Relazione progetto "Gym Manager"

Mattia Mari
Matteo Manzi
Daniele Manfredonia

Indice

Capitolo 1

Analisi

Il progetto ha lo scopo di realizzare un software per la gestione di una palestra. Il software offre strumenti necessari per gestire: clienti, abbonamenti, personale della palestra e attrezzi.

1.1 Requisiti

Requisiti funzionali

- accesso al software tramite login, con username e password per ogni dipendente;
- ogni dipendente può avere un ruolo differente che gli permette o impedisce di svolgere certe operazioni;
- gestione dell'anagrafica di clienti e dipendenti;
- possono essere creati diversi tipi di abbonamento, ciascuno con il proprio nome e prezzo;
- due macro-tipologie di abbonamenti: a tempo e con ingressi a scalare;
- ogni cliente iscritto alla palestra può sottoscrivere un abbonamento che gli permetterà l'ingresso alla palestra, tramite tornello con lettore di tessere o simili;
- registrazione degli attrezzi presenti nella palestra;
- gestione dei turni di lavoro del personale;
- il cliente può visualizzare lo stato del proprio abbonamento;
- il cliente può utilizzare una bilancia automatica per registrare il proprio peso e altezza all'interno del software

Requisiti non funzionali

- rendere l'applicazione indipendente dal metodo di persistenza dei dati;
- garantire una certa facilità d'uso per gli utenti;
- il software deve essere facilmente estendibile, ad esempio per facilitare l'aggiunta di nuove macro-tipologie di abbonamenti e statistiche per il cliente.

1.2 Analisi e modello del dominio

Il software deve svolgere le funzioni necessarie per la gestione dei dati della palestra. La palestra comprende uno staff che svolge mansioni differenti.

Ogni addetto della palestra viene riconosciuto tramite username e password e potrà svolgere, tramite il software, solo le attività pertinenti al suo ruolo, ad esempio a chi gestisce la cassa potrebbe non essere permessa la modifica delle tipologie di abbonamento.

I ruoli, assegnati agli addetti in fase di creazione dell'utente, possono essere creati dall'amministratore e comprendono uno o più permessi, identificati da un nome univoco e una descrizione.

E' possibile iscrivere nuovi clienti alla palestra con i relativi dati anagrafici. Ai clienti si associano abbonamenti, che possono essere creati e modificati dal gestore della palestra in base ai servizi offerti.

Ogni tipologia di abbonamento ha il proprio prezzo e, al momento dell'iscrizione, potrà comprendere ulteriori servizi aggiuntivi (per esempio sauna o piscina), il cui prezzo va a sommarsi a quello dell'abbonamento.

Le tipologie di abbonamento fanno parte di due macro-tipologie: a tempo e con ingressi a scalare. Gli abbonamenti a tempo hanno una data di inizio e una durata; l'abbonamento sarà ritenuto non valido oltre la data di scadenza. Gli abbonamenti con ingressi a scalare permettono di accedere alla palestra finché il numero di ingressi non si azzera.

I clienti, dopo aver effettuato l'accesso tramite username e password possono accedere alla propria dashboard ed avere un riepilogo sui propri abbonamenti e servizi, inoltre possono tenere traccia del loro peso e altezza, tramite una bilancia automatica. Questi dati sono utilizzati per il calcolo del BMI e, più in generale, sono un modo per il cliente di valutare il proprio progresso.

La palestra utilizza uno o più attrezzi che necessitano di una manutenzione regolare. E' possibile tenere traccia delle date di manutenzione per ogni singolo attrezzo.

E' possibile gestire il calendario dei turni per lo staff, con turni ricorrenti (ad esempio *tutti i lunedì, mercoledì e giovedì dalle 13.00 alle 19.00*) e turni singoli (ad esempio *il 05/01/2020 dalle 08.00 alle 13.00*).

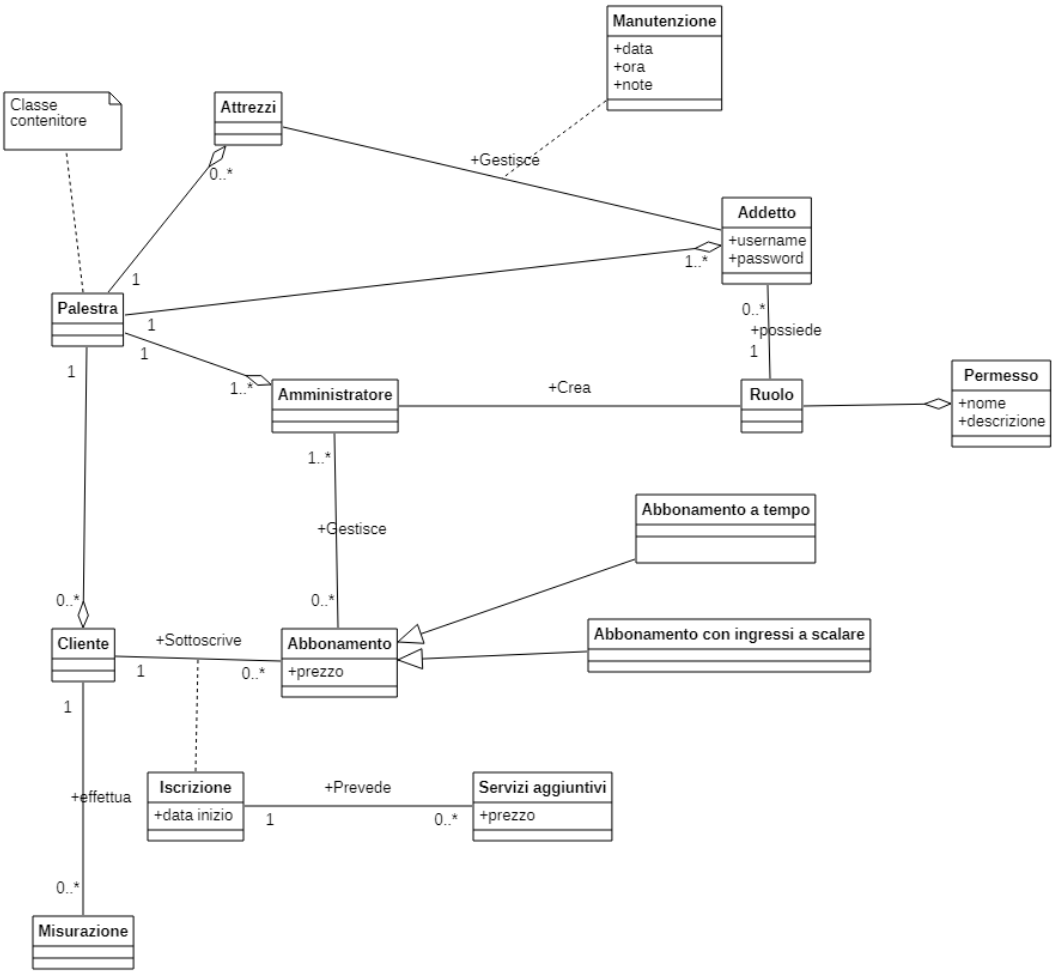

Figura 1.1: Schema UML che mostra l'analisi del dominio.

