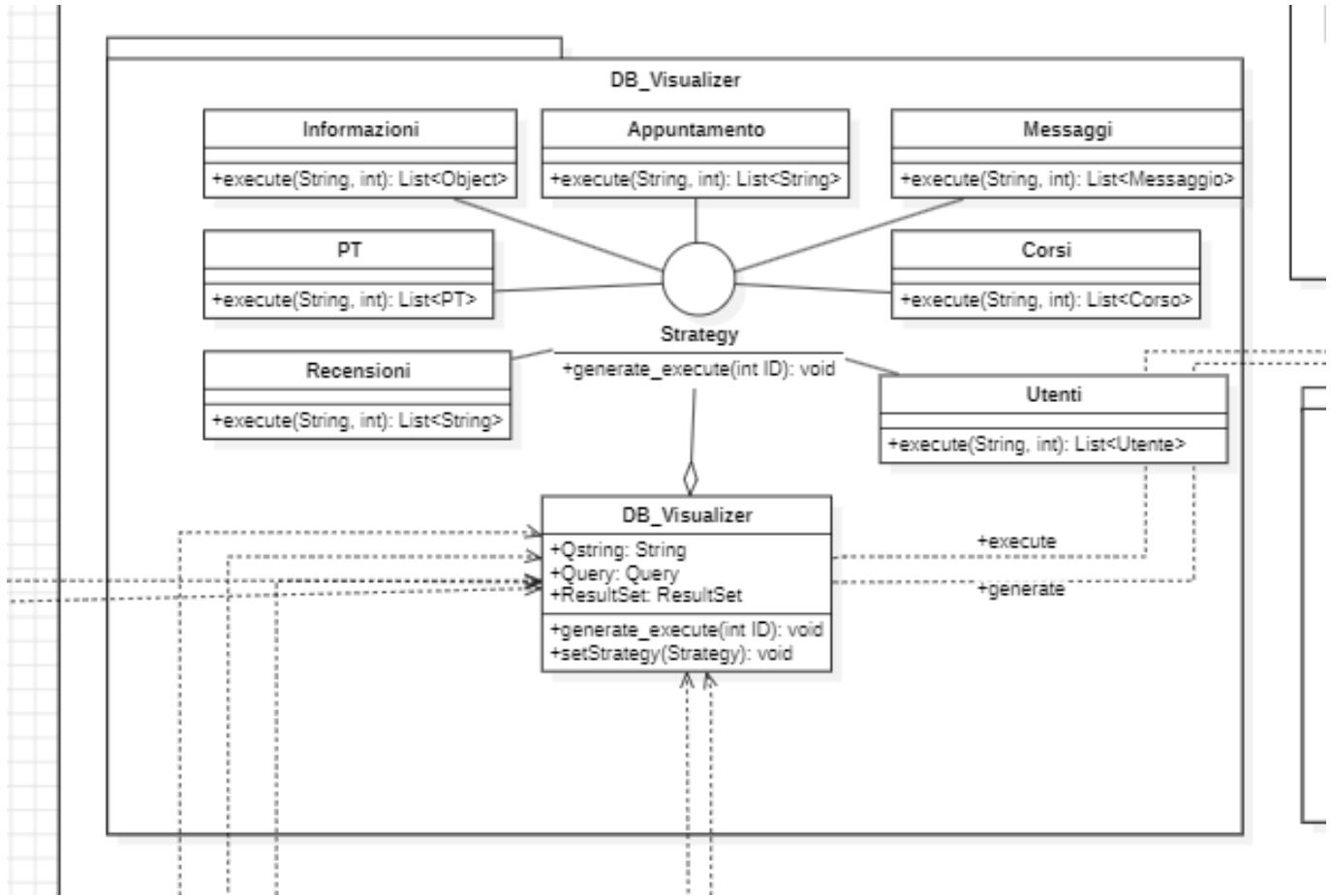


Figura 35: DB visualizer modificato

Passiamo ora alla sezione di modifiche che abbiamo apportato alla logica del database. Notiamo come è stata aggiunta un'altra strategia al design pattern strategy di cui abbiamo precedentemente discusso. Si tratta di una classe che gestisce tutte le richieste di visualizzazione di utenti, per cui potrà essere utilizzata sia dagli impiegati che dai personal trainer in modo diverso a seconda di chi invoca lo strategy.

