UNIVERSITÀ
DEGLI STUDI
FIRENZE

Corso di laurea triennale in ingegneria informatica

Ingegneria del software

PIATTAFORMA DI GESTIONE DI LEZIONI INDIVIDUALI PER STUDENTI E DOCENTI

Autori: Giunti Alberto, Necerini Ivan
Docente: Vicario Enrico

Novembre 2023

Indice

1 Introduzione

1.1 Obbiettivo e descrizione del progetto

Il nostro progetto è incentrato sulla creazione di una piattaforma avanzata per la gestione di lezioni private, progettata per soddisfare le esigenze tanto degli studenti quanto dei tutor (studenti universitari o docenti). Questa piattaforma offre un ambiente virtuale dinamico e interattivo in cui studenti e tutor possono connettersi, collaborare e organizzare lezioni personalizzate in vari campi di studio. Indipendentemente dal livello di istruzione o dal contesto disciplinare, la nostra piattaforma offre un'opportunità per gli studenti di apprendere in modo personalizzato e per i tutor di condividere le proprie conoscenze in modo flessibile e remunerativo.
Gli attori principali nella nostra piattaforma sono gli **studenti**, alla ricerca di supporto educativo su misura, e i **tutor**, che offrono le proprie competenze e competenze in diverse materie e discipline. La piattaforma facilita l'incontro tra domanda e offerta di **lezioni private**, semplificando il processo di prenotazione e pagamento attraverso un sistema di transazioni online sicure.

- I tutor possono creare degli slot per le lezioni (con la possibilità di modificarli o cancellarli in un secondo momento), specificando dettagli cruciali quali materia, orario, modalità (online o in presenza) e tariffa. Inoltre, hanno la facoltà di visualizzare gli annunci pubblicati e consultare il calendario delle lezioni prenotate dagli studenti.
- Gli studenti possono cercare le lezioni disponibili, prenotare lezioni secondo le proprie esigenze e gestire le proprie prenotazioni in modo comodo e intuitivo.

SOURCE CODE ON GITHUB

REPORT CODE ON GITHUB

1.2 Architettura e pratiche utilizzate

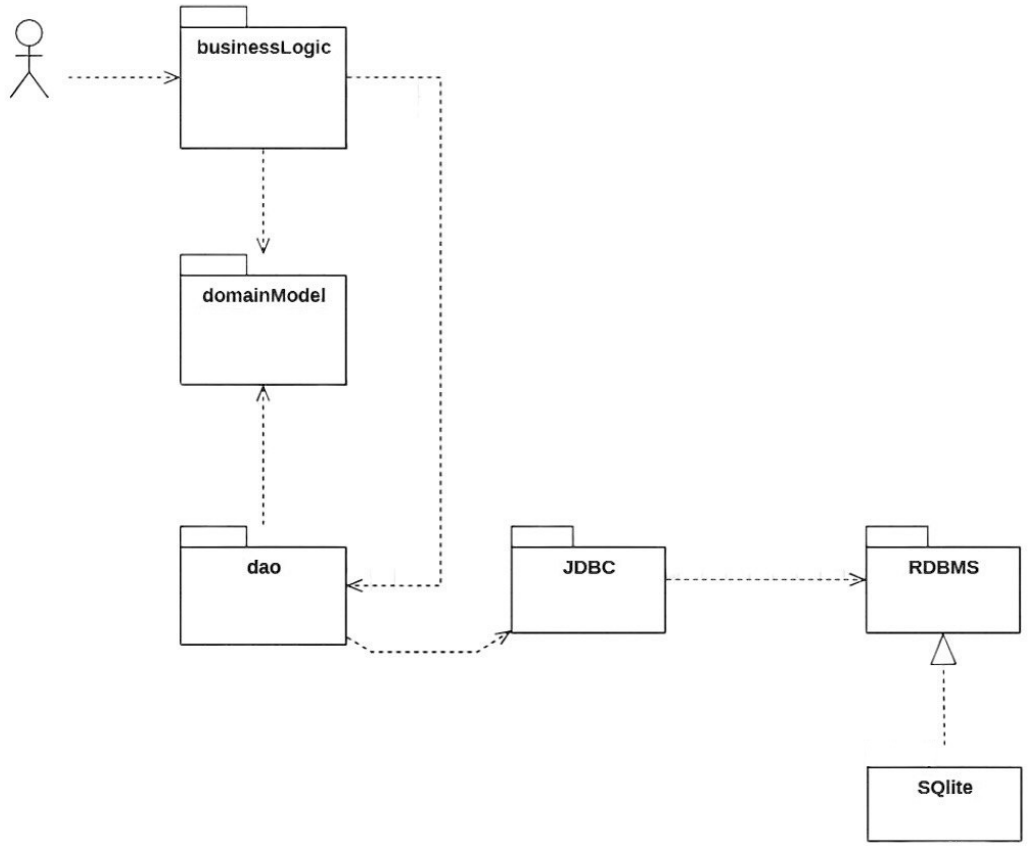

Figura 3: Diagramma delle dipendenze

Nella struttura del nostro progetto, mostrata nella Figura 3, abbiamo adottato un'architettura basata su Java per realizzare il software. Il modello dei dati è stato modellato attraverso il package *domainModel*, mentre la logica del sistema è stata implementata nel package *businessLogic*. Per garantire la persistenza dei dati, abbiamo utilizzato i *Data Access Objects (DAO)*, sfruttando la connessione al database *SQLite* tramite *JDBC (Java DataBase Connectivity)*.

