



UNIVERSITÀ
DEGLI STUDI
FIRENZE

UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Software per la gestione di abbonamenti e prenotazioni di una palestra

Autore:
Mattia Benincasa

N° Matricola:
7111171

Corso principale:
Ingegneria del software

Docente corso:
Enrico Vicario

Indice

1	Introduzione	2
1.1	Obiettivo del progetto e analisi del dominio applicativo	2
1.2	Attori coinvolti	2
1.3	Architettura software	3
1.4	Strumenti e tecnologie utilizzate nello sviluppo	4
1.5	Github	4
2	Progettazione	5
2.1	Diagramma dei casi d'uso	5
2.2	Templates	5
2.3	Mockups	9
2.4	Progettazione database e schema ER	11
2.5	Organizzazione delle classi e dettagli di progettazione	13
2.5.1	Domain model	14
2.5.2	Business Logic	16
2.5.3	ORM	17
2.5.4	Controllers	18
3	Implementazione	20
3.1	Domain model	20
3.1.1	Bookings	20
3.1.2	DailyEvents	20
3.1.3	DiscountStrategy	21
3.1.4	Membership	22
3.1.5	Users	22
3.2	Business logic	23
3.2.1	Sistema di Autenticazione	24
3.2.2	Prenotazioni e Validazioni	25
3.2.3	Servizi relativi agli acquisti	26
3.3	Controller	27
3.4	DAO	28
4	Testing	29
4.1	Test del domain model	29
4.2	Test della Business Logic	30
4.3	Test dell'ORM	31
4.4	Test d'integrazione	31

Elenco delle figure

1	Diagramma dei package	3
2	Diagramma dei casi d'uso	5
3	Mock-up 1: schermata di prenotazione di una classe per il profilo cliente	9
4	Mock-up 2: schermata di visualizzazione degli appuntamenti per un trainer	10
5	Mock-up 3: schermata di attivazione/acquisto di un abbonamento/bundle per il receptionist	10
6	Mock-up 4: schermata di gestione operazioni CRUD di un ADMIN	11
7	Diagramma ER database	12
8	Diagramma delle classi del domain model	14
9	Diagramma delle classi della business logic	16
10	Diagramma delle classi dell'ORM	17
11	Diagramma delle classi del Controller	18

1 Introduzione

1.1 Obiettivo del progetto e analisi del dominio applicativo

Il progetto ha come obiettivo la realizzazione di un software per la gestione degli abbonamenti e delle prenotazioni di una palestra. La palestra considerata mette a disposizione dei cliente diverse tipologie di piani di abbonamento. Ogni piano di abbonamento prevede l'accesso a diversi tipi di attività che la palestra offre. In particolare si hanno due distinti tipi di abbonamento: l'abbonamento alla sala pesi e l'abbonamento ai corsi con lezioni settimanali. L'abbonamento alla sala pesi prevede l'accesso illimitato alla struttura attrezzata della palestra. Il cliente che dispone di tale piano può richiedere delle consulenze mensili con uno degli istruttori della sala stessa. Il numero di volte per cui si può prenotare l'appuntamento con l'istruttore dipende dalla tipologia di abbonamento sala pesi: se di tipo BASE, il cliente ha a disposizione una prenotazione al mese; se di tipo PERSONAL può invece richiedere supporto tecnico ogni volta che vuole (ha dunque un numero di prenotazioni illimitate al mese). Gli abbonamenti ai corsi prevedono invece la possibilità di partecipare alle lezioni dei corsi che si tengono con cadenza settimanale nella struttura. Ogni lezione è tenuta da uno degli istruttori associati ad un particolare corso. In questo caso, per la partecipazione è richiesta la prenotazione in quanto le lezioni sono aperte ad un numero limitato di clienti. Il numero di volte che un cliente può prenotare è dettato dalla particolare formula di abbonamento attiva. Una volta effettuata la prenotazione, questa può essere eliminata fino ad un'ora prima dell'inizio della lezione stessa. Ad ogni modo, ogni abbonamento è caratterizzato da una durata, una scadenza ed un prezzo specifico. Alcuni abbonamenti possono essere organizzati in dei Bundle in modo da applicare degli sconti sul prezzo totale degli abbonamenti di cui è composto. Relativamente agli sconti, questi possono essere applicati anche agli abbonamenti singoli in caso di offerte speciali quali sconti stagionali, sconti basati su un tipo di cliente (eg. studenti, militari, anziani, staff palestra) o sconti generici percentuali o fissi. Se gli sconti sono indicati come *special offer* allora questi non sono cumulabili. Ad esempio, se si ha uno studente che acquista nel periodo estivo in cui abbiamo un'offerta stagionale un bundle su cui è presente uno sconto di €20, si applica al totale degli abbonamenti del bundle lo sconto di €20 e lo sconto più vantaggioso tra quello relativo allo studente e quello stagionale. Inoltre, per l'accesso alla struttura sportiva, e dunque anche per effettuare le prenotazioni, non basta avere solo l'abbonamento attivo, ma occorre anche pagare una tassa di iscrizione annuale che copre le spese di tesseramento, assicurazione e altre spese relative all'accesso alla struttura. Il pagamento dell'iscrizione ha validità annuale. È inoltre necessario avere un certificato medico attivo che ogni cliente deve registrare sul suo profilo tramite la reception.

1.2 Attori coinvolti

Di seguito si riportano gli attori coinvolti nel dominio del software in questione:

- Cliente: ogni cliente può attivare uno degli abbonamenti della palestra recandosi in reception. Può prenotare delle lezioni dei corsi se iscritto ad uno dei corsi oppure prenotare un appuntamento con uno degli istruttori disponibili della palestra.
- Trainer: i trainer possono essere sia personal trainer della sala pesi sia istruttori di un qualche corso (le due tipologie non si escludono a vicenda). Se sono trainer della sala pesi, possono mettere delle disponibilità settimanali per i clienti. Se sono trainer di corsi, possono essere associati da un gestore ad uno o più corsi a disposizione ed anche ad uno o più classi settimanali. Ogni classe infatti ha come riferimento uno degli istruttori del corso.
- Receptionist: il receptionist è responsabile della registrazione di nuovi clienti e dell'attivazione agli stessi degli abbonamenti
- Gestore: i gestori sono coloro che si occupano di creare/modificare abbonamenti, bundle, sconti e corsi. Possono, ad esempio aggiungere o rimuovere istruttori da un corso, oppure aggiungere o rimuovere uno o più sconti da un certo abbonamento o bundle.

Le azioni degli attori sopra citati sono dettagliate in modo specifico nel diagramma dei casi d'uso riportato in figura 2.

1.3 Architettura software

L'architettura del software consiste di più package, ciascuno dei quali ha una responsabilità ben precisa. Il package **DomainModel** contiene al suo interno i modelli dei dati che rappresentano gli oggetti di interesse del dominio applicativo. Il package **ORM** (Object Relational Mapper) si occupa di mappare ogni oggetto del DomainModel all'interno di un database relazionale, usando il pattern DAO. Ogni DAO espone i metodi principali CRUD e altri metodi utili al fine di garantire persistenza dei dati. Il package **BusinessLogic** contiene tutti i servizi che consentono di manipolare i modelli del DomainModel. In particolare, istanziano oggetti del DomainModel, effettuano su di essi dei cambiamenti di stato e salvano i cambiamenti effettuati nel database mediante i DAO dell'ORM. È inoltre presente un package **Controller**, le cui classi utilizzano i servizi offerti dal livello della BusinessLogic. Si è pensato di realizzare tale livello prendendo come riferimento il pattern *One controller per user type*¹, in modo da realizzare facilmente un semplice meccanismo di implementazione degli use case relativi a ciascuna tipologia di utente. Per assegnare le risorse ai servizi ed iniettare le dipendenze ai controller, si è pensato di realizzare un **ApplicationManager** che implementa proprio le operazioni sopra citate e gestisce le sessioni degli utenti che effettuano login e logout. Dettagli più specifici su come è stato implementato l'ApplicationManager sono presenti nel paragrafo 3.3 dedicato all'implementazione dei controller.

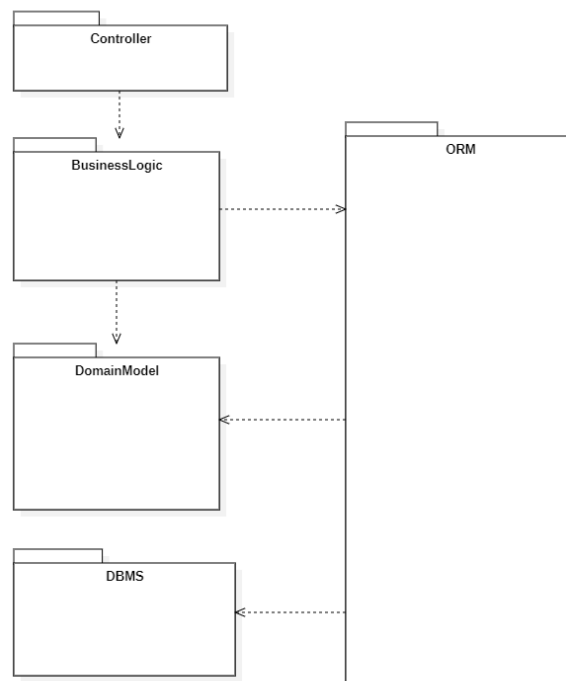


Figura 1: Diagramma dei package

¹In realtà, l'utente Admin dispone di più controller, ciascuno addetto alla gestione di un particolare aspetto del dominio. Dispone dunque di un controller per gestire gli utenti, uno per gestire gli abbonamenti ed un altro per gestire i corsi. Questo si è reso necessario in quanto l'admin presenta un numero elevato di use case diversi in ambiti di dominio anche molto diversi tra loro. In questo modo si è evitato un unico grande controller garantendo una migliore separazione delle responsabilità.

1.4 Strumenti e tecnologie utilizzate nello sviluppo

Il software è stato realizzato interamente in Java SE. Si è implementato un database postgres e si è utilizzato JDBC per la connessione del software con il database. Per il testing si è invece usato il framework JUnit, mentre per la gestione sicura delle password degli utenti si è usato BCrypt. Per la gestione delle dipendenze sopra riportate e del ciclo di build è stato utilizzato Apache Maven. Maven ha permesso di centralizzare la configurazione del progetto nel file pom.xml, semplificando l'integrazione di librerie esterne (come il driver PostgreSQL e BCrypt) e automatizzando i processi di compilazione ed esecuzione dei test. Per la realizzazione dei diagrammi si è usato il software StarUML, mentre per i mock-ups si è usato il software Lunacy.