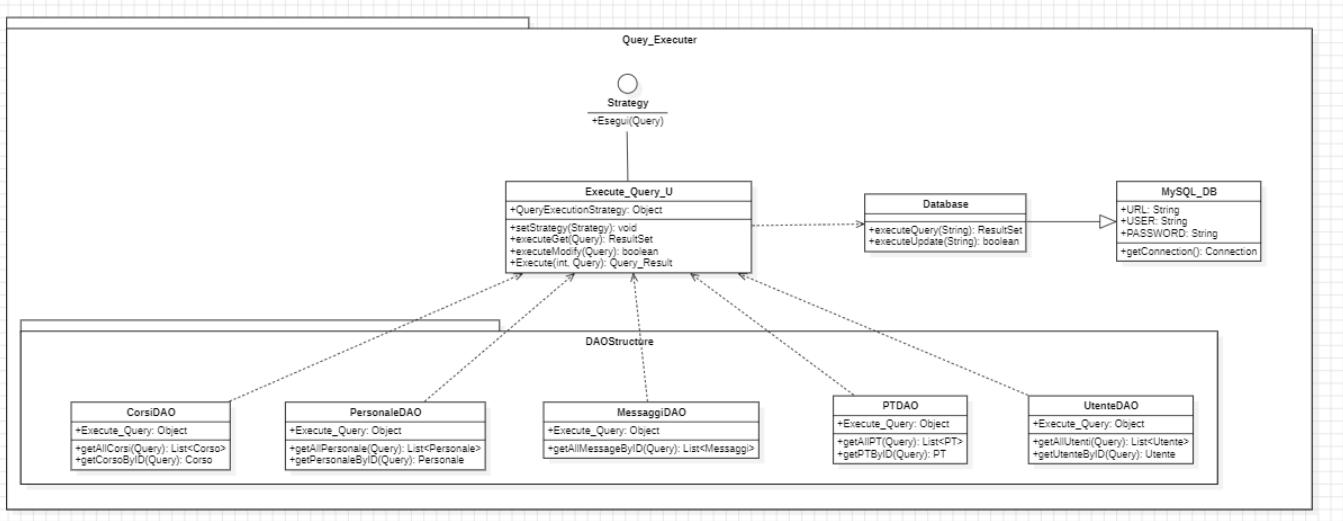


Figura 36: Query Executer modificato

Parliamo della modifica maggiore che abbiamo apportato al software. Abbiamo modificato la classe di esecuzione delle Query implementando in design pattern DAO. In questo modo siamo riusciti a dividere l'estrazione dei dati, dall'utilizzo di essi. Infatti, come possiamo notare, sono presenti delle classi DAO per ogni entità, in modo che sia delegato a loro l'estrazione dei dati dal database. Le funzioni che mettono a disposizione verranno poi usate dalle varie GUI per la gestione dei dati.