Capitolo 2

Design

2.1 Architettura

Data la natura del software è stato scelto il pattern MVC al fine di separare efficientemente la logica di business da quella di presentazione. Ogni view del software è realizzata tramite JavaFX e comprende una *pagina* e relativo controller. Il servizio di navigazione (*NavigationService*), simulando il comportamento che si avrebbe con una web app o app mobile, si occupa di gestire il passaggio da una pagina all'altra offrendo anche la possibilità di tornare indietro alla pagina precedente.

La gestione dei dati è stata realizzata tramite il **Repository Pattern** che migliora l'estesibilità del software, rendendolo agnostico al metodo di persistenza. Questo permette di cambiare il tipo di database senza dover modificare la logica di business, ma soltanto le implementazioni dei repository. Ogni istanza di ogni entità del Model è identificata da un id alfanumerico univoco, generato direttamente dal software, che è di fatto compatibile con qualsiasi tipo di database. Ogni tipo di entità possiede una sua interfaccia di repository, che eredita alcuni metodi generici (*add, get, remove*) e offre metodi specifici alla particolare entità ad esempio *getByUsername*. Questo fa in modo di centralizzare tutto ciò che riguarda l'accesso al metodo di persistenza all'interno del repository che può essere implementato, ad esempio, per supportare caching e failover nonchè migliorare le performance, demandando alcune operazioni di filtraggio e ricerca direttamente al DBMS.

Tutte le entità utilizzano il **Builder Pattern** che permette la creazione più agevole di oggetti con tante proprietà. Inoltre garantisce l'immutabilità che previene eventuali *side effects* e impedisce l'esistenza di istanze non valide.

Ove necessario si è fatto uso di tecniche di **dependency injection**, dove le dipendenze sono passate direttamente al costruttore delle classi che ne fanno uso. Questo permette un miglior unit testing (ad esempio utilizzando *mock* anziché vere istanze) nonché la condivisione di istanze senza dover ricorrere ai singleton. Pertanto, i vari repository e servizi vengono istanziati una sola volta nella *main class* del software e passati al costruttore di tutti i controller che ne hanno bisogno.

2.2 Design dettagliato

I seguenti schemi rappresentano lo scheletro del software: codice utilizzato da tutti gli altri componenti dell'applicazione.

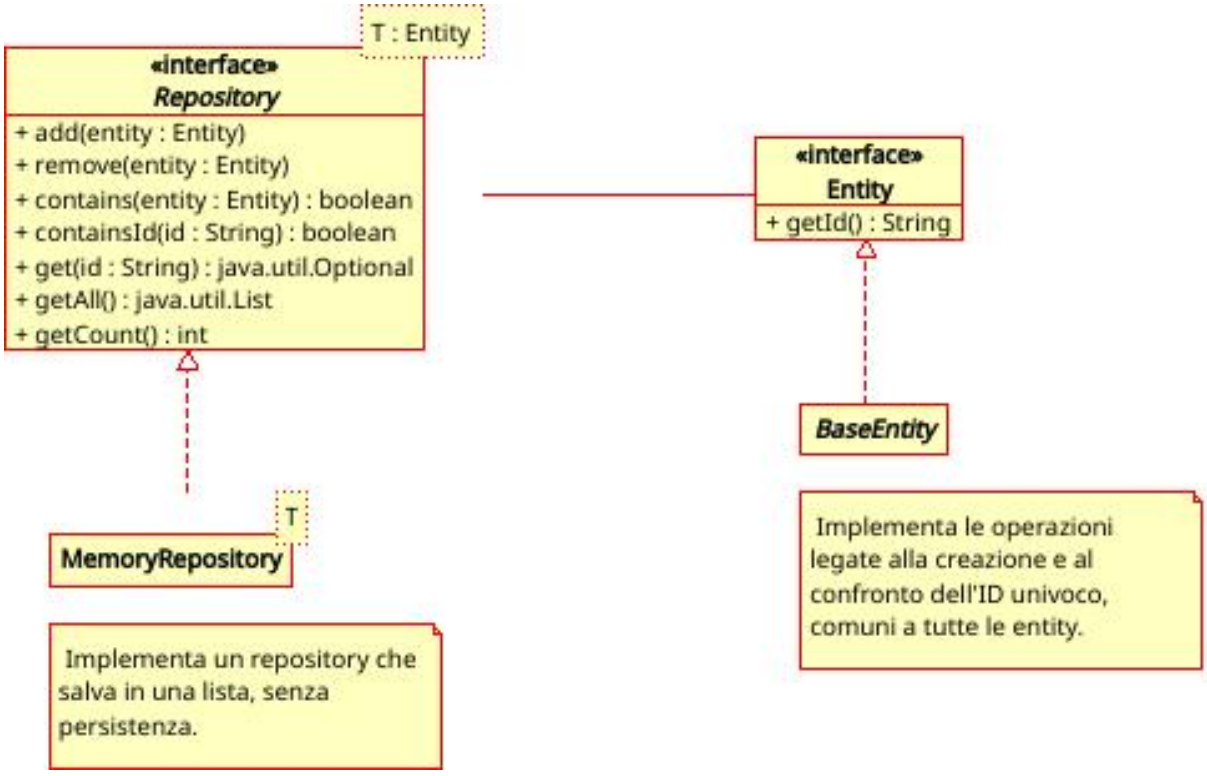

Figura 2.1: Schema UML delle classi di base per repository e entità.