1.5 Github

Si riporta il link al repository di Github per una visione completa del codice del progetto:
<https://github.com/MattiaBenincasa/GymManagementSystem>

2 Progettazione

In questo capitolo si descrivono i casi d'uso dell'applicativo. Si riporteranno inoltre le scelte progettuali più importanti riguardo l'organizzazione e la modellazione delle classi e dello schema del database.

2.1 Diagramma dei casi d'uso

I casi d'uso sono stati individuati a partire dall'analisi del dominio applicativo. Per ogni attore coinvolto si riportano i casi d'uso raffigurati in figura 2. Ciascuno di essi è preceduto da login (tranne la creazione del profilo cliente) e per ogni utente si considerano sottintesi i casi d'uso basilari che prevedono **cambio password** e **consultazione e modifica dei dati personali** (nome, username, mail...)



Figura 2: Diagramma dei casi d'uso

2.2 Templates

Si riportano i template di alcuni degli use case presenti nel diagramma dei casi d'uso in figura 2. Alcuni template hanno dei riferimenti numerici a dei test presenti nella tabella 12 dei test di integrazione. In alcune post condizioni, invece, è specificato il path di riferimento

dove si è testata quella specifica post condizione. Tutti i test si trovano in `src/test/java`. Per maggiori informazioni sui test si faccia riferimento al capitolo 4 dedicato.

Use Case #1	Prenota una classe (TST 13, MK 1)
Descrizione	Il cliente prenota un posto per una lezione giornaliera disponibile.
Livello	User goal
Attori	Customer
Precondizioni	Il cliente deve avere un abbonamento valido, deve avere l'iscrizione attiva e un certificato medico in corso di validità.
Basic course	<ol style="list-style-type: none"> 1. L'utente visualizza la lista delle lezioni disponibili. 2. Seleziona la lezione desiderata. 3. Il sistema verifica la disponibilità e le scadenze. 4. Il sistema registra la prenotazione.
Alternative course	<p>2a. Se non ci sono posti disponibili, la prenotazione viene rifiutata. (TST path: BusinessLogicTest/validatorsTest/ClassBookingInfoValidatorTest)</p> <p>3a. Se una delle scadenze non è valida, la prenotazione viene rifiutata. (TST path: BusinessLogicTest/validatorsTest/ClassBookingInfoValidatorTest))</p>
Postcondizioni	La prenotazione è registrata nel sistema e visibile all'utente.

Tabella 1: Prenota una classe.

Use Case #2	Prenota un appuntamento con un istruttore (TST 13)
Descrizione	Il cliente prenota un incontro con un istruttore, secondo le regole legate al tipo di abbonamento.
Livello	User goal
Attori	Customer, Trainer
Precondizioni	L'utente deve avere un abbonamento valido e rispettare i requisiti di certificato medico e iscrizione
Basic course	<ol style="list-style-type: none"> 1. L'utente visualizza la lista di disponibilità degli istruttori. 2. Seleziona la disponibilità dell'istruttore desiderato. 3. Il sistema controlla la validità della richiesta. 4. La prenotazione viene registrata.
Alternative course	3a. Se l'utente ha già raggiunto il limite di appuntamenti consentiti o non rispetta i requisiti di certificato medico e/o iscrizione, il sistema rifiuta la prenotazione. (TST path: BusinessLogicTest/validatorsTest/WeightRoomBookingInfoValidatorTest)
Postcondizioni	La prenotazione dell'appuntamento è memorizzata e visibile al cliente e all'istruttore.

Tabella 2: Prenota un appuntamento con un istruttore

Use Case #3	Visualizza prenotazioni effettuate
Descrizione	Il cliente visualizza tutte le proprie prenotazioni, sia per corsi che per appuntamenti con istruttori.
Livello	Summary
Attori	Customer
Precondizioni	L'utente deve avere effettuato una prenotazione
Basic course	1. L'utente accede alla sezione prenotazioni. 2. Il sistema recupera e mostra tutte le prenotazioni.
Alternative course	Nessuna.
Postcondizioni	L'utente ha una panoramica aggiornata delle proprie prenotazioni.

Tabella 3: Visualizza prenotazioni effettuate

Use Case #4	Attiva abbonamento ad un cliente (TST 11, MK 3)
Descrizione	Il receptionist attiva un nuovo abbonamento per un cliente.
Livello	User goal
Attori	Receptionist, Customer
Basic course	1. Il receptionist seleziona il cliente. 2. Sceglie l'abbonamento da attivare. 3. Esegui l'acquisto e registra l'attivazione.
Alternative course	2a. Se pagamento non va a buon fine, il sistema segnala errore.
Postcondizioni	L'abbonamento è attivo per quel cliente nel sistema.

Tabella 4: Attiva abbonamento ad un cliente

Use Case #5	Completa profilo cliente (TST 10)
Descrizione	Il receptionist inserisce i dati anagrafici e di contatto di un cliente.
Livello	User goal
Attori	Receptionist, Customer
Precondizioni	Il cliente deve aver effettuato la creazione preliminare di un account. (TST 9)
Basic course	1. Seleziona il cliente. 2. Inserisce i dati mancanti (nome, cognome, telefono, data di nascita, categoria cliente). 3. Salva le modifiche nel sistema.
Alternative course	2a. Se alcuni dati non sono validi, il sistema mostra un messaggio di errore.
Postcondizioni	Il profilo cliente è completo e aggiornato.

Tabella 5: Completa profilo cliente

Use Case #6	Aggiunge certificato medico al cliente (TST 10)
Descrizione	Il receptionist registra o aggiorna il certificato medico di un cliente.
Livello	User goal
Attori	Receptionist, Customer
Basic course	1. Seleziona il cliente. 2. Inserisce i dettagli del certificato medico (data di emissione, scadenza, è competitivo). 3. Registra l'informazione nel sistema.
Postcondizioni	Il certificato medico è registrato e associato al cliente.

Tabella 6: Aggiungi certificato medico al cliente

Use Case #7	Gestisci pagamento
Descrizione	Il receptionist registra il pagamento per un servizio o abbonamento.
Livello	Function
Attori	Receptionist, Customer
Basic course	1. Seleziona il cliente e il bundle/abbonamento/tassa iscrizione. 2. Scegli modalità di pagamento. 3. Esegui l'operazione.
Alternative course	2a. Se il pagamento non va a buon fine, l'operazione viene annullata.
Postcondizioni	Il pagamento risulta registrato nel sistema.

Tabella 7: Gestisci pagamento

Use Case #8	Vedi appuntamenti con i clienti (MK 2)
Descrizione	Il trainer visualizza la lista degli appuntamenti fissati con i clienti.
Livello	Summary
Attori	Trainer, Customer
Basic course	1. Accede alla sezione appuntamenti. 2. Il sistema mostra gli appuntamenti programmati.
Alternative course	Nessuna.
Postcondizioni	Il trainer conosce gli appuntamenti pianificati.

Tabella 8: Vedi appuntamenti con i clienti

Use Case #9	Annulla una lezione di un corso
Descrizione	Il trainer annulla una lezione programmata per un corso.
Livello	User goal
Attori	Trainer, Customer
Basic course	1. Il trainer seleziona la lezione da annullare. 2. Conferma l'annullamento. 3. Il sistema notifica i clienti iscritti.
Alternative course	1a. Se la lezione inizia fra due ore, il sistema non consente l'annullamento. (TST path: DomainModelTests/DailyClassTest)
Postcondizioni	La lezione è annullata e non più prenotabile.

Tabella 9: Annulla lezione di un corso

Use Case #10	CRUD trainer (TST 3)
Descrizione	L'amministratore può creare, leggere, modificare ed eliminare profili trainer.
Livello	Summary
Attori	Admin, Trainer
Basic course	1. Accede alla sezione gestione trainer. 2. Esegue l'operazione desiderata (create/read/update/delete).
Postcondizioni	La modifica è applicata ai dati dei trainer nel sistema.

Tabella 10: CRUD trainer

Use Case #11	CRUD corsi (TST 5, MK 4)
Descrizione	L'amministratore può creare, leggere, modificare ed eliminare corsi.
Livello	Summary
Attori	Admin, Trainer, Customer
Basic course	1. Accede alla sezione gestione corsi. 2. Esegue l'operazione desiderata (create/read/update/delete).
Postcondizioni	Il corso è aggiornato secondo le modifiche richieste.

Tabella 11: CRUD corsi

2.3 Mockups

In questa sezione si mostrano alcuni possibili mock-ups relativi all'interfaccia grafica, con lo scopo di mostrare le informazioni che dovrebbero apparire in una interfaccia grafica del software realizzato. Si riporta il mock-up di una schermata per ciascuno degli attori dell'applicativo.

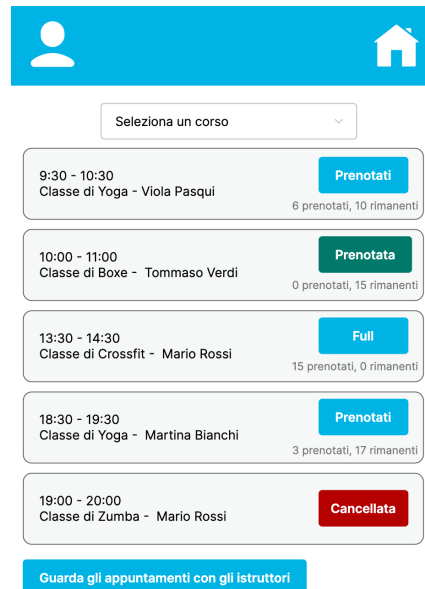


Figura 3: Mock-up 1: schermata di prenotazione di una classe per il profilo cliente



AREA TRAINER



Gli appuntamenti di oggi


9:30 - 10:30 Giuseppe Neri	NOTE PER IL TRAINER: cambio scheda
10:30 - 11:30 Tommaso Verdi	NOTE PER IL TRAINER: Spiegazione esercizi della scheda
15:30 - 17:00 Mario Rossi	NOTE PER IL TRAINER: cambio scheda

Guarda le tue disponibilità

Aggiungi disponibilità

Figura 4: Mock-up 2: schermata di visualizzazione degli appuntamenti per un trainer

STAFF - Profilo receptionist

mario.rossi 

Attiva Abbonamento

Modifica profilo cliente

Vedi il tuo profilo

Seleziona un cliente

Nome cliente

Scegli/Cambia la categoria

Categorie clienti

Seleziona un abbonamento

Abbonamenti disponibili

Scegli data di attivazione

Seleziona un bundle

Bundle disponibili

Scegli data di attivazione

☒

Aggiungi tassa iscrizione

Scegli data di attivazione

Totale per il cliente selezionato: XXX

Sconti applicati: XXX

Procedi al pagamento

Figura 5: Mock-up 3: schermata di attivazione/acquisto di un abbonamento/bundle per il receptionist