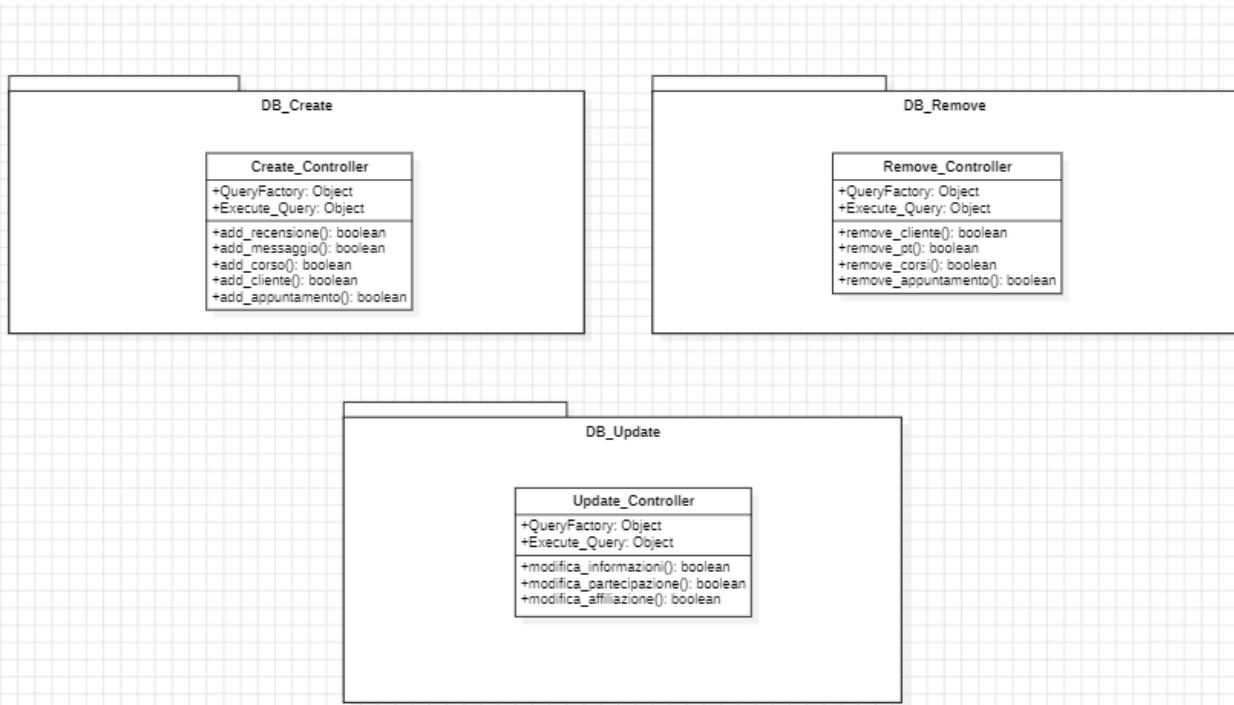


Figura 37: DB modify modificato

Un'altra modifica importante alla struttura di progettazione iniziale è stata quella su DB\_Modify: durante la progettazione avevamo separato in classi diverse le funzioni necessarie per interagire e modificare le operazioni sul Database , ma , durante lo sviluppo ci siamo accorti si trattava di una strategia ridondante e inutile, quindi abbiamo creato 3 classi che gestiscono i tipi di modifica del database (CRUD) , e che al loro interno contengono le funzioni necessarie per esaudire tutte le richieste delle GUI , in questo modo lo sviluppo c'è stato enormemente semplificato , in quanto un servizio era facilmente individuabile dipendente dal tipo operazione che richiedeva .

Ulteriormente queste funzioni di interazione con il DB condividono tra di loro variabili di accesso a Query Factory e execute\_query , permettendo quindi di ridurre la quantità di codice necessario.

## 4 Test

In questa sezione sono presenti i test del software, realizzati tramite il framework JUnit , per integrare Junit del progetto abbiamo scelto di implementare un modulo di tipo Maven , che ci permettesse di dichiarare le dependecies e le librerie associate in modo efficace , siamo poi passati allo sviluppo dei test sulle operazioni principali effettuabili sul Database attraverso le interfacce , da vari enti differenti.

Abbiamo eseguito sia dei test funzionali che test di unità, per verificare il corretto funzionamento dei principali metodi dell'applicazione e delle singole componenti del codice. I test sono stati organizzati in base alle classi specifiche, garantendo una migliore organizzazione delle cartelle.

Nelle prossime sezione riportiamo alcuni esempi, più importanti, di test. Abbiamo selezionato gli esempi che descrivono meglio le caratteristiche e funzioni del package in questione.