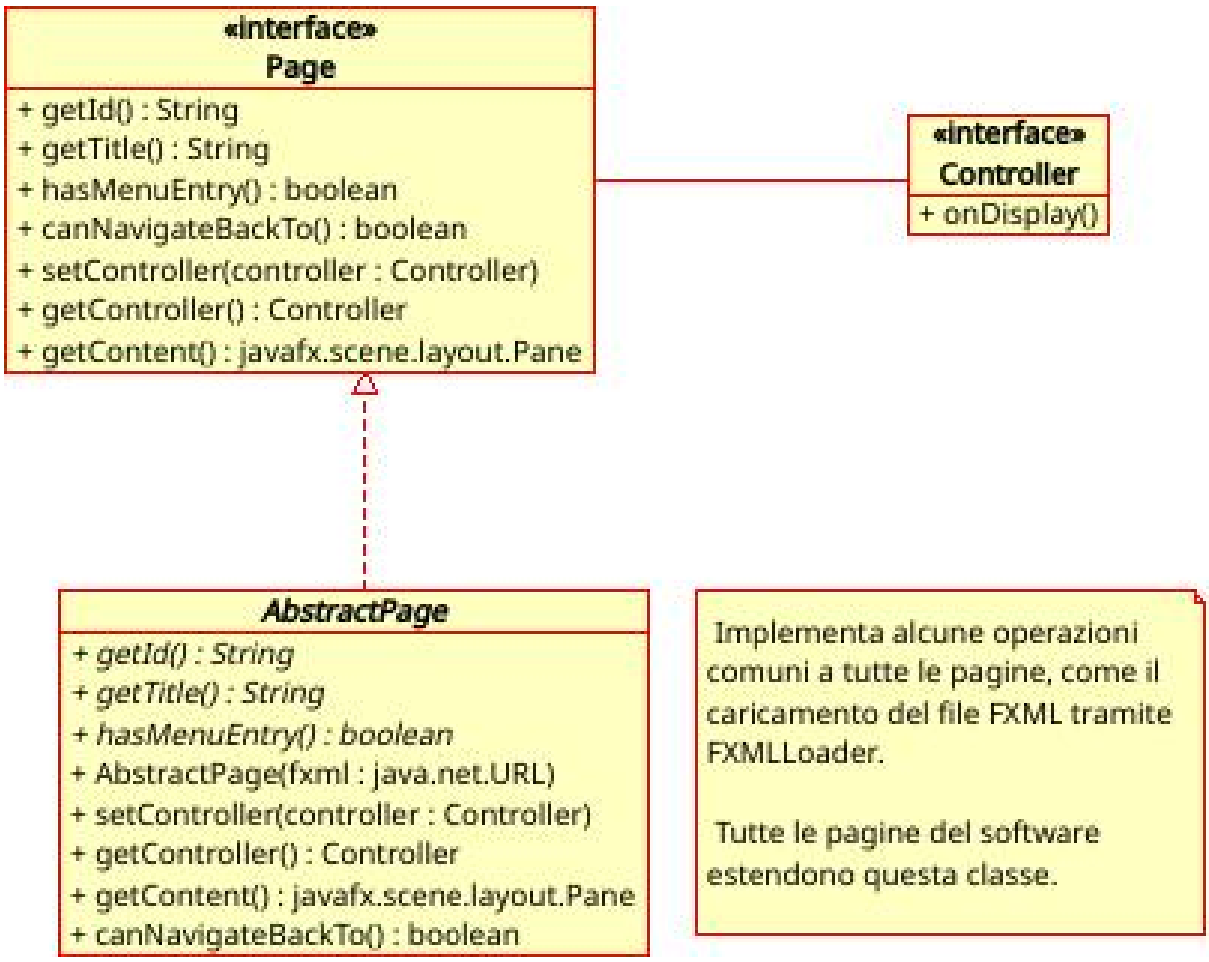

Figura 2.2: Schema UML delle classi di base per pagine e controller.

Mattia Mari

Nota: negli schemi UML, le relazioni di aggregazione che hanno un'interfaccia ad un capo della relazione sono rappresentate come associazioni semplici perchè il software utilizzato (Umbrello) non permette l'uso della rappresentazione corretta.

I campi privati, quando mostrati, indicano la presenza di getter generati automaticamente da Lombok.

Authentication e Authorization

Autenticazione e autorizzazione sono gestite da *AuthService*. Il servizio fa uso di *UserRepository* per salvare le istanze di *User*. L'autenticazione è effettuata in modo classico tramite username e password: se è presente un utente con lo username specificato allora viene calcolato l'hash SHA-256 della password e confrontato con l'hash salvato nell'oggetto *User*.

L'utente che ha effettuato l'accesso può essere ottenuto tramite uno specfico metodo del servizio. E' possibile registrare callback che vengono invocati a login avvenuto; il controller principale ne fa uso per popolare il menù laterale dell'applicazione, inizialmente vuoto e per svuotarlo al logout.

L'autorizzazione è gestita tramite ruoli che possono essere assegnati ad ogni utente. Il ruolo contiene una lista di permessi che permettono all'utente di accedere a specifiche pagine e funzionalità. Ogni pagina può registrare i permessi messi a disposizione tramite un metodo del servizio. I permessi registrati saranno poi elencati nella schermata di creazione dei ruoli per permetterne la scelta. Generalmente ad ogni controller viene passata l'istanza di AuthService e ognuno di essi registra indipendentemente i propri permessi. Un controller può verificare se l'utente loggato possiede uno specifico permesso passandone il nome al metodo *userHasPermission*.

La gestione dei permessi così concepita permette di aggiungere nuovi controller arbitrariamente senza dover mettere mano ad altre parti del software.

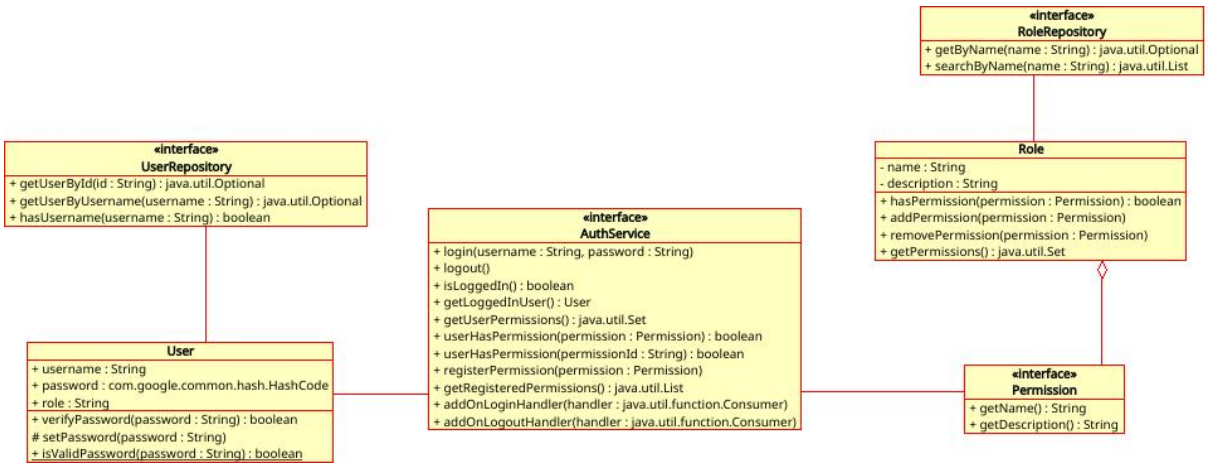

Figura 2.3: Schema UML del componente di autenticazione e autorizzazione.

Navigation

La navigazione tra le varie pagine del software è gestita da *NavigationService*. Il servizio permette di registrare oggetti di tipo *Page* (che contengono un id e un titolo) e di tenere traccia della navigazione tramite uno stack. E' possibile navigare verso una pagina specificandone l'id e anche tornare alla pagina precedente. Il servizio permette la registrazione di callback che vengono invocati ogni volta che si naviga verso una pagina. Questa funzionalità è utilizzata dal controller principale del software per poter mostrare a schermo la nuova pagina.

Per i casi in cui è necessario passare dati tra una pagina e l'altra, il servizio mette a disposizione l'oggetto *NavParams* che funge da contenitore chiave-valore. Un controller può inserire oggetti di qualsiasi tipo all'interno del contenitore per poter essere poi richiesti da un altro controller. Data la natura multi-tipo del contenitore, il controller che riceve un oggetto deve essere a conoscenza del suo tipo al fine di eseguirne il cast nel tipo appropriato.