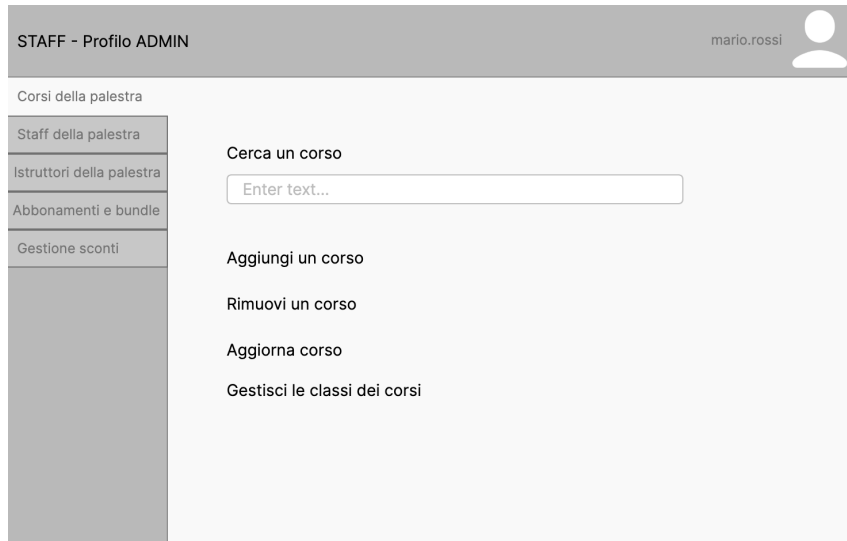


Figura 6: Mock-up 4: schermata di gestione operazioni CRUD di un ADMIN

2.4 Progettazione database e schema ER

La progettazione del database è stata guidata dal diagramma delle classi del *domain model* (figura 8), utilizzato come base per derivare lo schema concettuale e, successivamente, lo schema logico in SQL. In particolare, la presenza di classi astratte e specializzazioni ha portato a scelte progettuali specifiche: per rappresentare le gerarchie di generalizzazione si è adottato il pattern “**table-per-subclass**”, ossia una tabella per la classe base e una tabella per ogni classe derivata, legata alla prima da una relazione 1:1.

Ad esempio:

- La classe astratta **User** è stata mappata nella tabella **User**, contenente tutti gli attributi comuni a qualsiasi utente del sistema (username, password, dati anagrafici, ruolo).
- Le sottoclassi **Customer** e **Trainer** sono state rappresentate rispettivamente nelle tabelle **Customers** e **Trainer**, ciascuna con una relazione 1:1 verso la tabella **User** tramite la chiave primaria **id**, condivisa tra le due.

Questa scelta garantisce che ogni utente, indipendentemente dal ruolo, possieda un identificativo univoco all'interno del sistema. Tale approccio ha fornito vantaggi concreti in più punti del progetto, ad esempio:

- **Gestione semplificata dell'autenticazione:** tutti gli utenti condividono lo stesso spazio di identificativi e le stesse credenziali sono gestite in un unico punto (tabella **User**), evitando duplicazioni.
- **Riferimenti uniformi:** qualsiasi entità che necessiti di collegarsi a un utente può farlo tramite un singolo attributo **user_id**, senza distinzione tra tipi di utente.

Tuttavia, il modello *table-per-subclass* introduce anche alcune complessità:

- **Query più articolate:** per ottenere tutti i dati specifici di un utente è necessario effettuare JOIN tra la tabella **User** e la tabella della sottoclasse corrispondente.
- **DAO più complessi:** i metodi di accesso ai dati devono conoscere la struttura di queste relazioni 1:1 e comporre oggetti a partire da più tabelle, talvolta richiedendo più query o query più complesse.

In altri casi, come per la gestione degli sconti, si è adottata una strategia diversa: tutte le tipologie di sconto sono state memorizzate in un'unica tabella **Discount**, distinguendo il tipo di strategia applicata tramite un attributo **type**. Questa scelta, rispetto alla creazione di una tabella separata per ogni sottoclasse di sconto, ha il vantaggio di semplificare notevolmente le query e le operazioni CRUD, permettendo di gestire l'aggiunta di nuovi tipi di sconto senza modificare la struttura del database. Lo svantaggio principale è che alcune colonne possono rimanere non utilizzate per determinate tipologie di sconto, ma nel complesso la riduzione della complessità di gestione è stata ritenuta prioritaria in questo contesto.

- **Trainer**(id → User(id), isPersonalTrainer, isCourseCoach)
- **TrainerAvailability**(id, trainer_id → Trainer(id), day, startTime, finishTime)
- **Appointment**(trainerAvailability_id → TrainerAvailability(id), customer_id → Customers(id), notes)
- **DailyClass**(id, course_id → Course(id), coach_id → Trainer(id), day, startTime, endTime, maxParticipants)
- **Booking**(customer_id → Customers(id), dailyclass_id → DailyClass(id))
- **MembershipInBundle**(bundle_id → Bundle(id), membership_id → Membership(id))
- **WRMembership**(id → Membership(id), type)
- **CourseMembership**(id → Membership(id), course → Course(id), weeklyAccess)
- **CustomerMembership**(customer_id → Customers(id), membership_id → Membership(id), beginDate, expDate)
- **CustomerMedCertificate**(customer_id → Customers(id), expiryDate, isCompetitive)
- **CustomerFee**(id, customer_id → Customers(id), start_date, expiry_date)
- **CourseTrainer**(course_id → Course(id), trainer_id → Trainer(id))
- **MembershipDiscounts**(membership_id → Membership(id), discount_id → Discounts(id))
- **BundleDiscounts**(bundle_id → Bundle(id), discount_id → Discounts(id))
- **MonthWithDiscounts**(discount_id → Discounts(id), month)

2.5 Organizzazione delle classi e dettagli di progettazione

Di seguito si analizza nel dettaglio, come sono state organizzate le classi nei vari livelli dell'architettura e come queste sono in relazione tra loro. In ogni sezione si farà vedere prima il diagramma UML per una visione d'insieme delle classi e poi si darà una descrizione testuale dello stesso.

2.5.1 Domain model

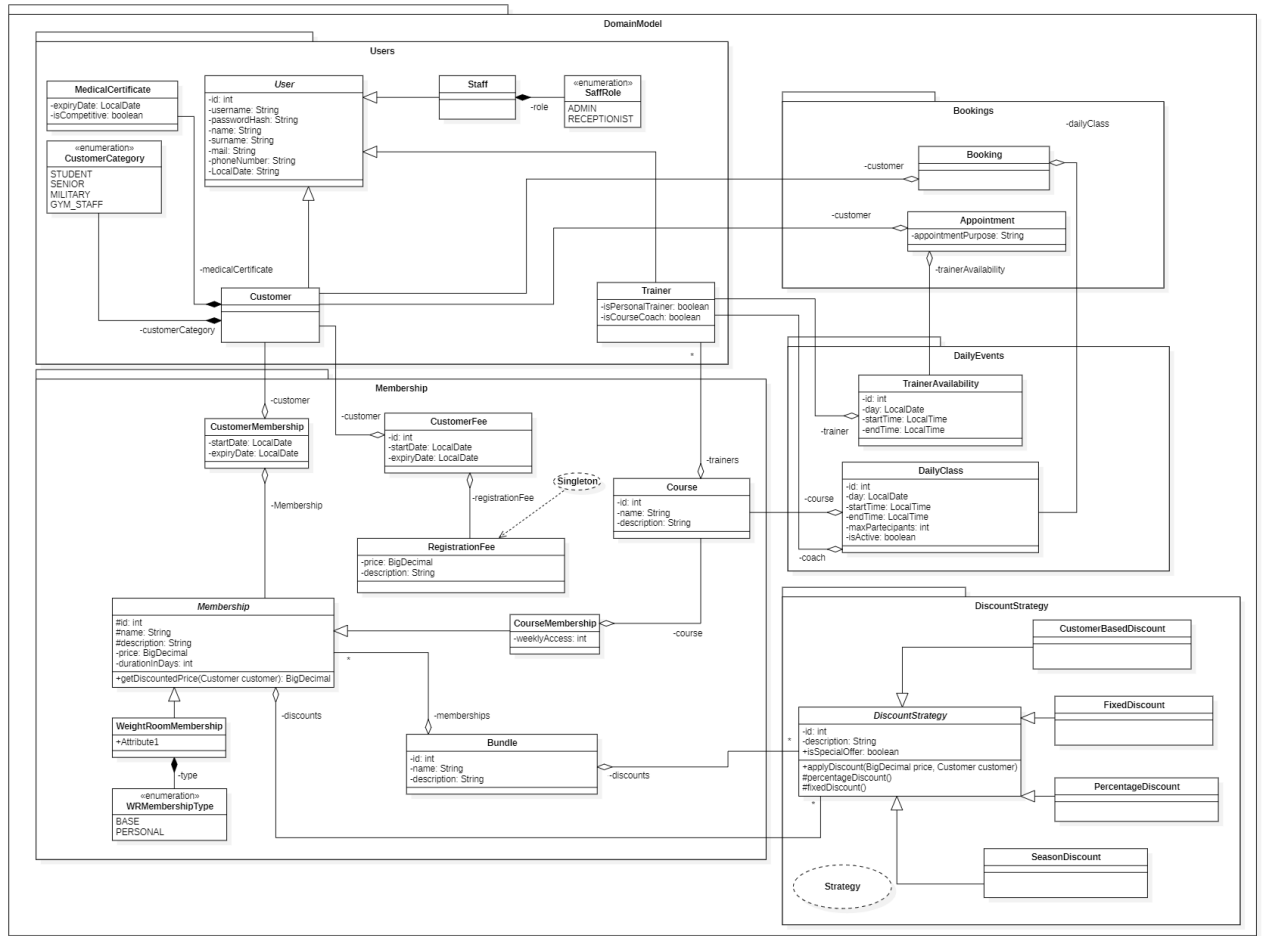


Figura 8: Diagramma delle classi del domain model

Le classi del domain model sono state suddivise nei seguenti packages:

- **Bookings**: contiene le classi relative agli oggetti prenotabili. Sono state realizzate seguendo il pattern Mapper in quanto hanno come responsabilità quella di mappare un certo cliente ad una classe giornaliera di un certo corso, oppure ad un appuntamento con un trainer.
- **DailyEvents**: contiene le classi relative agli eventi giornalieri messi a disposizione dalla palestra (classi giornaliere dei corsi e disponibilità dei trainer della sala pesi). La classe **DailyClass** è stata implementata usando il pattern Java **Builder**. In tal modo si è resa più fluida la creazione di un oggetto di tipo **DailyClass** evitando costruttori con un numero elevato di parametri creando oggetti immutabili. I setters sono stati usati solo per il cambiamento dell'istruttore o per annullare la lezione. Vedere paragrafo 3.1.2 per maggiori dettagli implementativi.
- **DiscountStrategy**: contiene le classi che rappresentano le varie tipologie di sconti che il sistema mette a disposizione. Le classi sono organizzate secondo il design pattern **Strategy** in modo da uniformare le diverse strategie di calcolo dei prezzi di abbonamenti e bundle conservando un'unica interfaccia comune. Esistono 4 tipologie di sconto diverse, sconto percentuale, sconto fisso, sconto percentuale stagionale, sconto percentuale basato su categoria cliente. Il pattern strategy consente anche una semplice

espansione di nuove tipologie di sconto. Bundle e Abbonamenti hanno più strategie di sconto che vengono applicate in modo sequenziale. Qualora ci siano più offerte marcate come *specialOffer*, queste vengono applicate considerando solo quella che porta ad avere il prezzo più basso (sconto più vantaggioso).