Volgiamo porre l'attenzione ai metodi *SetUp* e *TearDown*, che sono responsabili di inizializzare la connessione con il database e una volta eseguiti i testi vanno a eliminare tutti i parametri utilizzati o le informazioni inserite per il test in modo da ripulire il database e renderlo riutilizzabile per successive richieste.

```
@BeforeAll  
static void setUp() throws SQLException  
{  
    System.out.println("Executing setup");  
    testConnection = DriverManager.getConnection(URL, USER, PASSWORD);  
}
```

Figura 38: Metodo SetUp()

```
@AfterAll  
public static void tearDown() throws SQLException  
{  
    System.out.println("Executing teardown");  
    String query = "DELETE FROM utenti WHERE utenti.idUtenti = ?";  
    PreparedStatement pstmt = testConnection.prepareStatement(query);  
    pstmt.setInt(1,C.getID());  
    pstmt.executeUpdate();  
    pstmt.close();  
    if(testConnection!= null)  
    {  
        testConnection.close();  
    }  
}
```

Figura 39: Metodo TearDown()

Anticipiamo anche che per alcune funzioni di testing la TearDown è stata leggermente modificata, per esempio vediamo la funzione per il test dei messaggi, che va a eliminare il messaggio creato per la verifica, presente nel database.



```
@AfterAll
public static void tearDown() throws SQLException
{
    System.out.println("Executing teardown");
    String query = "DELETE FROM messaggio_u_pt WHERE destinatario = ?";
    PreparedStatement pstmt = testConnection.prepareStatement(query);
    pstmt.setInt(1,10);
    pstmt.executeUpdate();
    pstmt.close();
    if(testConnection!= null)
    {
        testConnection.close();
    }
}
```

Figura 40: Metodo TearDown() per test di messaggi

Vediamo anche la TearDown per il test riguardante l'iscrizione dei corsi. Come abbiamo già visto andremo a ripulire il database dai corsi creati.



```
@AfterAll
public static void tearDown() throws SQLException
{
    System.out.println("Executing teardown");
    String query = "DELETE FROM partecipazione WHERE partecipazione.idUtente = ?";
    PreparedStatement pstmt = testConnection.prepareStatement(query);
    pstmt.setInt(1,1000);
    pstmt.executeUpdate();
    pstmt.close();
    if(testConnection!= null)
    {
        testConnection.close();
    }
}
```

Figura 41: Metodo TearDown() per test di corsi

## 4.1 Test di inserimento utente

In questo test abbiamo testato che un impiegato possa iscrivere un nuovo cliente alla palestra. Il software ha il compito di creare un cliente ed aggiungerlo alla lista dei clienti presente nel database. Dopodiché, tramite le funzione messe a disposizione da JUnit abbiamo comparato le informazioni che abbiamo inserito attraverso l'interfaccia con quelle all'interno del database, per verificare il corretto funzionamento.



```
● ● ●

@Test
void test_addCliente()
{
    try
    {
        Employee E = new Employee(1,"marco","marroni");

        int id = count++;
        String nome = "Mario";
        String cognome = "Rossi";
        Date scadenza = Date.valueOf("2027-11-12");
        Subscription abbonamento = new Subscription(false, scadenza);
        C = new Client(id, nome, cognome, abbonamento);

        Interface_Client EC = new Interface_Client(E);

        EC.aggiungi_cliente(id,nome,cognome,scadenza);

        System.out.println("Tentativo di recuperare il nuovo cliente inserito : ");

        String query = "SELECT * FROM utenti WHERE utenti.idUtenti= ? ";

        System.out.println(query);
        PreparedStatement pstmt = testConnection.prepareStatement(query);

        pstmt.setInt(1, C.getID());

        System.out.println("statement preperad")
        ResultSet rs = pstmt.executeQuery();
        assertTrue(rs.next());

        assertEquals("Mario", rs.getString("nome"), "Il nome non corrisponde.");
        assertEquals("Rossi", rs.getString("cognome"), "Il cognome non corrisponde.");
        assertEquals(Date.valueOf("2027-11-12"), rs.getDate("scadenza"), "La scadenza non corrisponde.");
        pstmt.close();

    } catch (SQLException e) {
        fail("Errore nella connessione al database di test: " + e.getMessage());
    }
}
```