I diagrammi delle classi e dei casi d'uso, conformi allo standard *UML (Unified Modeling Lan-*

guage), sono stati creati mediante l'uso del software "StarUML", offrendo una rappresentazione visiva chiara e intuitiva della struttura del nostro sistema. Per assicurarci della correttezza e della robustezza del nostro software, abbiamo condotto attività di testing utilizzando il framework *JUnit*. Questo ci ha consentito di eseguire test automatizzati per verificare il comportamento delle diverse componenti del sistema in modo ripetibile e affidabile.

2 Progettazione

2.1 Diagramma dei casi d'uso

Figura 4: Diagramma dei casi d'uso

All'interno del sistema sono definiti due attori: Studenti e Tutor. Ciascun ruolo ha responsabilità specifiche e modalità uniche di interazione con il sistema.

Nel diagramma dei casi d'uso (Figura 4), sono state rappresentate le interazioni tra gli attori e il sistema. È importante notare che, sebbene i casi d'uso relativi a recensioni e pagamenti non siano stati implementati nella versione attuale del gestionale, sono stati progettati come possibili estensioni future del sistema.

Gli **studenti** hanno la possibilità di effettuare diverse azioni nel sistema. Possono eseguire ricerche per trovare lezioni private, prenotarle, annullare prenotazioni e visualizzare un elenco delle lezioni a cui parteciperanno. D'altra parte, i **tutor** hanno il potere di creare nuove lezioni, cancellarle o modificarle in un secondo momento. Inoltre, hanno accesso allo storico completo delle lezioni, indipendentemente dallo stato in cui si trovano (disponibili, prenotate, svolte o cancellate).

Questa progettazione dettagliata delle interazioni tra gli attori e il sistema garantisce un'esperienza utente ottimale e una gestione efficiente delle lezioni private all'interno del nostro gestionale.

2.1.1 Templates dei casi d'uso

Di seguito, sono presentati i modelli dei casi d'uso effettivamente implementati. Ogni caso d'uso è descritto in dettaglio, specificando gli attori coinvolti e il flusso principale delle operazioni. In alcuni casi d'uso, sono anche documentati scenari alternativi, le condizioni iniziali e gli effetti sul sistema dopo il completamento dell'azione. Questa documentazione fornisce una panoramica esaustiva del comportamento del sistema e delle interazioni di tutor e studenti con esso.

Use case 1	**Cerca lezione**
Description	Lo studente cerca una lezione di suo interesse.
Level	User goal
Actors	Studente
Basic course	1. Lo studente inserisce i dati della ricerca 2. Vengono visualizzati i risultati della rircerca
Alternative course	2. La ricerca non produce nessun risultato

Figura 5: Cerca lezione

Use case 2	**Prenota lezione**
Description	Lo studente visualizza la lezione e si prenota.
Level	User goal
Actors	Studente
Basic course	1. Lo studente seleziona una lezione disponibile 2. Lo studente prenota 3. Lo studente paga la lezione
Alternative course	1. Lo studente non può prenotarsi perché ha un'altra lezione in quell'orario
Post-conditions	Il tutor viene notificato dell'avvenuta prenotazione. La lezione appena prenotata appare nella sezione "Le mie prenotazioni" dello studente. La lezione appena prenotata appare nel calendario del tutor. La lezione non appare più nelle ricerche della piattaforma e il suo stato passa a "Prenotato"

Figura 6: Prenota lezione

Use case 3	**Cancella prenotazione**
Description	Lo studente cancella la prenotazione a una lezione.
Level	User goal
Actors	Studente
Basic course	1. Lo studente va nella sezione "Le mie prenotazioni" 2. Lo studente seleziona la prenotazione desiderata 3. Lo studente cancella la prenotazione 4. Lo studente viene reindirizzato alla sezione "Le mie prenotazioni"
Alternative course	3. Lo studente non può cancellare la lezione perché troppo vicino alla data di essa
Pre-conditions	Deve esistere almeno una lezione alla quale lo studente è prenotato.
Post-conditions	Il tutor viene notifcato dell'avvenuta cancellazione. Lo studente viene rimborsato. La prenotazione scompare dalla sezione "Le mie prenotazioni" dello studente. La lezione scompare dal calendario del tutor e il suo stato torna a "Disponibile". Il tutor ha la possibilià di inserire nuovi annunci in quell'orario.

Figura 7: Cancella lezione

Use case 4	Visualizza le mie lezioni
Description	Lo studente visualizza le lezioni a cui è prenotato.
Level	User goal
Actors	Studente
Basic course	1. Lo studente va nella sezione "Le mie prenotazioni" 2. Viene mostrato l'elenco di prenotazioni
Alternative course	2. Non è presente nessuna prenotazione

Figura 8: Visualizza le mie lezioni (dello studente)

Use case 5	Crea lezione
Description	Il tutor crea una nuova lezione.
Level	User goal
Actors	Tutor
Basic course	1. Il tutor inserisce informazioni riguardanti la lezione da creare 2. La lezione viene creata
Alternative course	2. La lezione non viene creata perché ne esiste un'altra con gli stessi dati
Post-conditions	La lezione deve essere correttamente mostrata come risultato di una ricerca. La lezione ha stato "Disponibile".

Figura 9: Crea lezione

Use case 6	Cancella lezione
Description	Il tutor cancella una lezione.
Level	User goal
Actors	Tutor
Basic course	1. Il tutor va nella sezione "Il mio calendario" 2. Il tutor seleziona una lezione 3. Il tutor cancella la lezione selezionata
Alternative course	3. Il tutor non può cancellarla perché troppo vicino alla data di essa
Pre-conditions	Il tutor deve avere almeno una lezione nel calendario
Post-conditions	Se la lezione è nello stato "Prenotata", allora lo studente viene notificato dell'avvenuta cancellazione e rimbrsato. Lo stato della lezione permuta in "Cancellata". La lezione non compare più nella sezione "Le mie prenotazioni" dello studente.