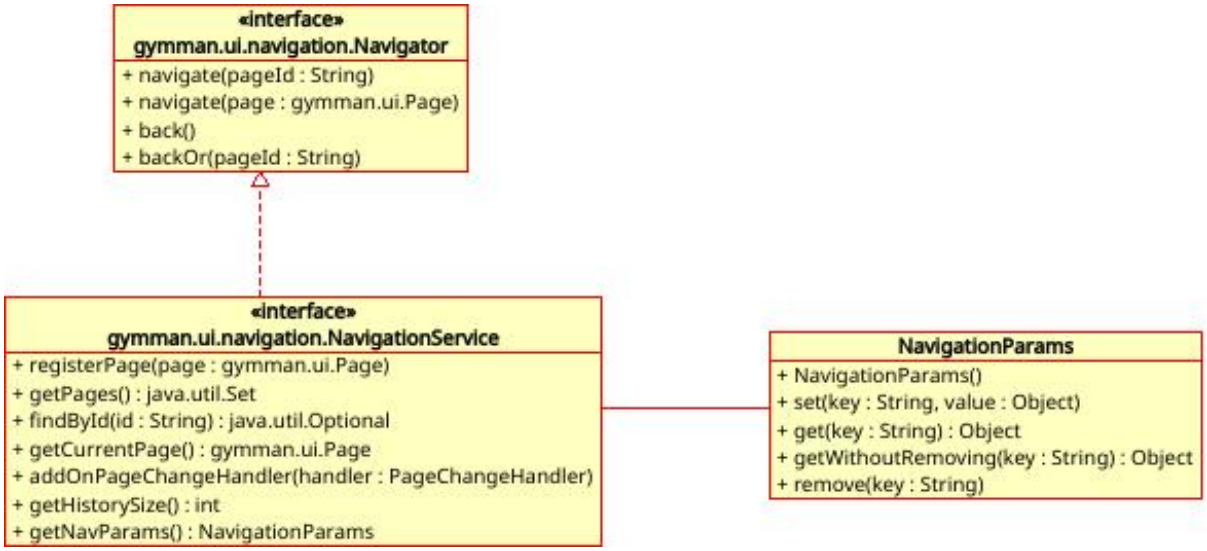

Figura 2.4: Schema UML del componente di navigazione.

Employee

Employee rappresenta un generico membro dello staff ed estende *User*. Come per tutte le altre entità, anche Employee sfrutta il builder pattern.

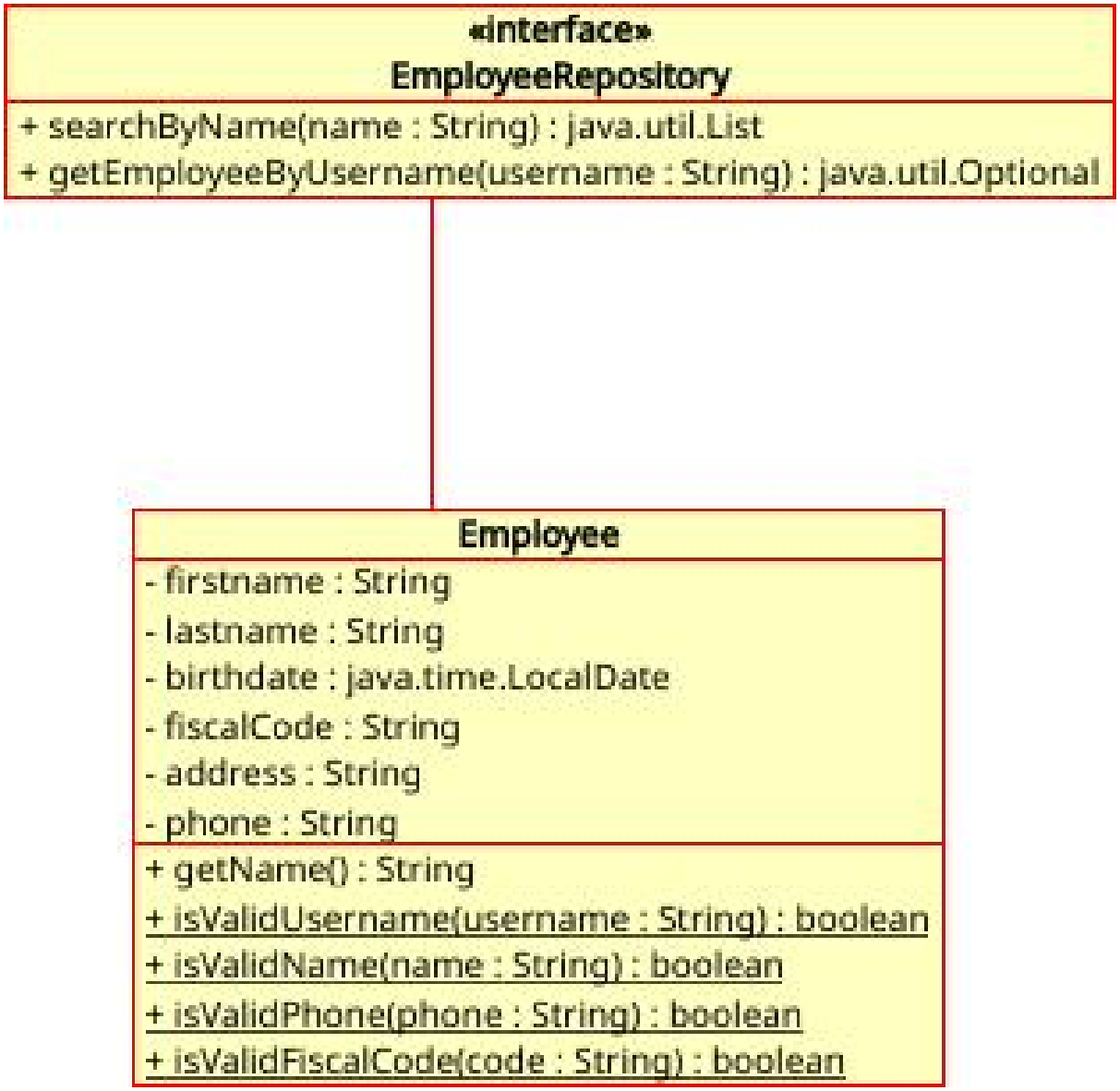

Figura 2.5: Schema UML delle classi relative agli impiegati.

Calendario turni

Il calendario è gestito tramite *WorkShift* che possono essere ricorrenti oppure singoli. Con i turni ricorrenti è possibile selezionare i giorni della settimana interessati e l'orario di inizio e di fine. I turni singoli, invece, permettono la selezione di una data specifica.

I *WorkShift* sono salvati in un repository dedicato e gestiti da *CalendarService*. Richiedendo il calendario per una particolare settimana i *WorkShift* interessati sono raggruppati in una lista di *CalendarDay* che rappresenta i turni di una data specifica. I turni ricorrenti vengono inseriti nel *CalendarDay* sulla base del giorno della settimana a cui si riferisce. Questo evita di doversi preoccupare del tipo di turno rendendo possibile trattare tutti i turni allo stesso modo.