Figura 42: Test inserimento utente

## 4.2 Test Iscrizione ad un corso

In questo test puntavamo a verificare l’iscrizione ad un corso selezionato, da parte dell’utente. Notiamo come viene inizialmente creato un corso ed un cliente, successivamente andiamo a testare che l’interfaccia, che ha il compito di aggiungere il cliente alla lista di partecipanti, funzioni correttamente. Controlliamo con i metodo di JUnit se effettivamente è presente un’iscrizione al corso sul database.



```
@Test
void test_iscrizione()
{
    try
    {

        Date scadenza = Date.valueOf("2027-11-12");
        Subcription abbonamento = new Subcription(false, scadenza);
        Client C = new Client(1000,"Mario","Rossi",abbonamento);

        class_Interface CI = new class_Interface(C);

        Time durata = Time.valueOf("19:30:00");
        Corso corso = new Corso(1,"pilates",durata,0);

        System.out.println("tentativo di iscrizione al corso: ");
        CI.iscrizione_corso(corso);

        System.out.println("tentativo di recuperare la nuova partecipazione: ");
        String query ="SELECT * FROM partecipazione WHERE partecipazione.idUtente= ? ";

        PreparedStatement pstmt = testConnection.prepareStatement(query);
        pstmt.setInt(1, C.getID());

        System.out.println("Statement ready");
        ResultSet rs = pstmt.executeQuery();
        System.out.println("Query executed");

        assertTrue(rs.next());
        assertEquals(C.getID(),rs.getInt("idUtente"));
        assertEquals(corso.getId(),rs.getInt("idCorso"));

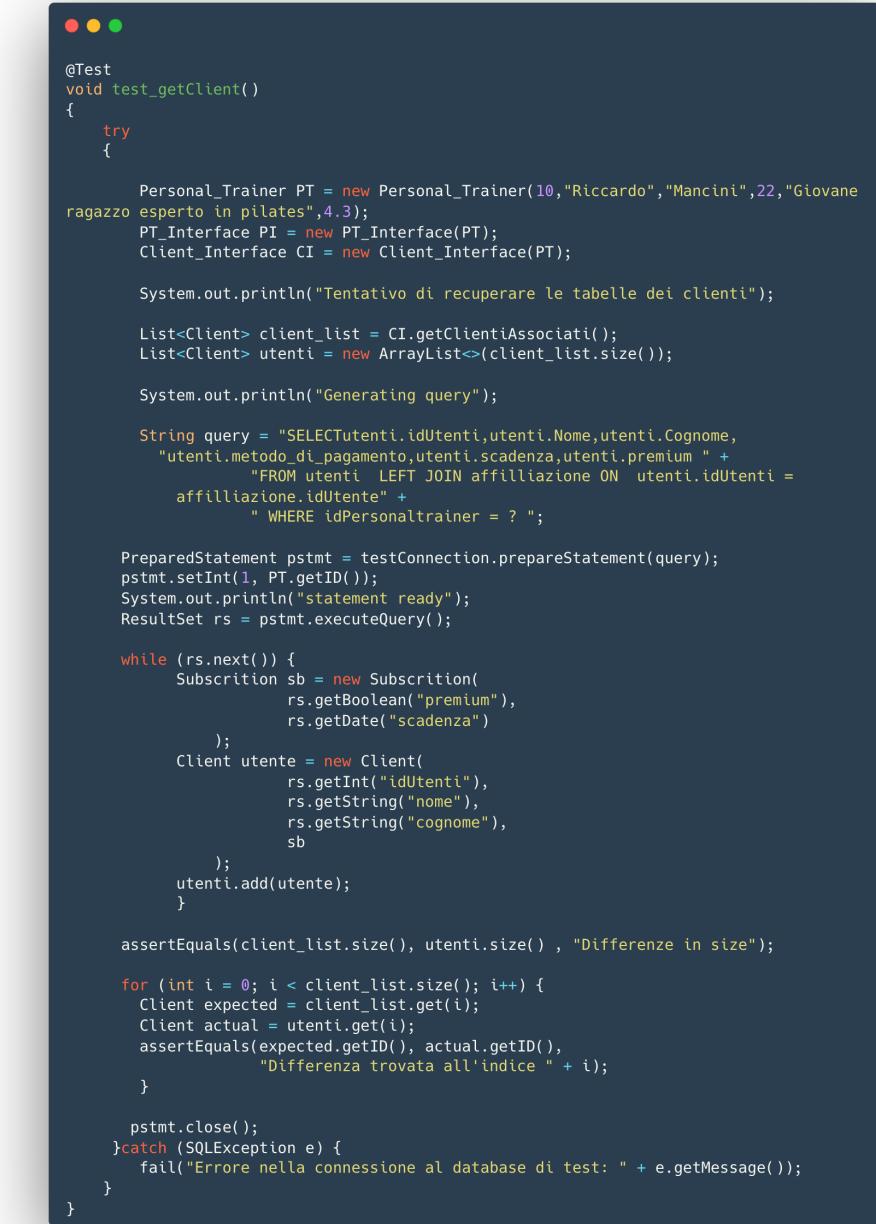
        pstmt.close();

    }catch (SQLException e) {
        fail("Errore nella connessione al database di test: " + e.getMessage());
    }
}
```