Figura 10: Cancella lezione

Use case 7	Modifica lezione
Description	Il tutor modifica le informazioni relative a una lezione.
Level	User goal
Actors	Tutor
Basic course	1. Il tutor va nella sezione "Il mio calendario" 2. Il tutor seleziona una lezione 3. Il tutor applica le modifiche volute
Alternative course	3. Il tutor non può modificarla perché troppo vicino alla data di essa
Pre-conditions	Il tutor deve avere almeno una lezione nel calendario
Post-conditions	Se la lezione è nello stato "Prenotata", allora lo studente viene notificato dell'avvenuta cancellazione e la lezione appare modificata nella sezione "Le mie lezioni" dello studente. La lezione appare modificata nel calendario del tutor. La lezione appare con i dati modificati nelle ricerche degli utenti.

Figura 11: Modifica lezione

Use case 8	Visualizza le mie lezioni
Description	Il tutor visualizza lo storico delle sue lezioni.
Level	User goal
Actors	Tutor
Basic course	1. Il tutor va nella sezione "Il mio calendario" 2. Viene mostrato l'elenco di lezioni
Alternative course	2. Non è presente nessuna lezione nello storico

Figura 12: Visualizza le mie lezioni (storico del tutor)

2.2 Diagramma delle classi

Di seguito, in figura 13, è riportato il *diagramma delle classi*, il quale offre una panoramica delle classi implementate nel sistema e delle loro interazioni. Questo diagramma è anche arricchito dalla visualizzazione dei design pattern applicati, i quali verranno dettagliatamente spiegati nella sezione 2.3.

Le classi implementate sono organizzate in diversi package, ognuno dei quali ha uno specifico obiettivo:

- `businessLogic`: Questo package ospita le classi che gestiscono la business logic del sistema. Essa fa riferimento alle regole, ai processi e alle funzionalità che guidano il comportamento dell'applicazione in risposta alle diverse esigenze degli utenti. All'interno di questo package, sono presenti classi come *LessonController* per la gestione delle lezioni, *StudentController* per gli studenti e *TutorsController* per i tutor.
  - E' presente anche una classe astratta *PeopleController* che viene implementata nei controllori degli studenti e dei tutor citati precedentemente.
- `domainModel`: All'interno del package domainModel sono contenute le classi che definiscono il modello dei dati. Queste classi costituiscono la rappresentazione strutturata dei dati e degli oggetti all'interno dell'applicazione. Esse delineano in che modo i dati sono stati organizzati e rappresentati all'interno del sistema. Dentro questo package sono presenti 3 sotto-package:
  - `search`: Questo package è dedicato all'ottimizzazione del processo di ricerca delle lezioni. Esso fa uso del design pattern Decorator (per ulteriori dettagli si rimanda a

2.3.1), consentendo l'applicazione di una serie di filtri. Questa approfondita strategia di ricerca rende il processo di individuazione delle lezioni estremamente versatile.

  - `state`: Questo package è dedicato alla gestione dinamica del comportamento di una lezione in relazione al suo stato interno. Fa uso del design pattern *State*, un pattern comportamentale che consente a un oggetto di variare il suo comportamento quando il suo stato cambia. Nell'ambito del suddetto sistema, una lezione può esistere in quattro stati distinti: *available*, *booked*, *cancelled* e *completed*. Per dettagli specifici sul funzionamento di questi stati, si rimanda a 2.3.2.
  - `tags`: Questo package consente l'associazione di tag specifici alle lezioni, fornendo dettagli quali il livello, il soggetto, la zona geografica e il formato della lezione (online o in presenza). Tali informazioni extra permettono una categorizzazione avanzata delle lezioni, agevolando la ricerca e la selezione degli studenti. In sostanza, il package *domainModel*. Tags si sposa armoniosamente con il pattern Decorator, arricchendo le lezioni con informazioni specifiche (vedi 2.3.4).

- `DAO`: Questo package ospita le classi relative al *DAO (Data Access Object)*, che costituiscono il livello incaricato della gestione della persistenza dei dati. In altre parole, il DAO agisce come un intermediario tra il sistema e il database, gestendo l'accesso e la manipolazione dei dati in modo efficiente e sicuro. Questo approccio consente una separazione chiara tra la logica dell'applicazione e l'interazione con il database, contribuendo a una struttura modulare e manutenibile del sistema.