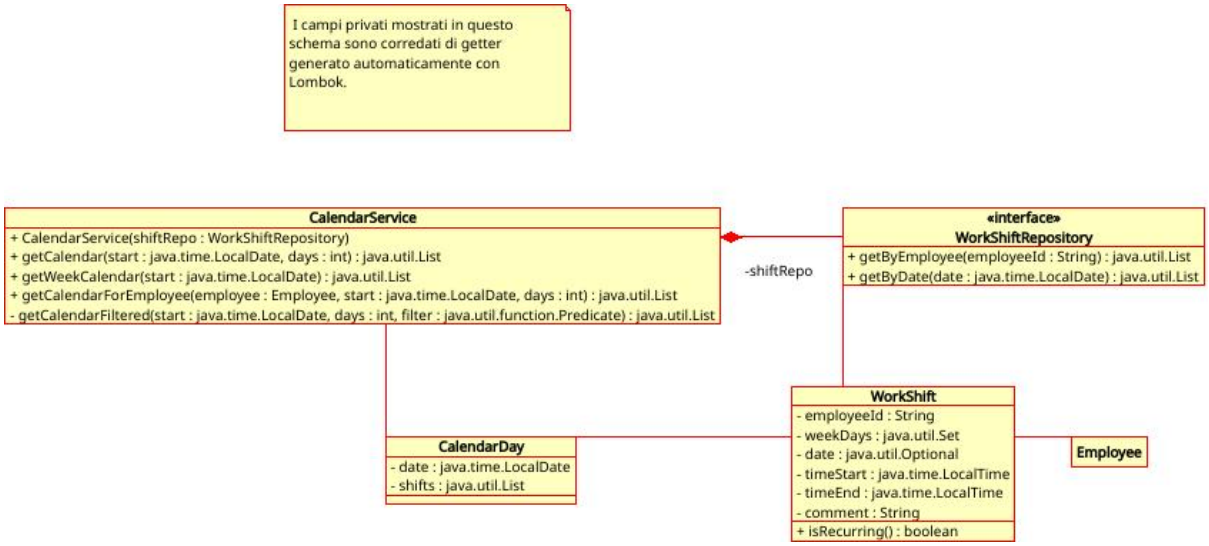

Figura 2.6: Schema UML delle classi relative alla gestione del calendario.

Matteo Manzi

Tool

La classe *Tool* è la gestione e creazione del singolo attrezzo. Per gestire questa figura ho deciso di applicare il **Builder Pattern**, avendo molteplici campi obbligatori. La visione e ricerca dell'elenco degli attrezzi è riservata ai dipendenti. La creazione dell'attrezzo è un modulo a cui si interfacciano solo i dipendenti della palestra. La classe Tool è unica per numero seriale e nome attrezzo. La ricerca e visione è data da un elenco di:

- Nome del singolo attrezzo creato.
- Descrizione delle funzionalità ed utilizzo dell'attrezzo.
- Seriale del singolo attrezzo.
- Manutenzione, che corrisponde al periodo il quale deve essere soggetto a manutenzione per la corretta esecuzione.

Tramite questa interfaccia è possibile anche l'eliminazione e la modifica dell'attrezzo, in caso l'utente ha il ruolo adatto per farlo.

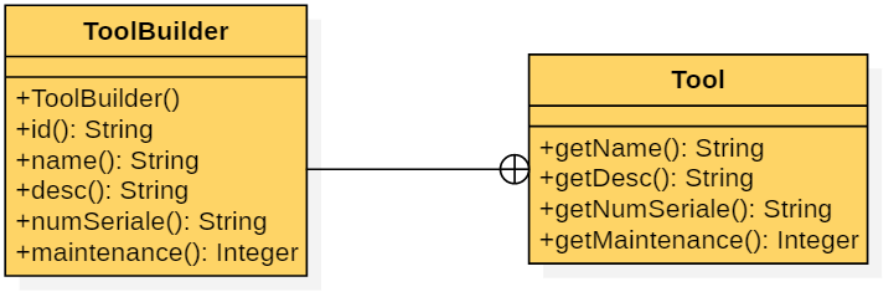

Figura 2.7: Schema UML che mostra la creazione di un attrezzo.

Bmi

La classe *Bmi* implementa l'aggiunta dei dati per calcolare il Body Mass Index di un cliente. Anche qua ho utilizzato il **Builder Pattern** poiché i campi sono obbligatori per l'utente. I campi coinvolti sono:

- Altezza
- Peso

Ad ogni campo vengono eseguiti dei controlli sui valori inseriti.

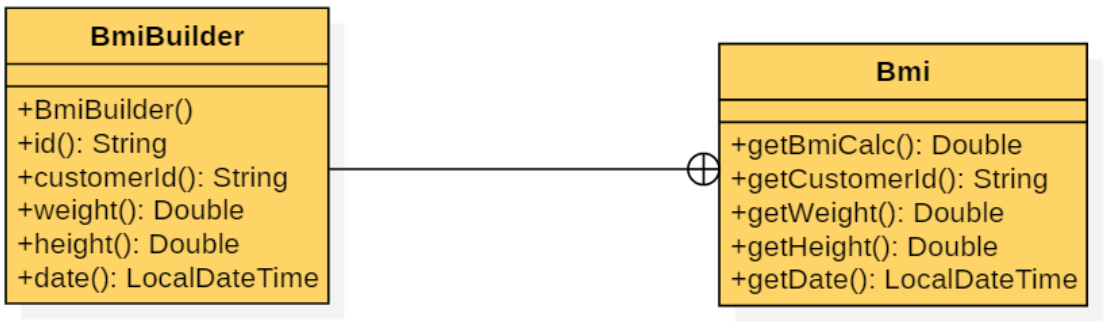

Figura 2.8: Schema UML che mostra la gestione della classe Bmi.

InfoClient

La classe *InfoClient* illustra in una interfaccia la sintesi di tutte le informazioni principali di un cliente all'interno della palestra, è univoco per ogni utente e vi si può accedere solo tramite username e password. L'interfaccia è suddivisa in:

- Dati Personali
- Calcolo BMI
- Abbonamento e Servizi Aggiuntivi

- Legenda Bmi

L'utente può interagire con la schermata Bmi per aggiornare il suo stato di sviluppo, successivamente viene calcolato ed espresso nella sezione *Calcolo Bmi*; viene evidenziato anche nella Legenda Bmi il range in cui rientra l'utente.

Daniele Manfredonia

Customer

La classe *Customer* implementa la figura del cliente. Per gestire questa figura ho deciso di applicare il **Builder Pattern** visti i tanti campi di questa classe e la necessità di averli tutti presenti. Oltre ai classici campi anagrafici(nome, cognome, data di nascita...) sono presenti campi come email, telefono, password e nome utente. Infatti il *Customer*, essendo un utente del sistema, estende la classe *User* da cui eredita i campi necessari per effettuare il login. Ogni istanza di *Customer* è unica sia rispetto al proprio codice fiscale sia rispetto al nome utente. Per quanto riguarda la parte di controller ho implementato una tabella contenente tutte le istanze di *Customer* aggiunte al *CustomerRepository*. Tramite una apposito campo di testo è possibile filtrare i cliente in base al nome, cognome o nome utente in modo da rendere più agevole la ricerca di un determinato cliente. Inoltre da questa schermata è possibile passare alla schermata di creazione di un nuovo cliente. Durante l'inserimento dei dati il bottone per la creazione è disattivato e viene attivato solo nel caso in cui tutti i campi siano stati riempiti. Questo non è sufficiente affinché la creazione di un cliente avvenga infatti nella classe *Customer* sono presenti controlli per tutti i dati inseriti(ad esempio sul formato del codice fiscale). Per agevolare l'inserimento i campi che presentano valori non validi verranno evidenziati di rosso e nella casella di testo relativa al numero di telefono possono essere inseriti solo valori numerici. Il campo relativo alla password viene riempito automaticamente alla creazione di un cliente mentre verrà lasciato vuoto in caso di modifica di un cliente. Per far rimanere invariata la password basterà lasciare quel campo vuoto altrimenti si può inserire una nuova password che verrà memorizzata. Non potendo esistere due clienti con lo stesso codice fiscale o nome utente, si aprirà una finestra di dialogo contente un messaggio di errore nel caso si provi a creare un cliente con questi valori ripetuti. Con la tabella dei clienti è possibile visualizzare tutti i clienti iscritti alla palestra, apportare modifiche ai dati del cliente e nel caso eliminare un cliente dalla tabella sfruttando la colonna contenente questi tasti. Sempre dalla tabella sarà possibile iscrivere un cliente a nuovi corsi e monitorare le iscrizioni.