- **Membership:** contiene le classi che modellano tutte le tipologie di abbonamenti acquistabili, bundle e tassa di iscrizione. Si è pensato di implementare una classe astratta **Membership** da cui derivano le due diverse tipologie di abbonamento, **CourseMembership** e **WeightRoomMembership**. In questo modo è stato possibile uniformare i dettagli comuni come nome, descrizione, data di scadenza, prezzo e sconti. In particolare, ciò ha facilitato l'assegnazione degli abbonamenti ai clienti che vengono mappati ad un abbonamento (sia esso della sala pesi o di un corso) attraverso la classe **mapper CustomerMembership**. La classe **Bundle** contiene più abbonamenti, ciascuno con la sua durata e le sue caratteristiche. La sua utilità è quella di assegnare ad un insieme di abbonamenti (magari quelli acquistati spesso insieme) degli sconti. Tuttavia, al momento dell'acquisto, questi sono attivati separatamente, portando alla creazione di tanti **customerMembership** quanti sono gli abbonamenti nel bundle. Ultima particolarità di questo package riguarda la tassa di iscrizione che è stata implementata come **singleton** in modo da garantire la sua unicità all'interno del software ed aggiornare, quando necessario, descrizione e prezzo dell'unica istanza presente. Anche in questo caso, la tassa di iscrizione è mappata con il cliente attraverso **CustomerFee** in modo che il cliente possa avere più istanze relative alla tassa di iscrizione facilitando così il rinnovo della stessa.
- **Users:** contiene tutte le classi che rappresentano le diverse tipologie di utente. Come detto in precedenza esistono 4 diversi utenti (cliente, istruttore, receptionist e admin). Per cliente e istruttore si è pensato a due classi distinte, rispettivamente **Customer**, **Trainer** in quanto è presente la necessità di salvare nelle rispettive classi informazioni diverse e specifiche. Mentre per receptionist e admin si è creata una unica classe **Staff** distinguendo i due tipi attraverso una enum. Il ruolo assegnato ad uno staff è immutabile. Le tre classi sono state accomunate da una stessa classe base astratta **User** che permette di raggruppare i dati comuni dei 4 attori in quanto utenti. Ciò permette di avere una interfaccia comune per operazioni come login, logout e cambio di informazioni personali come username, password, mail, numero di telefono etc. Di seguito alcuni dettagli relativi alla modellazione del cliente. Ogni cliente deve avere un certificato medico e può essere assegnato ad una determinata categoria. In questo caso non si è optato per una classe mapper poiché un cliente deve avere un solo certificato medico associato (il rinnovo non viene effettuato dalla palestra, dunque, se un cliente porta un nuovo certificato medico, questo verrà semplicemente sostituito a quello vecchio). Le categorie sono gestite usando una enum **CustomerCategory** i cui elementi implementano ciascuno una diversa strategia di validazione per assegnazione delle categorie ai clienti (maggiori dettagli a riguardo nel paragrafo 3.1.5 relativo all'implementazione).

2.5.2 Business Logic

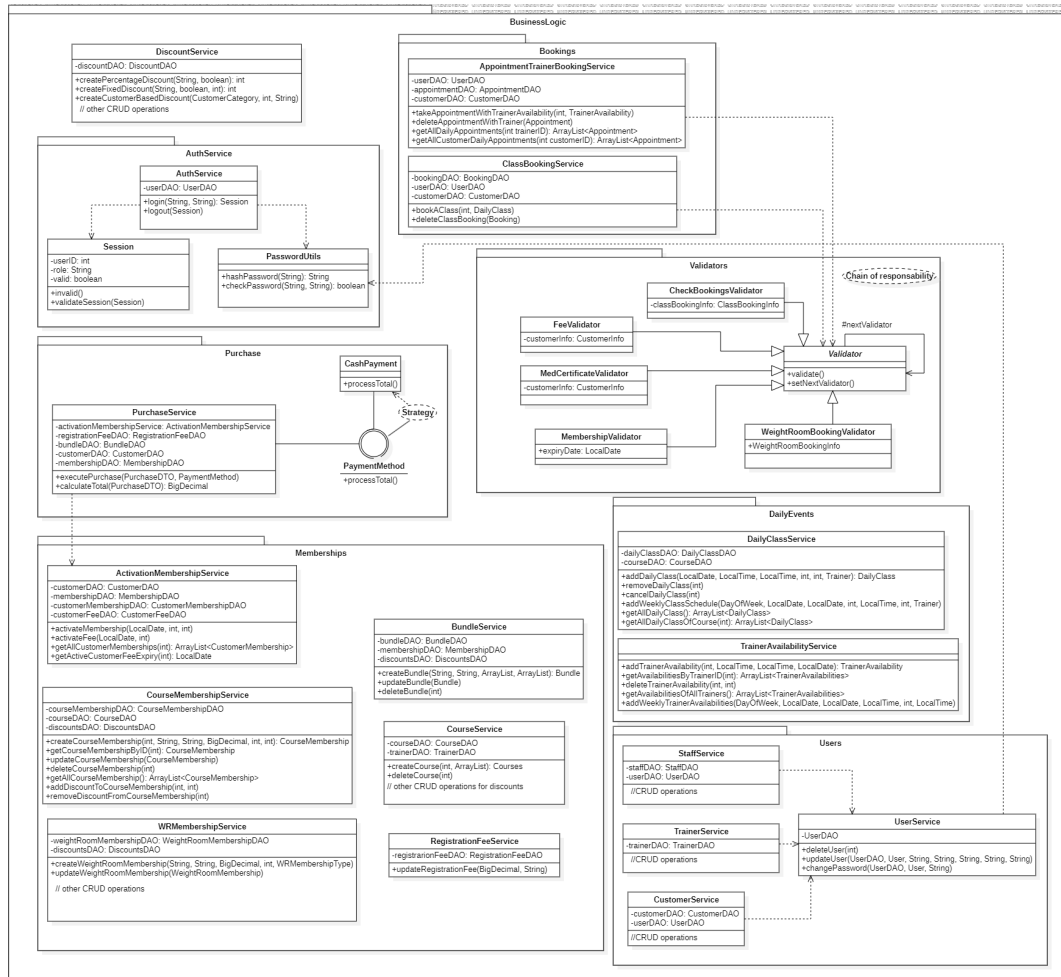


Figura 9: Diagramma delle classi della business logic

All'interno del livello di Business Logic ritroviamo una suddivisione in package simile a quella del domain model, ciascuna che implementa i servizi relativi ai vari elementi del dominio applicativo e che includono operazioni di modifica dello stato di tali elementi. Compiono qua alcuni package relativi esclusivamente alla logica di business. In particolare troviamo il package **AuthService** che gestisce le operazioni di autenticazione dei vari utenti e hashing delle password. Troviamo qua un oggetto di tipo `Session` che contiene l'id e la tipologia di utente autenticato (cliente, receptionist, admin, trainer). Il metodo `login(String username, String plainPassword)` di `AuthService` restituisce un oggetto di tipo `Session` con i dati dell'utente autenticato. Si ha poi il package **Purchase** per l'esecuzione e l'elaborazione degli acquisti (solo simulata). È presente una piccola struttura a Strategy che può essere ampliata in un contesto reale con un vero meccanismo di elaborazione dei pagamenti. L'ultimo package rilevante in termini di dettagli di progettazione è il package **Validators**. Si era presentato il problema della validazione di alcuni requisiti prima di effettuare una prenotazione di una classe o un appuntamento con un istruttore della sala pesi. In particolare, si deve controllare nel seguente ordine che:

- Prenotazione corso:
 1. L'utente abbia pagato la tassa di iscrizione

2. l'utente abbia un certificato medico attivo oppure scaduto da non più di 20 giorni
 3. l'utente abbia l'abbonamento attivo del corso per cui si intende effettuare la prenotazione
 4. il numero di iscritti alla classe sia inferiore del numero massimo di partecipanti
 5. il numero di prenotazioni effettuate dall'utente nella settimana corrente sia inferiore a quelle previste dal suo abbonamento
- Prenotazione appuntamento sala pesi:
 1. L'utente abbia pagato la tassa di iscrizione
 2. l'utente abbia un certificato medico attivo oppure scaduto da non più di 20 giorni
 3. l'utente abbia un abbonamento per la sala pesi attivo
 4. il numero di prenotazioni effettuate dall'utente nel mese corrente sia inferiore a quelle previste dalla tipologia del suo abbonamento sala pesi (BASE prevede un appuntamento al mese, PERSONAL prevede un numero illimitato di appuntamenti al mese)

Si nota dunque che è necessario verificare questi requisiti in tale ordine di priorità e che i due casi condividono alcuni controlli. Si è pensato di risolvere questo problema ricorrendo al design pattern *Chain of Responsibility*. Si è definita una classe base astratta comune **Validator** che espone un metodo **Validate()** con un riferimento ad un altro Validator. In questo modo si facilita la creazione di catene di validatori secondo l'ordine desiderato che è possibile validare chiamando il solo metodo **validate()**. Ciò rende non solo semplice la verifica di più validatori in sequenza in modo chiaro e pulito, ma permette anche di creare nuovi validatori organizzati secondo ordini diversi e con nuove regole semplicemente estendendo la classe **Validator**. Altri dettagli implementativi possono essere trovati nel paragrafo 3.2.2.