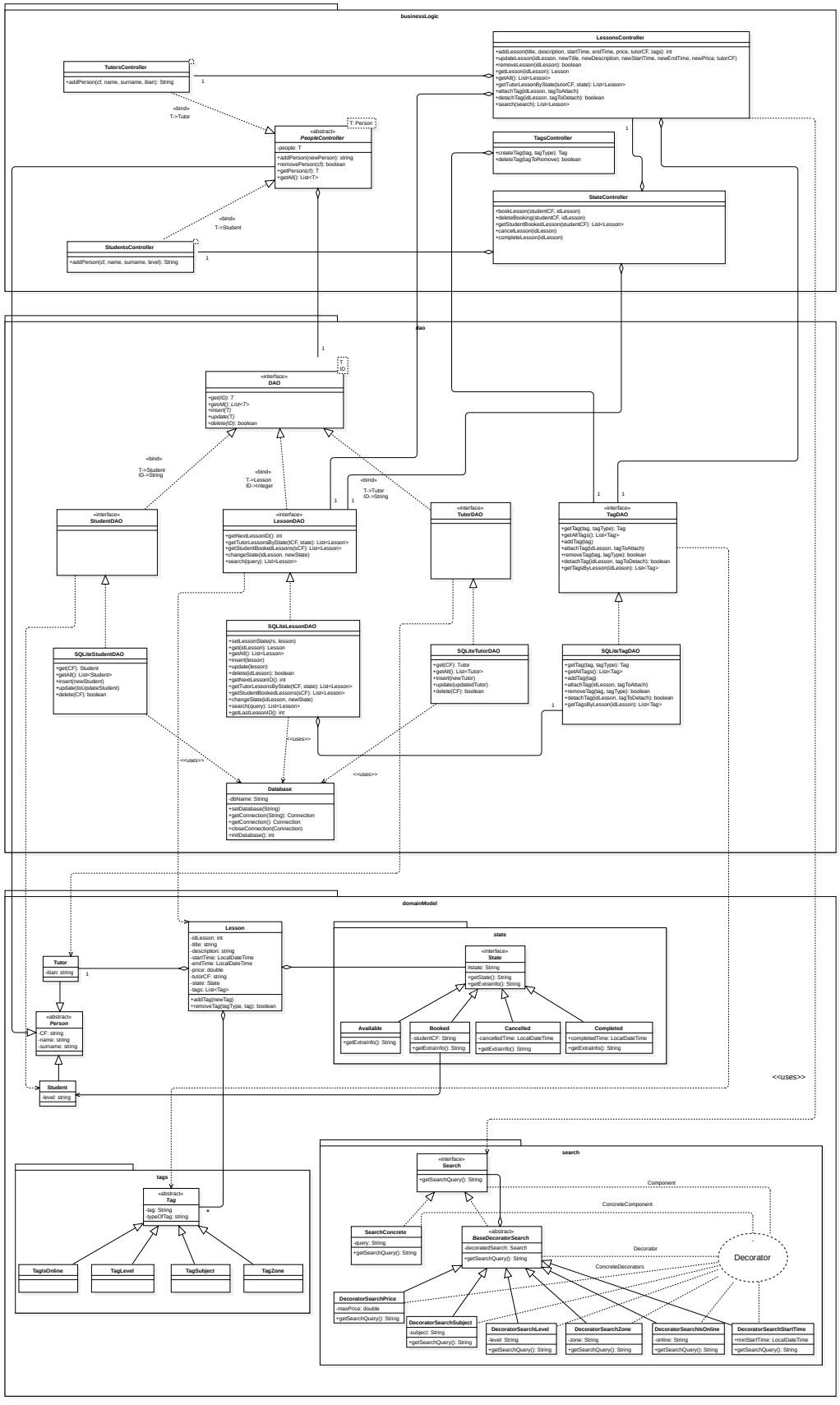

Figura 13: Diagramma delle classi

2.3 Aspetti rilevanti della progettazione

Di seguito vengono descritti in dettaglio i *design patterns* utilizzati per la realizzazione del progetto e altre logiche di programmazioni utili.

2.3.1 Decorator pattern

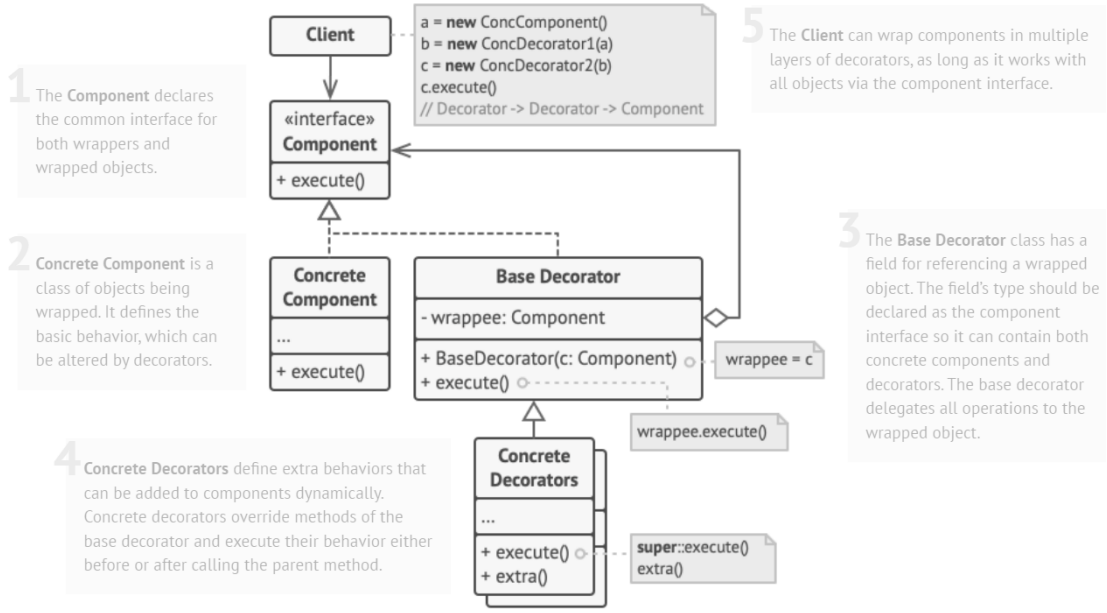

Figura 14: Pattern Decorator
(1)

Per ottimizzare e semplificare la ricerca delle lezioni per gli studenti, nonché per massimizzare l'efficienza del gestionale, è stata implementata una robusta struttura di ricerca attraverso l'impiego del design pattern strutturale *Decorator*. Questa implementazione consente una personalizzazione avanzata della ricerca delle lezioni tramite diversi filtri, tra cui *materia, livello di istruzione, modalità online o in presenza, zona geografica, data e prezzo*.