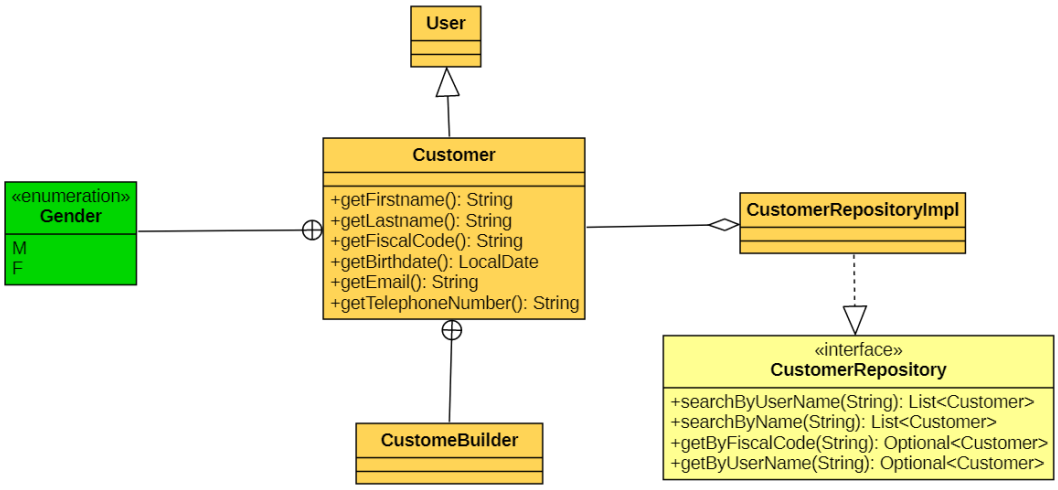

Figura 2.9: Schema UML che mostra il concetto di cliente.

Subscription type

La classe *SubscriptionType* rappresenta la tipologia di abbonamento e quindi tutti i possibili abbonamenti/corsi a cui un cliente può iscriversi. Questa classe presenta solo tre campi ma per maggiore chiarezza ed eleganza ho deciso di implementarla con il **Builder Pattern**. I campi presenti indicano il nome che viene assegnato all'abbonamento che deve essere univoco(non sarà quindi possibile creare due abbonamenti con lo stesso nome),la descrizione che brevemente spiega le caratteristiche di quel corso e il costo mensile dell'abbonamento. Nella parte di controller una tabella contiene tutte le tipologie di abbonamenti con i relativi dati e tramite questa si può gestire il *SubscriptionTypeRepository* creando, modificando ed eliminando un abbonamento. Da qui è possibile effettuare la ricerca di un abbonamento in base al nome e con l'apposito tasto passare alla schermata di creazione dell'abbonamento. Anche in questo caso non è possibile creare un abbonamento finché tutti i campi non saranno riempiti. Nel campo di testo relativo al prezzo mensile che si vuole associare all'abbonamento non è possibile inserire valori diversi da quelli numerici e dal punto.

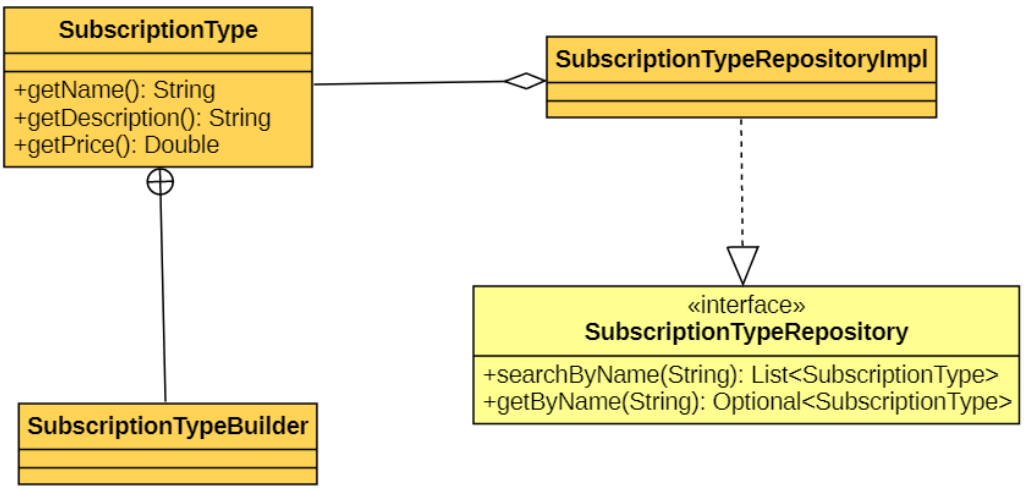

Figura 2.10: Schema UML che mostra il concetto di abbonamento.

Additional service

La classe *Additional Service* è una entità che ho deciso di implementare per rendere più completa la mia parte. Infatti questa classe rappresenta il concetto di "servizio aggiuntivo" e quindi tutte le attività extra che si possono trovare in una palestra diverse dai classici corsi. Con questa scelta sarà possibile estendere questo software oltre che alle piccole palestre a palestre più all'avanguardia che offrono un maggior numero di servizi ai propri clienti. Anche per questa classe è stato applicato il **Builder Pattern** e i campi sono gli stessi della classe *Subscription Type* nonostante rappresentino due concetti formalmente diversi. Anche la parte di controller è molto simile infatti tramite una tabella si riesce a monitorare tutti i servizi aggiuntivi presenti nel *AdditionalServiceRepository*,fare la ricerca sul nome e impedire la creazione in caso in cui tutti i campi non siano stati compilati.

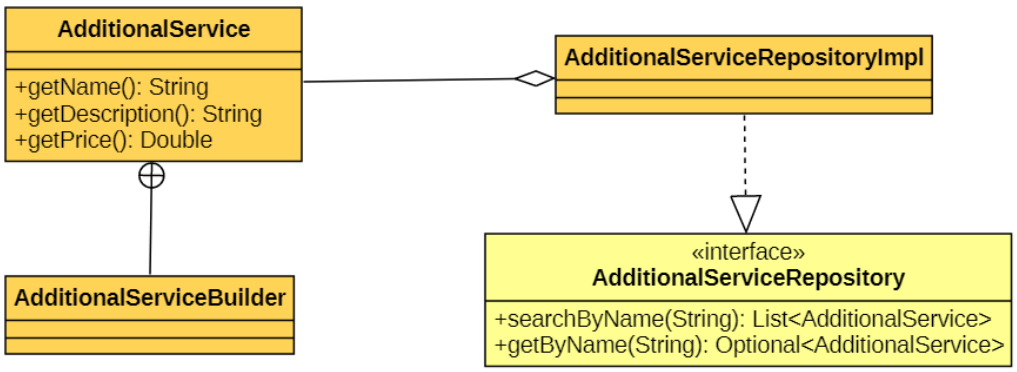

Figura 2.11: Schema UML che mostra il concetto di servizio aggiuntivo.