Figura 43: Test iscrizione ad un corso

### 4.3 Test di visualizzazione dei clienti associati ad un Personal trainer

In questo test volevamo verificare se la funzione che ha il compito di prelevare la tabella dei clienti associati ad un particolare Personal trainer funzionasse correttamente. Notiamo la creazione di un Personal trainer e delle interfacce di interesse. Dopodiché andiamo ad eseguire la Query per estrarre quei particolari dati e verifichiamo che sia lo stesso risultato della funzionalità testata.



The screenshot shows a Java code editor with a JUnit test class named `test_getClient`. The code implements a test for retrieving associated clients from a database. It creates a `Personal_Trainer` object, initializes `PT_Interface` and `Client_Interface`, prints a message, retrieves a list of clients, generates a query string, prepares a `PreparedStatement`, executes it, and iterates through the results to verify the size and contents of the retrieved list against the expected list. It also handles potential `SQLException`.

```
@Test
void test_getClient()
{
    try
    {
        Personal_Trainer PT = new Personal_Trainer(10,"Riccardo","Mancini",22,"Giovane ragazzo esperto in pilates",4.3);
        PT_Interface PI = new PT_Interface(PT);
        Client_Interface CI = new Client_Interface(PT);

        System.out.println("Tentativo di recuperare le tabelle dei clienti");

        List<Client> client_list = CI.getClientiAssociati();
        List<Client> utenti = new ArrayList<>(client_list.size());

        System.out.println("Generating query");

        String query = "SELECTutenti.idUtenti,utenti.nome,utenti.cognome,
                       \"utenti.metodo_di_pagamento\",utenti.scadenza,utenti.premium " +
                       "FROM utenti LEFT JOIN affilliazione ON utenti.idUtenti =
                       affilliazione.idUtente" +
                       " WHERE idPersonaltrainer = ? ";

        PreparedStatement pstmt = testConnection.prepareStatement(query);
        pstmt.setInt(1, PT.getID());
        System.out.println("statement ready");
        ResultSet rs = pstmt.executeQuery();

        while (rs.next())
        {
            Subscription sb = new Subscription(
                rs.getBoolean("premium"),
                rs.getDate("scadenza")
            );
            Client utente = new Client(
                rs.getInt("idUtenti"),
                rs.getString("nome"),
                rs.getString("cognome"),
                sb
            );
            utenti.add(utente);
        }

        assertEquals(client_list.size(), utenti.size() , "Differenze in size");

        for (int i = 0; i < client_list.size(); i++)
        {
            Client expected = client_list.get(i);
            Client actual = utenti.get(i);
            assertEquals(expected.getID(), actual.getID(),
                        "Differenza trovata all'indice " + i);
        }

        pstmt.close();
    }catch (SQLException e)
    {
        fail("Errore nella connessione al database di test: " + e.getMessage());
    }
}
```