Questo pattern è descritto in modo preciso nella seguente descrizione presa da Wikipedia (2): "Il design pattern decorator consente di aggiungere nuove funzionalità ad oggetti già esistenti. Questo viene realizzato costruendo una nuova classe decoratore che "avvolge" l'oggetto originale. Al costruttore del decoratore si passa come parametro l'oggetto originale. È altresì possibile passarvi un differente decoratore. In questo modo, più decoratori possono essere concatenati l'uno all'altro, aggiungendo così in modo incrementale funzionalità alla classe concreta." In figura 14 si visualizza la struttura di questo pattern e di come i vari decoratori possanno essere combinati al fine di ottenere un'operazione di filtraggio personalizzata al massimo.

Ad esempio: Supponiamo di voler cercare lezioni di matematica (livello di scuola media) a Firenze. Questo obiettivo può essere raggiunto combinando la classe di ricerca di base, nota come SearchConcrete, con i decoratori DecoratorSearchSubject e DecoratorSearchZone. Questa concatenazione di decoratori consente agli studenti di effettuare ricerche altamente specifiche, affinando le opzioni in base alle loro esigenze uniche (vedi 13).

L'implementazione del pattern *Decorator* nel nostro gestionale di lezioni private offre flessibilità e potenza agli utenti, permettendo loro di trovare facilmente le opzioni di lezioni che meglio si adattano alle loro esigenze specifiche. Questa struttura di ricerca avanzata è stata progettata per garantire un'esperienza altamente efficiente nel nostro gestionale.

2.3.2 State pattern

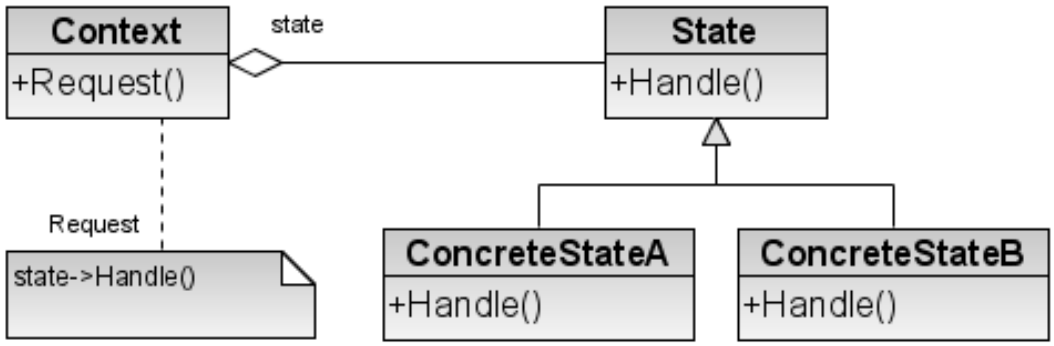

Figura 15: Pattern State
(3)

Il design pattern *State* è un pattern comportamentale che consente a un oggetto di modificare il suo comportamento quando il suo stato interno cambia. Esso consente ad un oggetto di cambiare il proprio comportamento a run-time in funzione dello stato in cui si trova (4).
In altre parole, permette di definire una famiglia di classi e di incapsulare ciascuna classe in un oggetto rappresentante lo stato. Questo oggetto di stato permette al contesto di variare il suo comportamento a seconda del suo stato corrente.

Come si vede in 13 il *context* è rappresentato dalla classe *Lesson* (in cui ci sarà un campo state di tipo "State"), l'interfaccia *State* è una classe astratta base e successivamente sono presenti quattro classi concrete che rappresentano i vari stati in cui si può trovare una lezione:

1. `Available`: Rappresenta lo stato in cui una lezione è disponibile per la prenotazione
2. `Booked`: Indica che una lezione è stata prenotata da uno studente.
3. `Cancelled`: Segnala che una lezione è stata cancellata.

4. `Completed`: Indica che una lezione è stata completata e include informazioni sulla data e l'ora di completamento.

L'uso del pattern State permette una gestione chiara e modulare degli stati delle lezioni, semplificando il codice e garantendo una facile estendibilità nel caso in cui nuovi stati debbano essere aggiunti in futuro. Grazie a questo approccio, il comportamento della lezione può essere adattato in modo dinamico, rispondendo ai cambiamenti di stato senza complicare eccessivamente la logica dell'applicazione.