2.5.3 ORM

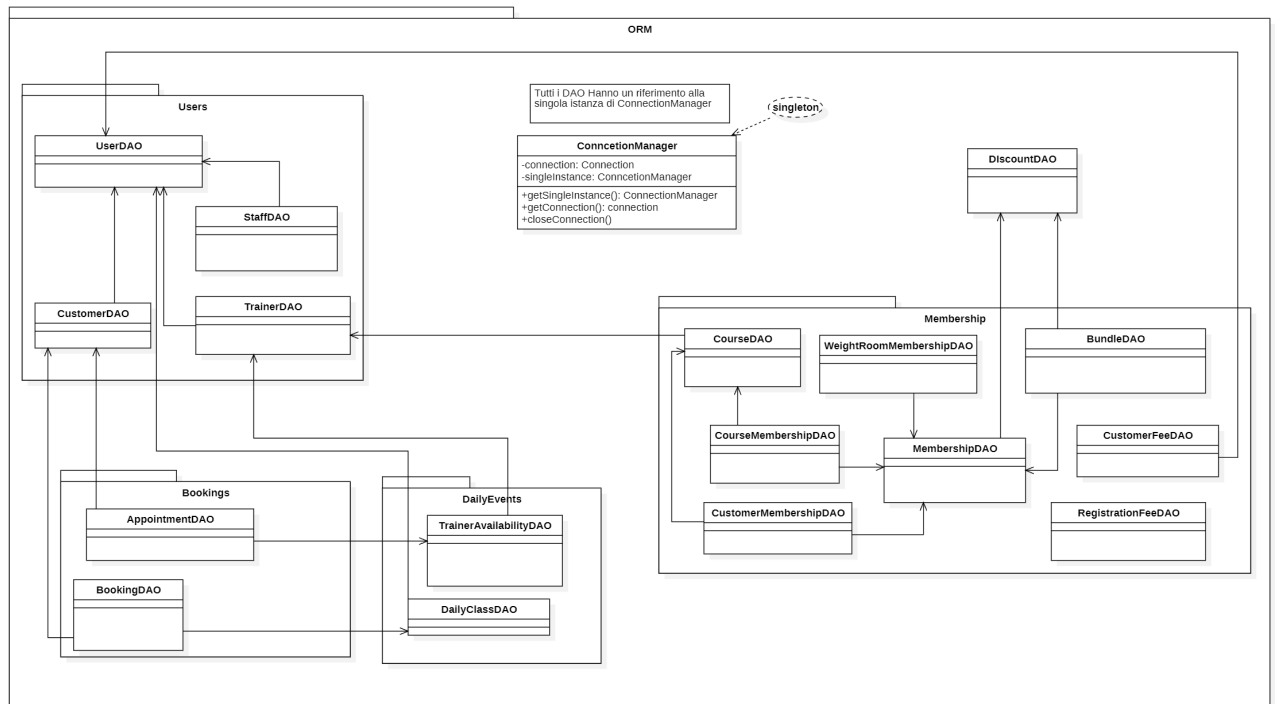


Figura 10: Diagramma delle classi dell'ORM

Come detto in precedenza, il package ORM si occupa della mappatura degli oggetti del domain model nel database relazionale postgres. In questo caso, ogni oggetto è stato mappato usando il pattern architetturale **DAO**. Il Data Access Object (DAO) pattern è un pattern architetturale che separa la logica di accesso ai dati dalla logica applicativa. Attraverso i DAO, tutte le operazioni di lettura e scrittura sul database sono incapsulate in classi dedicate, evitando che il resto dell'applicazione conosca i dettagli di persistenza. Questo approccio ha permesso di mantenere il codice più modulare, facilitare la manutenzione e rendere la logica di business indipendente dalla tecnologia di storage utilizzata. Nel progetto non sono state realizzate interfacce comuni per i DAO, in quanto l'applicazione accede esclusivamente a un database PostgreSQL e non è richiesta, nell'immediato, la sostituzione del sistema di persistenza. Tuttavia, la struttura dei DAO è stata progettata in modo modulare, rendendo possibile l'estensione o l'adattamento del codice a un diverso DBMS con modifiche circoscritte al livello di accesso ai dati. Una ulteriore nota importante in questo contesto riguarda la gestione della connessione con il database che è realizzata nella classe **ConnectionManager**. Tale classe è implementata come **singleton** in modo da garantire una sola istanza di **connectionManager** da cui ottenere la connessione al database. Si garantisce in questo modo una singola connessione aperta al db che viene usata da ogni DAO per effettuare le operazioni sul database.

2.5.4 Controllers

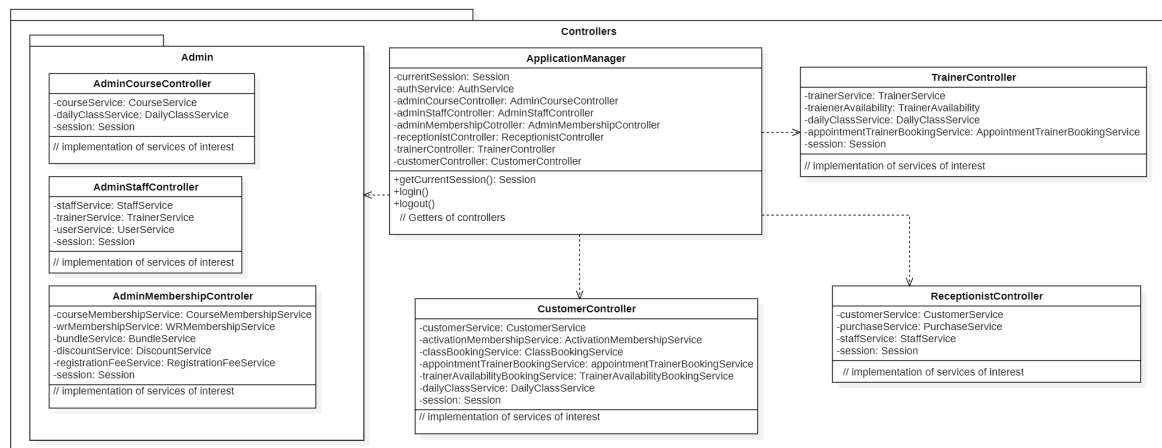


Figura 11: Diagramma delle classi del Controller

I controller costituiscono lo strato di coordinamento tra l'interfaccia di accesso alle funzionalità (nel nostro caso simulata nei test) e la logica di business dell'applicazione. Ogni controller riceve le richieste dell'utente, effettua eventuali controlli preliminari (eg. verifica la validità della sessione) e delega l'esecuzione delle operazioni ai servizi della Business Logic. Per la loro progettazione è stato adottato il pattern **one-controller-per-user**, in cui ogni ruolo del sistema (Customer, Trainer, Receptionist, Admin) dispone di un controller dedicato. Questo approccio consente di raggruppare in modo coerente i casi d'uso e semplificare la gestione dei permessi, poiché ogni controller gestisce esclusivamente le operazioni consentite al ruolo associato. Nel caso dell'Admin, a causa dell'elevato numero di funzionalità e ambiti di gestione (corsi, abbonamenti, personale), si è scelto di suddividere ulteriormente il controller in più unità specializzate: **AdminCourseController**, **AdminMembershipController** e **AdminStaffController**. Questa scelta migliora la leggibilità del codice, favorisce la manutenibilità e riduce la complessità di singoli controller troppo estesi. I controller non contengono logica applicativa complessa: si limitano a orchestrare i vari servizi e a restituire i risultati, mantenendo così una netta separazione tra livello di presentazione e logica di business. In questo caso si è preferito usare il pattern one-controller-per-user rispetto ad altri pattern di

strutturazione dei controller (eg. one-controller-per-page usata principalmente in un contesto web) in quanto il progetto consiste principalmente di una API backend che espone una serie di servizi e funzionalità. Il pattern scelto consente dunque:

- **Gestione semplificata dei permessi** - ogni controller gestisce esclusivamente operazioni consentite al ruolo associato, evitando controlli ridondanti sparsi nel codice.
- **Organizzazione chiara dei casi d'uso** - le funzionalità di uno stesso ruolo sono raccolte in un'unica classe, rendendo più intuitivo il loro reperimento e la loro modifica.
- **Manutenibilità** – riduce la duplicazione di codice e facilita l'aggiunta o rimozione di operazioni legate a un ruolo.

3 Implementazione

Si descrivono ora come sono stati implementati i vari punti individuati in fase di progetto e riportati nella sezione precedente. Una visione completa delle classi si può avere osservando i diagrammi delle classi presenti nel capitolo precedente oppure visionando direttamente il codice al link riportato nel paragrafo 1.5.

3.1 Domain model

Tutte le classi presenti nel package `DomainModel` sono state realizzate con lo scopo di rappresentare nel modo più fedele e pratico le entità principale che sono comparse durante la fase di analisi del progetto. Dove richiesto, si sono anche implementate regole di dominio tali da garantire l'inserimento di valori corretti e consistenti con le entità del dominio applicativo rappresentato. Si dà ora uno sguardo più dettagliato ai vari package del domain model.

3.1.1 Bookings

Qua si trovano principalmente classi con funzione di Mapper che contengono dunque dei riferimenti agli oggetti che mappano più attributi aggiuntivi come delle note per l'istruttore. Per preservare l'integrità e l'immutabilità degli oggetti in Bookings, si è adottata la tecnica della copia difensiva sia in fase di costruzione dell'oggetto, sia nei metodi getter. Questo approccio impedisce che modifiche esterne a Customer o DailyClass possano alterare lo stato interno di Booking senza passare dai metodi di business logic. In particolare, viene creata una nuova istanza di Customer e DailyClass a partire dai dati passati al costruttore, e i metodi getter restituiscono a loro volta copie degli oggetti interni, evitando la fuga di riferimenti mutabili. La copia difensiva è stata applicata dove richiesta e in particolare quando determinati oggetti possiedono liste di riferimenti che vengono restituite da qualche getter o inserite da un setter o dal costruttore.

3.1.2 DailyEvents

Come accennato nel paragrafo 2.5.1, la classe `DailyClass` è stata implementata usando il pattern Java Builder che ha consentito la creazione di oggetti in modo più efficace con controlli direttamente nel metodo `build()` della classe statica `Builder` contenuta all'interno della classe `DailyClass`.