Figura 44: Test visualizzazione clienti associati

#### 4.4 Test invio messaggio

In questo test volevamo provare che un messaggio inviato venga correttamente caricato, e che sia visualizzabile sul database, da parte del destinatario. Vediamo la creazione di un Utente e di un Personal trainer, dopodiché creiamo le interfacce di interesse. Una volta creato il messaggio lo carichiamo, tramite il metodo che stiamo testando, nel database e andiamo a verificare che il messaggio sia effettivamente presente nel database, confrontando i vari ID che contiene.



The screenshot shows a Java code editor with a dark theme. The code is a JUnit test for sending a message. It imports various classes and defines objects for a Client, Personal\_Trainer, and Notification\_Interface. It then sends a message to the Personal\_Trainer and performs a SELECT query on the database to verify the message was received. A catch block handles any SQL exceptions.

```
@Test
void test_messaggio()
{
    try
    {
        Date scadenza = Date.valueOf("2027-11-12");
        Subcription abbonamento = new Subcription(false, scadenza);
        Client C = new Client(1000, "Mario", "Rossi", abbonamento);
        Personal_Trainer PT = new Personal_Trainer(10, "Riccardo", "Mancini", 22, "Giovane ragazzo esperto in pilates", 4.3);

        Notification_Interface NI = newNotification_Interface(C);
        System.out.println("Tentaivo di invio messaggio");
        String message = "test";
        NI.message_reply(PT.getID(), message);
        System.out.println("Tentativo di recuperare il nuovo messaggio : ");
        String query = "SELECT * FROM messaggio_u_pt WHERE destinatario = ?";

        PreparedStatement pstmt = testConnection.prepareStatement(query);
        pstmt.setInt(1, PT.getID());

        System.out.println("statement ready");
        ResultSet rs = pstmt.executeQuery();
        System.out.println("Query executed");

        assertTrue(rs.next());
        assertEquals(C.getID(), rs.getInt("mittente"));
        assertEquals(PT.getID(), rs.getInt("destinatario"));
        assertEquals(message, rs.getString("messaggio"));

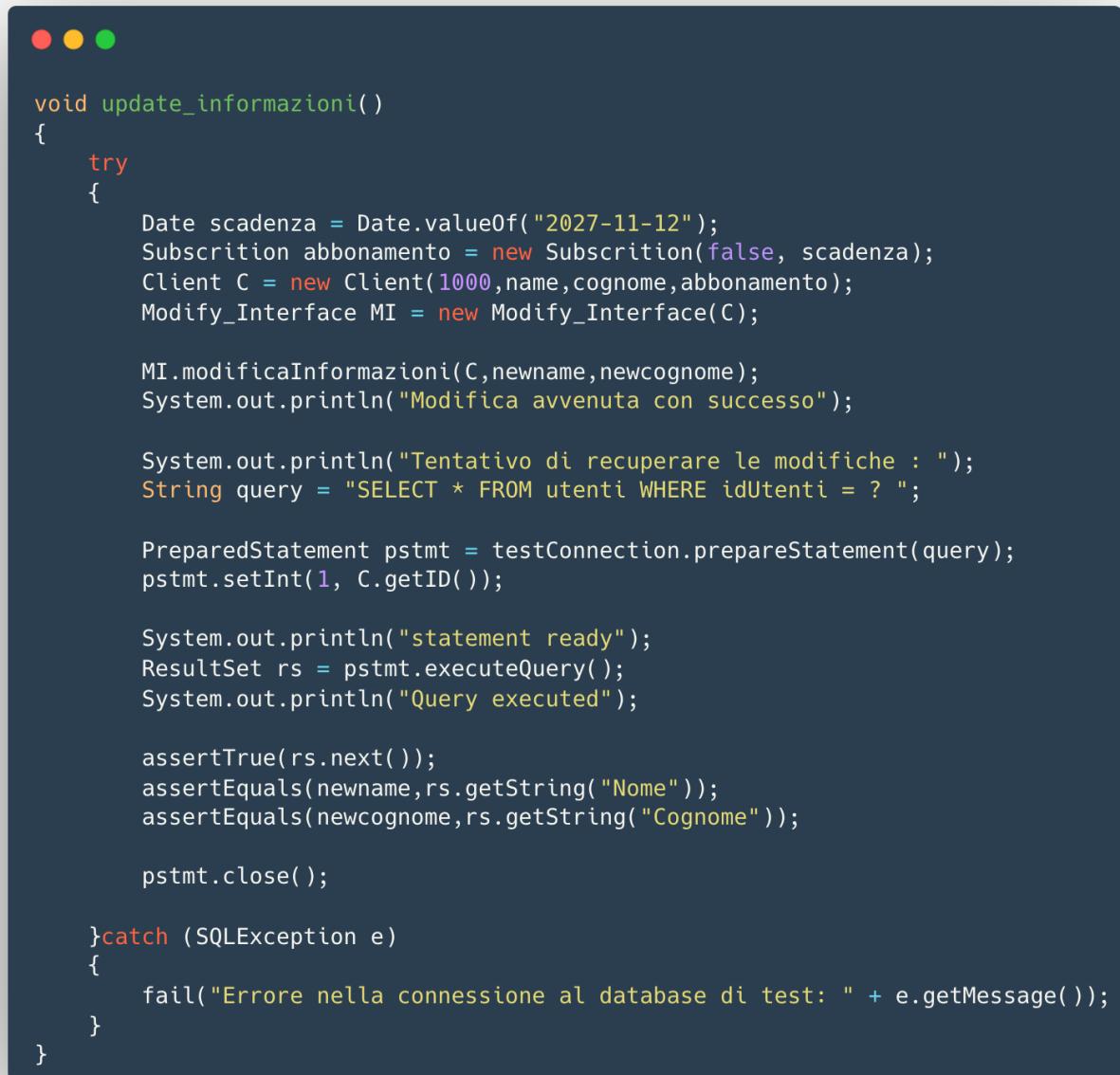
        pstmt.close();

    }catch (SQLException e) {
        fail("Errore nella conessione al database di test: " + e.getMessage());
    }
}
```