2.3.3 DAO

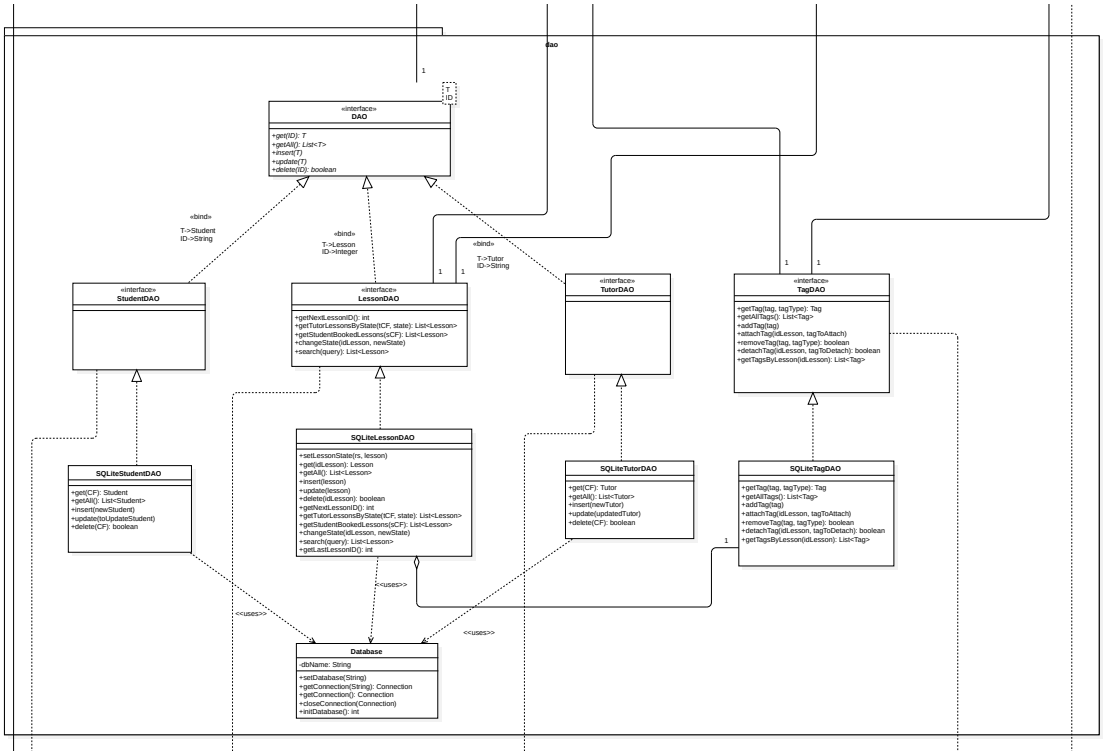

Figura 16: DAO

Il *Data Access Object (DAO)* rappresenta un pattern architetturale ampiamente diffuso nel contesto dello sviluppo software. Questo pattern si presenta come un'interfaccia astratta che si occupa della gestione della persistenza dei dati. Nel contesto del progetto, è stato utilizzato il DAO per interagire con un database SQLite, separando così la *business logic* dall'accesso diretto al database (*data layer*), in conformità al principio di singola responsabilità (*SRP*). I metodi del DAO, insieme alle rispettive query, vengono richiamati dalle classi della business logic (5).
Il DAO fornisce un'interfaccia standard alle classi del sistema per eseguire operazioni `CRUD` (Create, Read, Update, Delete) sui dati, mantenendo nascosti i dettagli implementativi della persistenza. Questo livello di isolamento favorisce una maggiore modularità e manutenibilità del codice, consentendo di apportare modifiche alla struttura del database senza impattare sul resto dell'applicazione.

Nel progetto, come illustrato nella figura 16, è stata definita un'interfaccia generica DAO contenente le operazioni CRUD che le classi DAO concrete devono implementare per gestire la persistenza. Ogni entità del sistema è associata a un DAO specifico, il quale è rappresentato da un'interfaccia (es. StudentDAO, LessonDAO, TutorDAO e TagDAO). Queste interfacce estendono l'interfaccia generica DAO e specificano i tipi di dati con cui le classi DAO concrete dovranno lavorare. Inoltre, è possibile aggiungere metodi aggiuntivi specifici per ciascuna entità.

Nel dettaglio, l'interfaccia TagDAO contiene operazioni specializzate relative alla gestione dei tag, come l'associazione dei tag con le lezioni e la gestione delle tipologie di tag. Queste operazioni non sono necessarie per tutte le entità nel sistema. Poiché l'interfaccia DAO fornisce operazioni standard come get, getAll, insert, update e delete, non tutte queste operazioni sono rilevanti per l'entità Tag. Pertanto, non è stata estesa direttamente l'interfaccia base DAO in TagDAO, mantenendo così le interfacce specifiche alle entità focalizzate e snelle.

Le classi concrete, come SQLiteStudentDAO, SQLiteTutorDAO, SQLiteTagDAO e SQLiteLessonDAO, implementano le interfacce specifiche. Ogni classe concreta è responsabile della gestione delle operazioni di accesso al database SQLite per l'entità corrispondente. Per esempio, SQLiteTagDAO si occupa dell'associazione tra i tag e le lezioni, mentre SQLiteLessonDAO gestisce gli stati delle lezioni in modo separato.

2.3.4 Tags

Il package *tags* svolge un ruolo cruciale nella categorizzazione e classificazione avanzata delle lezioni, contribuendo a una gestione più efficace delle informazioni associate alle lezioni stesse. Questo package consente l'associazione di tag specifici alle lezioni, fornendo dettagli come il livello, il soggetto, la zona geografica e il formato della lezione (online o in presenza). Queste informazioni extra arricchiscono il contesto delle lezioni, facilitando la ricerca e la selezione degli

studenti interessati. Ovviamente la classe *Lesson* nel package domainModel ha una lista di oggetti Tag come attributo.
In particolare, il package contiene la classe base astratta *Tags* (ogni tag ha un nome (tag) e un tipo (typeOfTag)), e diversi classi concrete che la estendono, ognuna delle quali rappresenta un tipo specifico di tag (*TagIsOnline, TagLevel, TagSubject, TagZone*).

Utilizzando il pattern Decorator insieme ai tag, è possibile separare in modo efficiente le responsabilità. I tag si concentrano sulla definizione degli attributi specifici, mentre il pattern Decorator gestisce l'estensione dinamica degli oggetti Lesson. Ciò rende il sistema più modulare, facilitando la manutenzione e l'estensione del codice nel tempo. Se in futuro nuovi tipi di tag o nuove funzionalità di decorazione sono necessari, è possibile aggiungerli senza dover modificare il codice esistente.

2.4 Entity-Relationship Diagram

In figura 17 è rappresentata la struttura del database relazionale implementato utilizzando il linguaggio SQL con il motore SQLite. Il DB è progettato per gestire diverse entità chiave nel contesto delle lezioni private: tutor (`Tutors`), studenti (`Students`), lezioni (`Lessons`), tag (`Tags`) e la tabella di associazione `lessonsTags`.
Ora verranno analizzate e spiegate nel dettaglio alcune scelte implementative, senza specificare quelle ovvie, ricavabili dal diagramma ER (17):

- **Tabella "Lessons"**: *Foreign key* = tutorCF. Questa scelta di collegare questo attributo alla *Primary key* della tabella `Tutors`, permette di verificare l'integrità referenziale; ogni lezione è associata a un tutor esistente, evitando l'orfanità dei dati e garantendo che ogni lezione sia gestita da un tutor valido.