Listing 1: Builder pattern

```
public final class DailyClass {
    private final int id;
    private final LocalDate day;
    //other attributs

    private DailyClass(Builder builder) {
        this.id = builder.id;
        this.day = builder.day;
        //setting other attributs
    }

    //getter methods

    //methods for cancelling class and change traier

    public static class Builder {
        private int id;
        private LocalDate day;
        //other attributs

        public Builder id(int id) {
```

```

        this.id = id;
        return this;
    }

    //other setter methods
    //builder with validation controls
    public DailyClass build() {
        if (day == null || startTime == null || endTime == null || course ==
            null) {
            throw new IllegalStateException("Required fields (day, startTime,
                endTime, course) not set.");
        }
        if (startTime.isAfter(endTime)) {
            throw new IllegalStateException("startTime must be before endTime.");
        }
        if (maxParticipants <= 0) {
            throw new IllegalStateException("maxParticipants must be greater
                than 0.");
        }
        if (!coach.isCourseCoach())
            throw new IllegalStateException("Only course coach can be set as
                coach");
        if (!course.getTrainers().contains(coach))
            throw new IllegalStateException("Only trainers of " +
                this.course.getName() + " can be selected as trainers of this
                class");
        return new DailyClass(this);
    }
}

```

Tale classe presenta un metodo per settare l'attributo `isActive` falso qualora un istruttore volesse cancellare la classe, operazione che può essere effettuata non oltre due ore prima l'inizio della lezione. Altrimenti il metodo lancia l'eccezione custom `LateCancellationException` che estende `RuntimeException`. Questa eccezione è usata spesso, anche nella business logic quando si ha a che fare con servizi che possono essere effettuati non oltre un certo tempo (prevalentemente cancellazioni). Da notare nello snippet riportato, come solo gli allenatori di un certo corso possono essere associati alla classe giornaliera del corso in questione. La costruzione mediante `Build()` garantisce dunque la correttezza dei dati dell'oggetto in costruzione.

3.1.3 DiscountStrategy

Questo è il package che contiene le classi che realizzano il design pattern Strategy per il calcolo dei prezzi. Si ha la classe base astratta `DiscountStrategy` che espone il metodo astratto `applyDiscount(BigDecimal price, Customer customer)`. Ogni classe derivata estenderà tale metodo secondo la sua logica ed effettuerà dei controlli basati sul cliente passato come argomento oppure sul periodo temporale se si tratta di sconto stagionale. Si è preferito l'utilizzo di una classe astratta piuttosto che una interfaccia in quanto, la gerarchia delle classi di sconto condivideva molti attributi. Perciò, si è preferito inserirli in una classe base astratta. Di particolare importanza è il metodo statico che esegue la logica di calcolo dei prezzi che considera la migliore offerta speciale. Si riporta di seguito uno snippet del codice:

Listing 2: Metodo per il calcolo dei prezzi scontati

```

public static BigDecimal totalDiscounted(BigDecimal price,
    ArrayList<DiscountStrategy> discounts, Customer customer) {
    BigDecimal discountedPrice = price;

```

```

Map<Boolean, List<DiscountStrategy>> partitionedDiscounts =
    discounts.stream()
        .collect(Collectors.partitioningBy(DiscountStrategy::isSpecialOffer));

List<DiscountStrategy> specialOffers = partitionedDiscounts.get(true);
List<DiscountStrategy> nonSpecialOffers = partitionedDiscounts.get(false);

//apply all nonSpecial offer discounts -> they are cumulative

for (DiscountStrategy discount : nonSpecialOffers)
    discountedPrice = discount.applyDiscount(discountedPrice, customer);

Map<DiscountStrategy, BigDecimal> specialDiscounts = new HashMap<>();

//get the best specialOffer discount
for (DiscountStrategy discount : specialOffers)
    specialDiscounts.put(discount, discount.applyDiscount(discountedPrice,
        customer));

if (specialDiscounts.isEmpty())
    return discountedPrice;
return specialDiscounts.values().stream().min(BigDecimal::compareTo);
}

```

Il metodo `totalDiscounted()` ha il compito di calcolare il prezzo finale di un prodotto applicando una lista di strategie di sconto. Il comportamento implementato si basa su una distinzione tra sconti ordinari (non speciali) e sconti speciali, *specialOffer*, ognuno dei quali viene gestito con una logica differente. La lista di sconti passata come parametro viene suddivisa in due insiemi distinti utilizzando lo stream API di Java e il metodo `Collectors.partitioningBy(DiscountStrategy::isSpecialOffer)`. Gli sconti per i quali il metodo `isSpecialOffer()` restituisce `false` vengono classificati come ordinari. Gli sconti per i quali `isSpecialOffer()` restituisce `true` vengono classificati come speciali. Gli sconti ordinari vengono applicati tutti in sequenza (approccio cumulativo). Si parte dal prezzo iniziale (`price`) e per ciascun sconto ordinario si invoca il metodo `applyDiscount`, aggiornando il prezzo a ogni iterazione. In questo modo, ogni sconto successivo viene calcolato sul prezzo già scontato dal precedente. Gli sconti speciali, a differenza di quelli ordinari, non sono cumulativi. Ognuno di essi viene applicato separatamente al prezzo già ridotto dagli sconti ordinari. I risultati vengono memorizzati in una mappa che associa ogni strategia al prezzo risultante dalla sua applicazione. Se non ci sono sconti speciali, il metodo restituisce direttamente il prezzo calcolato dopo l'applicazione degli sconti ordinari. Se invece ci sono offerte speciali, viene selezionata quella che produce il prezzo più basso (utilizzando `min` su `specialDiscounts.values()` con comparazione tramite `BigDecimal::compareTo`). Il metodo restituisce il prezzo più conveniente ottenuto dalle regole precedenti. In caso di assenza di sconti, il prezzo finale coincide con quello iniziale.

3.1.4 Membership

Il package `Membership` contiene tutte le classi che rappresentano i vari abbonamenti, bundle di abbonamenti, i corsi e la tassa di iscrizione implementata come singleton. Sono presenti anche le classi mapper immutabili che mappano gli utenti ad un abbonamento per `CustomerMembership` e alla tassa d'iscrizione `CustomerFee`. In queste classi si tiene traccia sia della data di attivazione, sia della data di scadenza per facilitare i controlli nella business logic.

3.1.5 Users

Le classi relative alle varie tipologie di utente estendono tutte la classe base astratta utente per i motivi espressi nel paragrafo 2.5.1. Di particolare importanza in termini di implemen-

tazione è l' enum `CustomerCategory` implementata come segue.

Listing 3: Enum per categorie utente

```
public enum CustomerCategory {
    STUDENT {
        @Override
        public boolean isValidFor(Customer customer) {
            int age = Period.between(customer.getBirthDate(),
                LocalDate.now()).getYears();
            return age <= 25;
        }
    },

    //...

    MILITARY {
        @Override
        public boolean isValidFor(Customer customer) {
            return true;
        }
    },

    //...

    public abstract boolean isValidFor(Customer customer);
}
```

In questo caso, nelle categorie in cui è possibile effettuare un controllo esplicito sul qualche attributo del cliente (come nel caso di `STUDENT` che può essere assegnato solo ai clienti che hanno età uguale o inferiore a 25 anni) è stato fatto un controllo più dettagliato. Per altre categorie in cui ciò non è possibile si restituisce un semplice `true` lasciando il controllo dell'assegnazione al receptionist che può assegnare una sola categoria per cliente. Le categorie possono essere espanse facilmente e servono esclusivamente per le offerte. Quando un cliente vuole effettuare un acquisto di qualche abbonamento o bundle, se sull'abbonamento o bundle è presente un qualche sconto basato sulla categoria del cliente, questo è automaticamente verificato e applicato all'abbonamento stesso passando il cliente che effettua l'acquisto come argomento al metodo `getDiscountedPrice(Customer customer)` dell'abbonamento che si vuole acquistare. Ovviamente, è necessario che il receptionist assegni prima la categoria corretta al cliente.

3.2 Business logic

Come già detto nei paragrafi precedenti, in questo livello le classi implementano i vari servizi relativi ai diversi elementi del domain model. Ritroviamo per quasi tutte le classi una struttura ricorrente: si passano i DAO di interesse al costruttore della classe che implementa i servizi. Ogni servizio/metodo userà le funzionalità messe a disposizione dal DAO o per il salvataggio di oggetti creati all'interno del metodo o per l'aggiornamento di oggetti già creati e passati come argomenti al metodo oppure per ottenere, dopo un'interrogazione da parte del DAO al database, uno o più oggetti già inizializzati con i corretti parametri. Si tende a passare ai vari metodi della business logic, identificativi o stringhe, quando non è strettamente necessario passare un oggetto, in modo da gestire il recupero dell'oggetto e la sua costruzione direttamente all'interno del metodo del servizio. In altri casi, come nel caso dell'aggiornamento dello stato di un oggetto, si tende a passare l'oggetto intero in modo da persistere il suo cambiamento di stato direttamente sul database usando il metodo del DAO pensato per la realizzazione di tale operazione. Di seguito si concentra l'attenzione sui dettagli implementativi di maggior interesse.

3.2.1 Sistema di Autenticazione

Il sistema di autenticazione è stato implementato tramite la classe `AuthService`, responsabile della gestione delle operazioni di login e logout. Il servizio riceve in input nel costruttore un'istanza di `UserDAO`, utilizzata per recuperare dal database le informazioni necessarie all'autenticazione: hash della password, ruolo e identificativo dell'utente.

L'operazione di login segue il seguente flusso:

1. Recupero dell'hash della password dal database tramite `UserDAO`, a partire dallo username fornito.
2. Verifica della password in chiaro tramite la classe `PasswordUtils`, che utilizza la libreria `BCrypt` per confrontare l'hash memorizzato con quello generato dalla password fornita.
3. In caso di autenticazione riuscita, viene creata una nuova istanza di `Session`, che incapsula l'identificativo dell'utente e il relativo ruolo.
4. In caso di errore (username inesistente o password non valida), viene lanciata un'eccezione di tipo `AuthenticationException`.

Listing 4: Metodo login di `AuthService`

```
public Session login(String username, String plainPassword)
    throws AuthenticationException {
    try {
        String hashPassword = userDAO.getHashPasswordFromUsername(username);
        if (!PasswordUtils.checkPassword(plainPassword, hashPassword))
            throw new AuthenticationException("Invalid password");

        String role = userDAO.getRoleFromUsername(username);
        int id = userDAO.getIdFromUsername(username);
        return new Session(id, role);
    } catch (DAOException e) {
        throw new AuthenticationException("Invalid username");
    }
}
```

La classe `Session` è responsabile della gestione dello stato di autenticazione di un utente. Contiene:

- l'identificativo univoco (`userID`);
- il ruolo (`role`);
- un flag booleano `valid` per rappresentare lo stato della sessione.