Figura 45: Test invio messaggio

## 4.5 Test modifica delle informazioni

In questo test volevamo verificare che i metodi, messi a disposizione per tutte le entità del software, avessero modo di modificare le loro informazioni personali. Possiamo vedere come un cliente venga creato e se ne modifichi le informazioni personali all'interno del database, tramite l'interfaccia apposita.



```
void update_informazioni()
{
    try
    {
        Date scadenza = Date.valueOf("2027-11-12");
        Subcription abbonamento = new Subcription(false, scadenza);
        Client C = new Client(1000, name, cognome, abbonamento);
        Modify_Interface MI = new Modify_Interface(C);

        MI.modificaInformazioni(C,newname,newcognome);
        System.out.println("Modifica avvenuta con successo");

        System.out.println("Tentativo di recuperare le modifiche : ");
        String query = "SELECT * FROM utenti WHERE idUtenti = ? ";

        PreparedStatement pstmt = testConnection.prepareStatement(query);
        pstmt.setInt(1, C.getID());

        System.out.println("statement ready");
        ResultSet rs = pstmt.executeQuery();
        System.out.println("Query executed");

        assertTrue(rs.next());
        assertEquals(newname,rs.getString("Nome"));
        assertEquals(newcognome,rs.getString("Cognome"));

        pstmt.close();

    }catch (SQLException e)
    {
        fail("Errore nella connessione al database di test: " + e.getMessage());
    }
}
```

Figura 46: Test modifica delle informazioni