- **Tabella "LessonsTags"**: *Foreign key 1* = tag, tagType, *Foreign key 2* = idLesson. Si tratta di una tabella di collegamento che stabilisce una relazione molti a molti tra le lezioni e i tags. Ha un ruolo fondamentale nella gestione del sistema di lezioni private. Ogni riga in questa tabella indica che una determinata lezione è associata a un certo tag. La tabella LessonsTags serve a consentire l'associazione flessibile e dinamica di uno o più tag a ciascuna lezione. In altre parole, permette di categorizzare ogni lezione in base a vari criteri come il soggetto, la modalità (online o presenziale), la zona geografica e così via. Alcuni aspetti degni di nota:

    1. *Flessibilità nell'associazione dei tag*: Le lezioni possono avere diversi tag, e questi possono variare ampiamente. Alcune lezioni potrebbero essere associate solo a un soggetto, mentre altre potrebbero avere multiple associazioni come soggetto, modalità e zona geografica. Questa flessibilità è fondamentale per gestire una vasta gamma di lezioni.

2. *Facilita la ricerca e la selezione*: Grazie ai tag, gli studenti possono facilmente cercare lezioni basate su criteri specifici. Ad esempio, uno studente potrebbe cercare tutte le lezioni online di matematica a Firenze. La tabella LessonsTags facilita questo processo consentendo di identificare rapidamente le lezioni che soddisfano determinati requisiti.
3. *Struttura normalizzata*: Utilizzando una tabella di collegamento, il database è normalizzato, il che significa che le informazioni sono organizzate in modo efficiente, riducendo la duplicazione dei dati e migliorando la coerenza e l'integrità dei dati.

- **Tabella "Students"**: da notare la presenza degli attributi *state (TEXT NOT NULL)* e *stateExtraInfo (TEXT)*. *Questo approccio elegante integra perfettamente lo stato della lezione con le informazioni sullo studente che ha effettuato la prenotazione. Non solo semplifica la gestione delle prenotazioni all'interno del database, ma consente anche di ottenere questa informazione senza la necessità di ulteriori query complesse, migliorando così l'efficienza e la velocità delle operazioni legate alle prenotazioni delle lezioni nel tuo sistema*. Per maggiori informazioni si rimanda alla sezione 3.1.5.

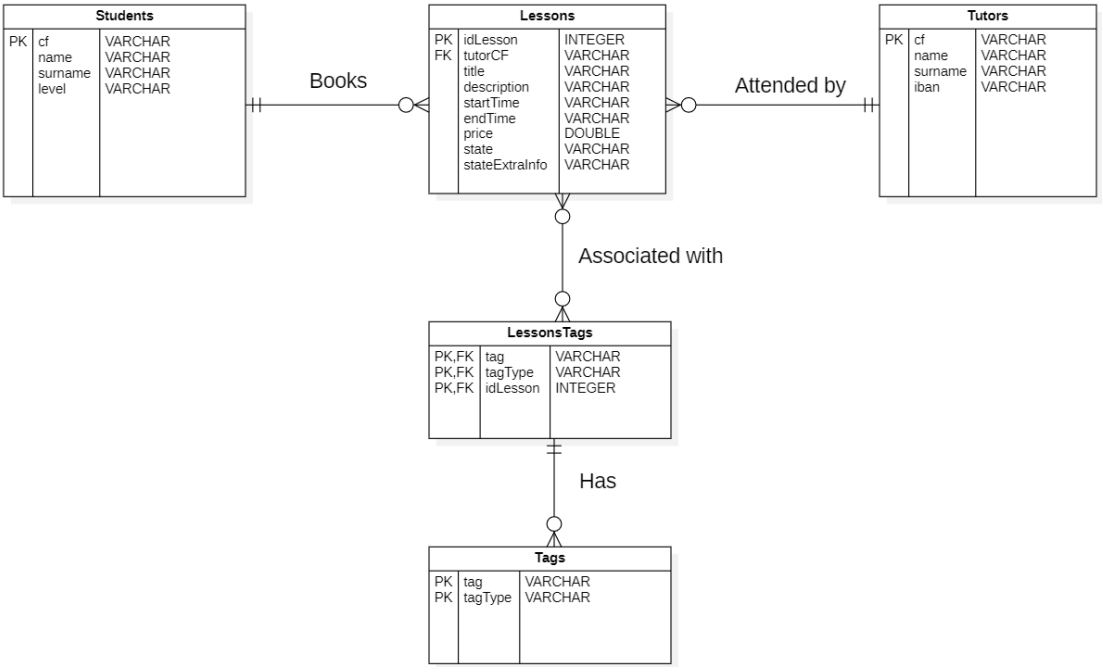

Figura 17: Diagramma Entity-Relationship del database

Snippet di codice che mostra la definizione di una delle tabelle presente nel diagramma:

Snippet 1: Esempio di definizione delle tabelle

```
-- Table: lessonsTags
CREATE TABLE IF NOT EXISTS lessonsTags
(
    tag         TEXT NOT NULL,
    tagType     TEXT NOT NULL,
    idLesson    INTEGER NOT NULL,
    PRIMARY KEY (tag, tagType, idLesson),
    FOREIGN KEY (tag, tagType) REFERENCES tags (tag, tagType
        ) ON
        UPDATE CASCADE ON DELETE CASCADE,
    FOREIGN KEY (idLesson) REFERENCES lessons (idLesson) ON
        UPDATE CASCADE ON DELETE CASCADE
);
```

3 Implementazione

Il progetto è stato realizzato in Java, seguendo le migliori pratiche del settore. Per la gestione delle librerie e l'organizzazione del progetto, è stato adottato Apache Maven, di conseguenza la struttura del progetto segue lo standard Maven (vedi (6)). Questa scelta ci ha permesso di mantenere un ambiente di sviluppo coerente, semplificando il processo di gestione delle dipendenze e di build del software.
Il codice è articolato come si vede nelle figure 18 e 19, e in questa sezione d'implementazione verranno descritti i contenuti di **src/main/java**.