Listing 5: Classe `Session`

```
public class Session {
    private final int userID;
    private final String role;
    private boolean valid;

    public Session(int userID, String role) {
        this.userID = userID;
        this.role = role;
        this.valid = true;
    }

    public boolean isValid() { return valid; }
}
```

```

    public void invalid() { this.valid = false; }

    public static void validateSession(Session session) {
        if (session == null || !session.isValid())
            throw new InvalidSessionException("Current session is invalid");
    }
}

```

Il metodo `logout()` di `AuthService` invalida la sessione corrente impostando il flag `valid` a `false`. Questo approccio consente di centralizzare la validazione tramite `Session.validateSession()`, garantendo un controllo uniforme in tutta l'applicazione. Tale metodo sarà poi usato dai vari controller prima di eseguire ogni loro metodo. L'architettura scelta permette di:

- separare la logica di autenticazione dal resto dell'applicazione;
- proteggere le password con un hashing robusto;
- gestire in modo esplicito lo stato delle sessioni, facilitando future estensioni verso un sistema multiutente.

3.2.2 Prenotazioni e Validazioni

Il sistema di prenotazione delle lezioni e degli appuntamenti richiede una serie di controlli preliminari per garantire che il cliente soddisfi tutte le condizioni necessarie (quota di iscrizione attiva, certificato medico valido, abbonamento attivo, limiti di prenotazioni non superati, disponibilità posti, ecc.). Per gestire in modo modulare ed estendibile tali controlli, è stato adottato il **pattern Chain of Responsibility**, che permette di concatenare più validatori, ciascuno responsabile di una singola verifica.

Poiché la validazione richiede un grande numero di informazioni che spesso coinvolgono operazioni di conteggio nel database, si è pensato di recuperare le informazioni necessarie in un'unica query (o poche query mirate), costruendo oggetti DTO (`CustomerInfo`, `ClassBookingInfo`) che racchiudono tutti i dati utili ai controlli. In tal modo si evitano interrogazioni successive al database da parte dei vari DAO corrispondenti ai vari dati. Ogni validatore riceve nel costruttore unicamente l'oggetto DTO o il campo di interesse, in modo da operare solo sui dati rilevanti senza effettuare ulteriori query. I dati vengono dunque estratti esternamente al validatore che si occupa di implementare esclusivamente la logica di validazione.

L'implementazione base del validatore astratto è la seguente:

Listing 6: Classe astratta Validator

```

public abstract class Validator {
    protected Validator nextValidator;

    public Validator setNextValidator(Validator nextValidator) {
        this.nextValidator = nextValidator;
        return nextValidator;
    }

    public void validate() {
        if (this.nextValidator != null)
            this.nextValidator.validate();
    }
}

```

Ogni validatore concreto estende `Validator` e ridefinisce il metodo `validate()` per eseguire il proprio controllo; se il controllo fallisce, viene sollevata un'eccezione specifica, interrompendo la catena.

Un esempio di utilizzo per la validazione di una prenotazione di una lezione:

Listing 7: Esempio di concatenazione dei validatori

```
public void bookAClass(int customerID, DailyClass dailyClass) {

    //retriving validators info and create DTOs customerInfo e classBookingInfo

    Validator classBookingValidators = new FeeValidator(customerInfo)
        .setNextValidator(new MedCertificateValidator(customerInfo))
        .setNextValidator(new
            MembershipValidator(classBookingInfo.getMembershipExpiry()))
        .setNextValidator(new CheckBookingsValidator(classBookingInfo));

    classBookingValidators.validate();

    // create Booking object and persist it with BookingDAO
}
```

In questo esempio:

- `FeeValidator` controlla la validità della quota di iscrizione.
- `MedCertificateValidator` verifica la validità del certificato medico.
- `MembershipValidator` controlla la scadenza dell'abbonamento.
- `CheckBookingsValidator` assicura che non siano stati superati i limiti di prenotazioni e che ci siano posti disponibili.

Questo approccio garantisce:

- modularità, grazie alla separazione dei controlli in classi indipendenti;
- facilità di estensione, potendo aggiungere nuovi validatori senza modificare gli altri;
- efficienza negli accessi al database, poiché tutti i dati necessari vengono caricati una sola volta prima dell'esecuzione della catena.

3.2.3 Servizi relativi agli acquisti

Le classi che contengono la logica di esecuzione degli acquisti sono contenute nel package della business logic `Purchase`. La classe principale che esegue l'acquisto è `PurchaseService` che ha un metodo `executePurchase(PurchaseDTO purchaseDTO, PaymentMethod paymentMethod)` che svolge due funzioni consecutive: il processamento dell'acquisto e l'attivazione di tutti gli abbonamenti e/o tassa di iscrizione. Per fornire al servizio le informazioni riguardo quali abbonamenti attivare, quando attivarli e a quale cliente attivarli, si utilizzano due DTO:

- `PurchaseItemDTO` che contiene identificativi e data di attivazione dei bundle o abbonamenti che si vogliono attivare;
- `PurchaseDTO` che contiene una lista di `PurchaseItemDTO` insieme ad altre informazioni riguardo l'utente e l'acquisto della tassa di attivazione.

Si è pensato di utilizzare un approccio basato su DTO in modo da facilitare il trasferimento di informazioni relative all'acquisto dal frontend (ad esempio riempiendo i dati del DTO per mezzo di un form, si veda il mockup 3 in fig. 5) al backend, in particolare al servizio di acquisto. Qua, successivamente, usando i metodi dei DAO, si ricostruiscono gli oggetti `Bundle` e `Membership` corretti che contengono tutte le informazioni relative alle strategie di sconto associate a tali oggetti per garantire il corretto calcolo del prezzo. L'esecuzione dell'acquisto, in questo caso, è solo simulata. Tuttavia, si è fornita un'interfaccia `PaymentMethod` con un metodo astratto `processTotal(BigDecimal total)` in modo da creare, in un contesto reale, strategie diverse per eseguire la transazione di denaro. Terminato il processamento dell'acquisto, la funzione `executePurchase(PurchaseDTO purchaseDTO, PaymentMethod paymentMethod)` stessa provvede all'attivazione di tutti gli oggetti (abbonamenti o tassa di iscrizione) acquistati dal cliente usando il servizio `ActivationMembershipService`.

3.3 Controller

I controller rappresentano il punto di accesso principale ai casi d'uso degli attori dell'applicazione. Ciascun controller è dedicato a un ruolo specifico (tranne l'admin che ne ha 3) e racchiude i metodi necessari per svolgere le operazioni consentite a quel ruolo. Possiedono inoltre un riferimento ad un oggetto di tipo `Session` che viene impostato nell'`ApplicationManager`, in particolare nei metodi getter che consentono di prendere i vari controller (vedere Listing 9). Prima dell'esecuzione di ogni operazione, i controller effettuano un controllo sulla validità della sessione corrente per assicurarsi che l'utente sia autenticato e la sessione sia valida. Quando si effettua il logout, il metodo `logout()` dell'application manager imposta la sessione corrente non valida passandola al metodo `logout(Session session)` dell'`AuthService`.

Un esempio di validazione della sessione in un metodo del controller è il seguente:

Listing 8: Validazione della sessione in un metodo del controller

```
public void changePersonalInfo(String username, String name, String surname, String
    mail, String phoneNumber) {
    Session.validateSession(this.session);
    trainerService.changeTrainerInfo(this.session.getUserID(), username, name,
        surname, mail, phoneNumber);
}

public void changePassword(String oldPassword, String newPassword) {
    Session.validateSession(this.session);
    trainerService.changePassword(this.session.getUserID(), oldPassword,
        newPassword);
}
```

Un ulteriore dettaglio che è possibile osservare da questo esempio è che l'utente autenticato possa modificare solo le proprie informazioni personali, poiché il metodo passa automaticamente al servizio l'ID dell'utente contenuto nella sessione corrente. In generale, questo approccio è stato usato in tutti quei metodi responsabili della modifica e consultazione di informazioni personali. Come già anticipato, l'accesso ai controller avviene attraverso l'`ApplicationManager`, che funge da **iniettore di dipendenze**: istanzia tutti i DAO, i servizi e i controller, e gestisce la sessione corrente. Prima di restituire un controller, l'`ApplicationManager` verifica il ruolo dell'utente, garantendo che un utente non possa accedere a controller di altri ruoli. Ad esempio, per il `TrainerController`:

Listing 9: Getter di un controller con validazione del ruolo

```
public TrainerController getTrainerController() {
    if (this.currentSession == null || !this.currentSession.isValid())
        throw new InvalidSessionException("Current session is invalid");

    if (!Objects.equals(this.currentSession.getRole(), "TRAINER"))
        throw new UnauthorizedException("You are not authorized to get this
            controller");

    this.trainerController.setSession(this.currentSession);
    return this.trainerController;
}
```

Questo semplice approccio garantisce che ogni operazione sia eseguita solo da utenti autenticati e autorizzati, mantenendo l'integrità e la sicurezza dell'applicazione. In future estensioni del progetto, l'`ApplicationManager` potrebbe essere esteso per gestire un *pool* di sessioni, permettendo l'utilizzo concorrente del sistema da parte di più utenti e garantendo così una maggiore scalabilità della piattaforma.

3.4 DAO

Il package `ORM` contiene i DAO i cui dettagli progettuali sono stati definiti nel paragrafo 2.5.3. Ogni DAO contiene un riferimento ad un oggetto di tipo `Connection` (proveniente da `java.sql.Connection`) che viene assegnato nei costruttori di tutti i DAO mediante la chiamata al metodo `ConnectionManager.getInstance().getConnection()` che restituisce la connessione al database della singola istanza `ConnectionManager` che stabilisce l'unica connessione al database di interesse. I metodi del DAO sono stati implementati in modo da mappare gli attributi degli oggetti nelle corrispondenti tabelle del database relazionale. I metodi `create` e `update` prendono infatti interi oggetti già correttamente costruiti in modo da effettuare il mapping nel database, mentre i metodi addetti all'estrazione di oggetti dal database o cancellazione di questi (comunque oggetti già mappati) prendono come argomento solo l'ID d'interesse. Molto spesso le classi DAO mantengono riferimenti ad altri DAO per consentire la costruzione completa e coerente di oggetti composti. Ad esempio, la classe `MembershipDAO` contiene un riferimento a `DiscountDAO` affinché, al momento dell'estrazione di un oggetto `Membership` dal database, questo venga restituito con l'elenco degli sconti associati. Tale approccio privilegia la coerenza e la correttezza dei dati estratti, a discapito di una lieve riduzione dell'efficienza, dovuta alla necessità di eseguire più query o operazioni di `join` per comporre l'oggetto completo. Nel contesto di questo progetto, si è scelto di privilegiare la coerenza dei dati rispetto all'ottimizzazione estrema delle prestazioni, poiché la completezza delle informazioni è risultata prioritaria per garantire l'affidabilità delle operazioni di business logic che richiedono oggetti correttamente costruiti.