Registration

La classe *Registration* implementa l'iscrizione di un cliente a un abbonamento. Le iscrizioni sono di due tipi: iscrizioni a tempo nella quale si specifica la durata dell'abbonamento e ad ingressi dove viene specificato il numero di ingressi possibili. Durante la fase di analisi si è notato subito come le due tipologie di iscrizioni che si voleva implementare presentavano caratteristiche comuni, come ad esempio la tipologia di abbonamento da associare e l'id del cliente a cui associare l'iscrizione. Per questo motivo e per minimizzare il riutilizzo del codice, ho deciso di creare una classe astratta *Registration* che modellasse il concetto più generale di iscrizione mantenendo astratti i metodi caratteristici delle due tipologie. In questo modo nelle due classi *NumberedRegistration* e *TermRegistration* ho implementato i metodi astratti e inserito i campi necessari per completare la classe. Anche qui per l'iscrizione l'utilizzo del **Builder Pattern** risulta appropriato visto che queste classi presentano campi che mostrano:

- la tipologia di abbonamento collegato all'iscrizione
- i servizi aggiuntivi associabili all'iscrizione
- la possibile percentuale di sconto applicabile

e altri campi specifici per ogni classe. Nella parte di controller ci sarà ovviamente la possibilità di scegliere che tipologia di iscrizione effettuare e alcuni campi verranno logicamente abilitati/disabilitati. Ad ogni campo vengono effettuati controlli relativi ai valori assumibili in questo modo non si potrà creare una iscrizione con valori non validi. In particolare la durata dell'abbonamento deve essere un numero intero positivo, la percentuale di sconto deve essere ovviamente compresa tra 0 e 100 e il giorno di inizio non può essere precedente al giorno in cui l'iscrizione viene effettuata. Inoltre non sarà possibile creare una iscrizione a tempo ad un corso se un'altra iscrizione relativa a quel medesimo corso risulta ancora attiva nella data di inizio scelta. Bisogna quindi scegliere una data successiva alla fine di quella iscrizione in modo da non duplicare le iscrizioni. Non sarà possibile creare una iscrizione a ingressi se esiste già una iscrizione relativa allo stesso corso. Nella parte di controller oltre alla possibilità di creare, modificare ed eliminare una iscrizione, lato gestore è possibile monitorare le iscrizioni controllando la data di inizio,la data di scadenza, la durata, il prezzo totale, i servizi aggiuntivi associati a quella iscrizione e gli ingressi rimasti. Nella schermata di creazione il bottone di creazione viene abilitato sono nel caso in cui tutti i campi siano riempiti. Il campo relativo alla durata dell'abbonamento accetta solo valori numerici mentre il campo relativo allo sconto che si vuole applicare accetta anche il punto in caso di sconto decimale. Inoltre attraverso il menù è possibile accedere alla schermata di gestione degli ingressi. Tramite questa schermata si possono scalare gli ingressi agli abbonamenti. Basterà inserire il nome utente del cliente a cui si vuole scalare l'ingresso e a quel punto selezionare l'abbonamento. Questa schermata ha lo scopo di simulare il comportamento di un tornello posto all'ingresso della palestra.

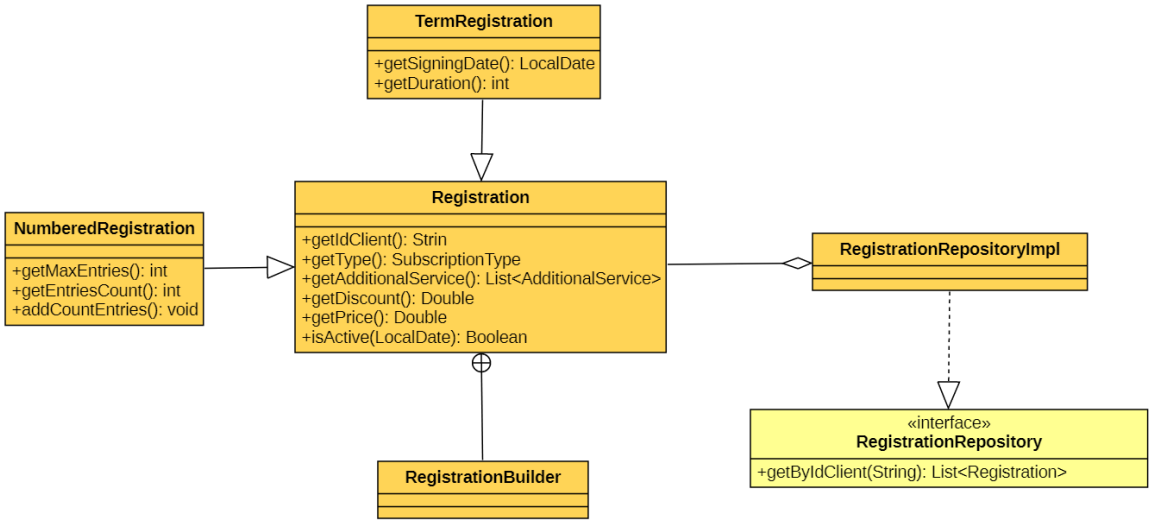

Figura 2.12: Schema UML che mostra il concetto di iscrizione.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il model e tutte le parti non banali e non facenti parte della view sono stati sottoposti a test tramite JUnit. Per alcune classi sono anche stati utilizzati mock (tramite la libreria Mockito) in modo da testarle in isolamento.

In particolare sono stati testati

- Authentication: AuthService e i repository associati;
- MemoryRepository: repository base, esteso da tutti gli altri repository;
- Customer: tutti i repository associati a clienti, abbonamenti e iscrizioni;
- BmiRepository: repository relativo al Bmi personale del cliente;
- ToolsRepository: repository per gli attrezzi della palestra;
- Employee: model e repository relativi agli impiegati e al calendario;
- NavigationService: servizio per la gestione delle pagine e della navigazione tra di esse.

Le view sono state testate solo manualmente, in quanto la complessità è tutta localizzata nel model.

Il software è stato testato e funziona correttamente su Windows e Linux. Non è stato possibile testare su OSX.

3.2 Metodologia di lavoro

La natura del software ci ha permesso una divisione delle aree di sviluppo piuttosto netta, dove ogni componente del gruppo si è occupato di uno o più aspetti della gestione della palestra. Questa separazione si riflette anche nella struttura dei package.

Come primo step, svolto in collaborazione stretta, sono stati analizzati gli aspetti del dominio applicativo e definita la struttura del software e le interfacce principali.

Il **Repository Pattern**, il **Builder Pattern** e la **dependency injection** sono stati adottati da tutti i membri del gruppo al fine di standardizzare la metodologia di sviluppo e rendere più semplice l'integrazione.