4 Testing

Nella creazione dei test si è cercato quanto più possibile di verificare il corretto funzionamento di tutte le componenti del software, concentrando l'attenzione su quei componenti e logiche critiche dal cui buon funzionamento dipende l'intero software realizzato. Sono presenti sia classi che testano singole componenti (Unit testing) sia classi che testano più componenti insieme (Test di integrazione). La strutturazione dei package dove sono contenuti i test rispecchia la suddivisione nei diversi livelli architetturali domain model, business logic, ORM. Inoltre, sono presenti due package aggiuntivi, `TestUtils` e `IntegrationTest`. Nel primo si trova una funzione `resetDatabase()` di fondamentale importanza nei test che prevedono operazioni dei DAO o servizi della Business logic. Questa infatti esegue un reset dei dati contenuti in tutte le tabelle del database comprese le sequenze dei contatori per gli identificativi. Il suo utilizzo prima dell'esecuzione dei test e dopo l'esecuzione dei test garantisce che ciascuno di essi possa essere eseguito partendo da condizioni iniziali prevedibili. In generale, si è cercato di mantenere quanto più possibile i test isolati l'uno dall'altro.

`IntegrationTest` contiene una classe con una serie di metodi che simulano un utilizzo sequenziale delle funzionalità più rilevanti dei vari controller disponibili. Si è pensato di realizzare tale test anche e soprattutto per mancanza di interfaccia grafica in modo da simulare un normale flusso di esecuzione del software stesso.

Tutti i test sono presenti nella directory `src/test/java/` e possono essere visionati ed eseguiti per verificarne il corretto esito.

4.1 Test del domain model

I test relativi al domain model riguardano principalmente il testing di regole di dominio che sono incorporate all'interno dei modelli stessi. Di particolare interesse sono i test legati ai pattern usati. A tal proposito, si è provveduto al testing del pattern Builder della classe `DailyClass` per vedere se le condizioni riguardo l'assegnazione degli istruttori di una classe giornaliera specificate nel metodo `Build()` fossero corrette e se fosse lanciata l'eccezione corretta al momento di una cancellazione in ritardo di una classe giornaliera. Ancora più importante è il testing del `DiscountStrategy` in cui si è non solo controllato che i metodi calcolassero corretti sconti percentuali o fissi a seconda del particolare tipo di strategia istanziata, ma si sono create alcune possibili situazioni di acquisto per vedere se la logica dell'applicazione dello sconto non cumulabile (offerta speciale) più favorevole venisse applicata senza errori. Ecco di seguito un esempio.

Listing 10: Test di un `discountStrategy`

```
//...

@BeforeEach
public void Setup() {
    this.membership = new WeightRoomMembership();
    this.membership.setPrice(new BigDecimal("350.00"));
}

//...

@Test
public void testMultipleDiscount() {
    /*
     * a student wants to buy a membership with both seasonal discount
     * and student discount in a winter month. Only the best discount
     * should be applied because they are both special offers.
     */

    Customer student = new Customer();
    student.setBirthDate(LocalDate.of(2003, 2, 18));
```

```

student.setCustomerCategory(CustomerCategory.STUDENT);

HashSet<Month> winterMonthsWithDiscount = new HashSet<>();
winterMonthsWithDiscount.add(Month.SEPTEMBER);
winterMonthsWithDiscount.add(Month.JANUARY);

Clock fixedClock = Clock.fixed(
    LocalDate.of(2025, 1,
        15).atStartOfDay(ZoneId.systemDefault()).toInstant(),
    ZoneId.systemDefault());
SeasonDiscount winterSeasonDiscount = new
    SeasonDiscount(winterMonthsWithDiscount, 10, "Winter season discounts",
        fixedClock);
CustomerBasedDiscount studentDiscount = new
    CustomerBasedDiscount(CustomerCategory.STUDENT, 20, "discount for
        students");

this.membership.addDiscount(winterSeasonDiscount);
this.membership.addDiscount(studentDiscount);

Assertions.assertEquals(new BigDecimal("280.00"),
    this.membership.getDiscountedPrice(student));

}

//...

```

4.2 Test della Business Logic

I test di questo package coprono i servizi della business logic, in particolare quei test che prevedono delle logiche un po' più articolate rispetto ad una semplice chiamata ad una operazione di un DAO per salvataggio o modifica di un oggetto. Si hanno dunque test per il servizio di autenticazione `AuthService` dove si testano successi, insuccessi di login (con password e/o username corretti/errati) e che la sessione contenga i ruoli corretti a seconda di chi ha effettuato il login. Sono stati effettuati poi test relativi all'acquisto (`PurchaseTest`) in cui si testa se, dopo l'esecuzione di un acquisto, vengono effettivamente attivati gli abbonamenti al corretto cliente con le corrette date di attivazione. Un po' più articolati sono i test relativi ai validatori delle prenotazioni in cui si è cercato di coprire un po' tutti i casi in cui una prenotazione possa fallire andando a creare appositi DTO più o meno fallimentari. I test hanno coperto sia casi di validatori isolati, come il seguente:

Listing 11: Test di un validatore singolo

```

//...
@Test
void testInvalid_ClassIsFull() {
    ClassBookingInfo classBookingInfo = new ClassBookingInfo.Builder()
        .membershipExpiry(LocalDate.now().plusYears(1))
        .classMaxParticipants(20)
        .totalBookingsInClass(20)
        .weeklyBookingLimit(2)
        .bookingsDoneByCustomerThisWeek(1)
        .build();

    Validator validatorChain = new CheckBookingsValidator(classBookingInfo);

    assertThrows(ValidatorException.class, validatorChain::validate);
}

```

```
//...
```

Sia validatori multipli incatenati, come il seguente:

Listing 12: Test di un validatore singolo

```
@Test
void testValidUserInformation() {
    //no med cert but sub 19 days ago
    LocalDate feeBegin_1 = LocalDate.now().minusDays(19);
    LocalDate feeExpiry_1 = feeBegin_1.plusYears(1);
    CustomerInfo customerInfo_1 = new CustomerInfo(null, feeBegin_1,
        feeExpiry_1);

    Validator firstValidator = new FeeValidator(customerInfo_1)
        .setNextValidator(new MedCertificateValidator(customerInfo_1));

    assertDoesNotThrow(firstValidator::validate);

    //with med. cert. but it is expired 10 days ago
    LocalDate medCertExpiry_2 = LocalDate.now().minusDays(10);
    LocalDate feeBegin_2 = LocalDate.now().minusDays(50);
    LocalDate feeExpiry_2 = feeBegin_1.plusYears(1);
    CustomerInfo customerInfo_2 = new CustomerInfo(medCertExpiry_2, feeBegin_2,
        feeExpiry_2);

    Validator secondValidator = new FeeValidator(customerInfo_2)
        .setNextValidator(new MedCertificateValidator(customerInfo_2));

    assertDoesNotThrow(secondValidator::validate);

    //with med. cert. and registration fee
    LocalDate medCertExpiry_3 = LocalDate.now().plusYears(1);
    LocalDate feeBegin_3 = LocalDate.now();
    LocalDate feeExpiry_3 = LocalDate.now().plusYears(1);
    CustomerInfo customerInfo_3 = new CustomerInfo(medCertExpiry_3, feeBegin_3,
        feeExpiry_3);

    Validator thirdValidator = new FeeValidator(customerInfo_3)
        .setNextValidator(new MedCertificateValidator(customerInfo_3));

    assertDoesNotThrow(thirdValidator::validate);
}
```

Qua, ad esempio, si è cercato di verificare tutte le condizioni in cui un utente rispettasse i vincoli di scadenza relativi a tassa di iscrizione e certificato medico.

4.3 Test dell'ORM

I test presenti in questo package sono tutti abbastanza simili poiché testano le varie operazioni CRUD degli oggetti DAO. Troviamo dunque metodi che testano inserimento, cancellazione, aggiornamento, lettura e creazione di oggetti dal database per ogni metodo dei DAO. Il corretto funzionamento di questi test è fondamentale per tutti i servizi della Business logic e in generale per il buon funzionamento dell'intero software.

4.4 Test d'integrazione

Come detto nell'introduzione di questo capitolo, tale test d'integrazione finale è stato utile per verificare, in assenza di interfaccia grafica, la corretta integrazione di tutte le com-

ponenti del software. Inoltre, è stata testata anche l'iniezione di tutte le dipendenze e la gestione della sessione da parte dell'ApplicationManager, oltre che il funzionamento dei getter dei controller per assicurarsi che lanciassero le giuste eccezioni quando richiesto. In questo caso, i test risultano essere dipendenti l'uno dall'altro, dunque è necessario eseguirli nel corretto ordine. Per garantire tale requisito si è utilizzato il decoratore `@TestMethodOrder(MethodOrderer.OrderAnnotation.class)` sopra la definizione della classe `IntegrationTest` e la specifica dell'ordine `@Order(i)` sopra la definizione dell'i-esimo test da eseguire in sequenza. Di seguito una tabella che mostra i test effettuati con una breve descrizione dello stesso.

Test	Descrizione
test1	test login e logout dell'admin
test 2	cambio informazioni personali admin
test 3	operazioni CRUD su personale palestra (receptionist e trainers)
test 4	cambio di informazioni personali da parte degli utenti creati dall'admin
test 5	operazioni CRUD dei corsi da parte dell'admin
test 6	operazioni CRUD degli abbonamenti da parte dell'admin
test 7	operazioni CRUD degli sconti da parte dell'admin
test 8	operazioni CRUD dei bundle da parte dell'admin
test 9	inizializzazione del profilo di un utente
test 10	completamento del profilo utente da parte di un receptionist
test 11	attivazione di abbonamenti da parte del receptionist ad un cliente
test 12	un trainer aggiunge una disponibilità
test 13	il cliente prende appuntamento per la disponibilità del trainer e prenota una classe
test 14	Il trainer può vedere l'appuntamento

Tabella 12: Test di integrazione effettuati.