Lo sviluppo è cominciato separatamente, partendo dalle classi che implementano le entità, nell'attesa che l'implementazione del repository base e della navigazione fossero ultimate. Inizialmente ogni componente ha utilizzato una propria *Main Class* e ha lavorato effettuando commit nel proprio branch GIT, implementando, qualora possibile, anche view e controller in modo completamente separato dal resto del codice. Una volta ultimata la base del software, che consiste in una *Main Class* comune, *MemoryRepository* e navigazione, ogni componente ha potuto implementare il necessario all'integrazione della propria parte di model e view con lo "scheletro" del software e con la parte prodotta dai colleghi. A questo punto lo sviluppo delle varie feature si è spostato in branch dedicati, uniti periodicamente al branch "develop". L'uso di repository pattern e dependency injection ha reso molto semplice l'integrazione, in quanto è stato sufficiente istanziare gli oggetti condivisi (repository e services) nella *Main Class* e passarli semplicemente al costruttore dei vari controller, anch'essi istanziati nella *Main Class*.

La divisione dei compiti, come già accennato nella sezione di Design di Dettaglio è la seguente:

- **Mari**: Mi sono occupato della gestione dei dipendenti e del loro calendario (package *gymman.employees*) nonché di autenticazione (*gymman.auth*), navigazione (*gymman.ui.navigation*) e delle interfacce base per l'implementazione dei *Repository*.
- **Manzi**: ho realizzato la porzione di Model riguardante le schede della creazione dell'Attrezzo (aggiunta/modifica/eliminazione), la visualizzazione dei dati di un cliente e l'implementazione del BMI (aggiunta/modifica/eliminazione).
- **Manfredonia**: gestione del cliente, degli abbonamenti, dei servizi aggiuntivi e delle iscrizioni(package *gymman.customer*).

3.3 Note di sviluppo

Il goal di questo software è l'estendibilità e l'efficienza del codice. All'applicazione si possono aggiungere nuovi concetti o strumenti senza "sporcare" il codice già implementato. Nello sviluppo dell'applicazione si è cercato di utilizzare, dove possibile, gli elementi avanzati del linguaggio Java visti a lezione, quali lambda expressions e Streams per avere un codice più facilmente comprensibile e semplice da scrivere.

Mattia Mari

Per la realizzazione delle feature a me assegnate ho cercato di utilizzare un approccio *test first* che ha contribuito a diminuire il numero di bug e ha aiutato a rivedere alcune interfacce che da progetto risultavano scomode da utilizzare. Come conseguenza ho dovuto far uso di mock, tramite la libreria Mockito, per testare in isolamento le classi interessate. Oltre all'utilizzo di Optional, Streams e lambda, ho fatto uso di generici per la definizione dell'interfaccia *Repository* che contiene le definizioni di metodi base per lavorare con classi di tipo *Entity*. Si nota anche l'uso di Google Guava per l'hashing delle password e Lombok, sfruttato anche dai miei colleghi, per creare facilmente builders, getters e setters.

Matteo Manzi

Le due parti sviluppate con il Builder Pattern sono Tool e Bmi poichè questa scelta permette di garantire la semplificazione e leggibilità del codice, ove ogni valore è assegnato ad un singolo parametro.

Daniele Manfredonia

Nella parte da me sviluppata ho utilizzato gli Stream e le lambda per semplificare le operazioni sulle collezioni e per rendere il codice più leggibile. Inoltre è stato utile lavorare con gli Optional in molte situazione ad esempio per differenziare il caso di modifica di un'entità da quello di creazione.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Mattia Mari

Tutto sommato sono soddisfatto del software che io e il mio gruppo abbiamo prodotto. Certamente non è da considerarsi un prodotto ben levigato ma, piuttosto, un esercizio di stile che però mi - e ci - ha permesso di scontrarci con tutta una serie di problematiche e sfide legate alla produzione di un software. La difficoltà maggiore che ho riscontrato è stata senza dubbio l'interfacciarsi con persone senza esperienza nello sviluppo in team. Essendo abituato a collaborare in un piccolo team nell'azienda in cui lavoro, mi sono accorto di dover spesso esplicitare dettagli che normalmente avrei dato per scontati.

Matteo Manzi

Lo sviluppo del progetto mi ha permesso di interfacciarmi direttamente con OOP sul lato pratico, comprendendo argomenti che non avevo capito al meglio. Grazie al team sono migliorato a livello di sviluppo del codice e sulla struttura su come scriverlo. E' stata la prima esperienza di lavoro di gruppo ed è risultata molto stimolante e costruttivo poichè in caso di difficoltà son riuscito a confrontarmi con i compagni e comprendere la risoluzione di un problema. Ho intenzione di continuare nello sviluppo del progetto con queste funzionalità:

- *Grafico*: Un grafico per illustrare il miglioramento del peso nel tempo.
- *Calendario Attività*: Aggiungere un calendario dove è possibile segnarsi le attività/corsi svolti o in programma da fare durante la settimana.

Daniele Manfredonia

Sono complessivamente soddisfatto del lavoro svolto per la realizzazione di questo progetto. Per la prima volta mi sono trovato a dover lavorare in team in un progetto per niente banale. La possibilità di collaborare con gli altri membri del gruppo lo ritengo un aspetto positivo per imparare a organizzare al meglio il lavoro. Nello specifico Mari è stato disponibile e di aiuto nel dare consigli su alcune parti. Sicuramente questo progetto mi ha aiutato a comprendere in maniera più specifica il linguaggio di programmazione

object-oriented andando oltre agli scolastici esercizi visti in passato. Le difficoltà maggiori le ho incontrate soprattutto nel dover avere a che fare con argomenti mai visti prima a cui ho dovuto dedicare più tempo. Un aspetto che in futuro si potrebbe implementare per rendere ancora più completa la mia parte è la tessera punti associata a ciascun cliente. In questo modo ogni cliente avrebbe ricevuto punti per ogni iscrizione effettuata e avrebbe potuto usarli per ottenere sconti nelle iscrizioni successive.

Appendice A

Guida utente

Nota: saranno illustrate solo le operazioni non banali.

Simulazione ingresso

- Selezionare dal menù la voce *Ingresso*
- Inserire il nome utente del cliente interessato
- Avviare la ricerca con l'apposito tasto
- Selezionare l'abbonamento a cui si vuole scalare l'ingresso
- Premere sul bottone *Conferma*

Un utente abilitato alla creazione degli Attrezzi può visionare l'elenco di essi nella schermata Attrezzi, nella quale li può modificare/eliminare o creare.

L'utente dopo aver effettuato l'accesso può visualizzare i dettagli personali e abbonamenti attivi con i relativi servizi aggiuntivi. L'utente per tenere traccia del proprio avanzamento, può aggiungere i dati del proprio peso ed altezza per poi calcolare il Bmi e verificare lo stato di sviluppo leggendo la legenda.

Gestione dei ruoli

I ruoli vanno creati prima di poter essere associati ad un impiegato. Cliccando su "Ruoli utente" e successivamente su "Nuovo ruolo" è possibile creare un ruolo. Dopo aver inserito un nome ed eventualmente una descrizione è possibile selezionare uno o più permessi cliccando sul simbolo "+" a destra. Ora è possibile creare un nuovo impiegato cliccando su "Staff" e successivamente su "Nuovo dipendente" e scegliere il ruolo appena creato dal menù a tendina "Ruolo".