SOURCE CODE ON GITHUB

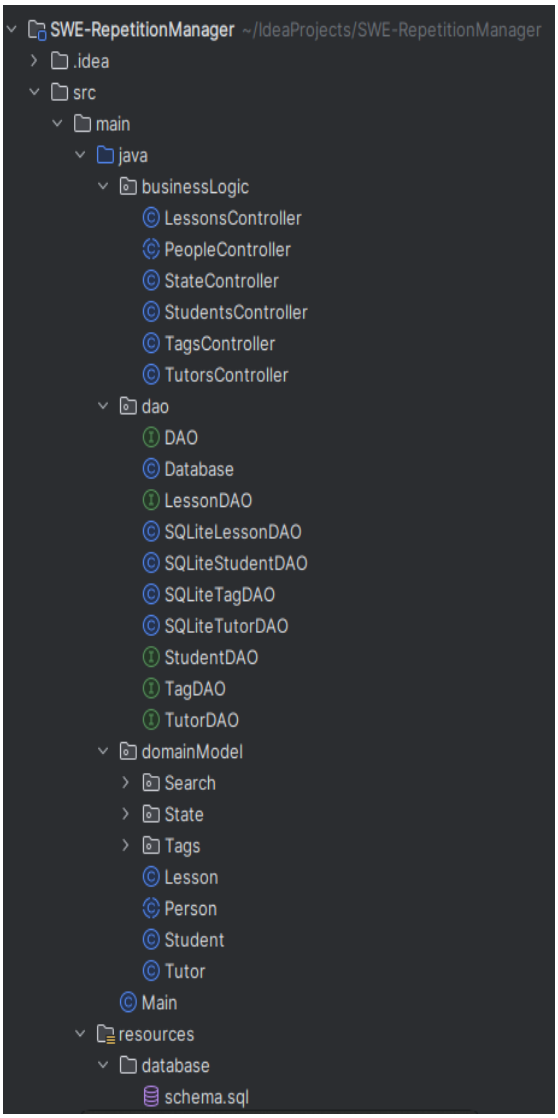

Figura 18: Suddivisione del codice

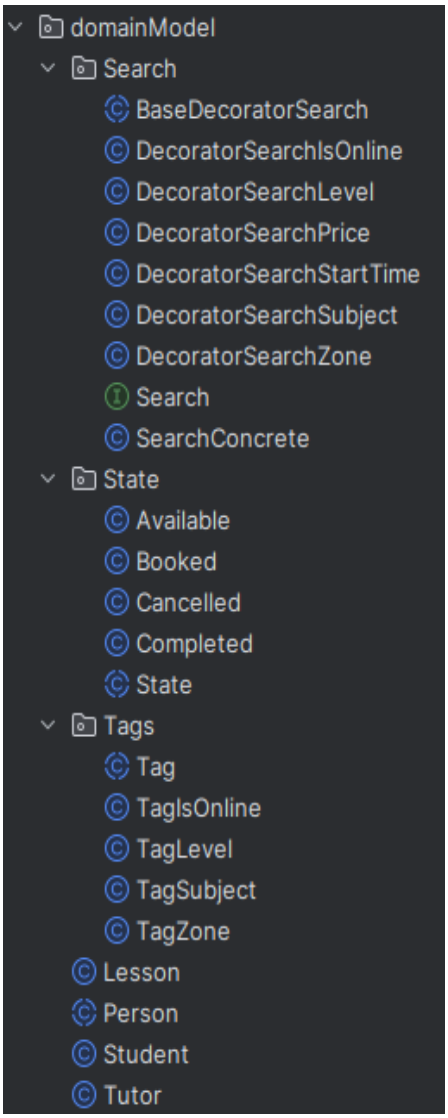

Figura 19: Zoom nel package domainModel

3.1 domainModel

Questo package è il cuore del dominio del progetto e modella le classi principali che caratterizzano il sistema.

3.1.1 Person, Student, Tutor

Come si nota dal diagramma delle classi (vedi 13), la classe astratta *Person* rappresenta una persona generica con attributi come codice fiscale, nome e cognome. Le classi *Student* e *Tutor* estendono questa classe per rappresentare rispettivamente studenti e tutor, aggiungendo l'attributo `level` per gli studenti e `iban` per i tutor.

3.1.2 Lesson

Questa classe rappresenta una lezione all'interno del sistema. Ha attributi come `idLesson`, `title`, `description`, `startTime`, `endTime`, `price`, e `tutorCF`. Inoltre, mantiene una lista di `tag` (vedi 2.3.4) che categorizzano la lezione. La classe è in grado di gestire lo stato della lezione attraverso le classi del package *State* (vedi 2.3.2). Ad esempio, può essere in uno stato "Available" o "Booked." La classe Lesson collega l'entità Lesson con i tag e gli stati del sistema.

3.1.3 State package

Questo package gestisce gli stati delle lezioni, consentendo la transizione tra gli stati *"Available", "Booked", "Cancelled" e "Completed"*. Ogni stato è rappresentato da una classe con un metodo "getExtraInfo()" che restituisce informazioni aggiuntive come il codice fiscale dello studente che ha prenotato, il momento in cui è stata cancellata o completata la lezione. Tutte e quattro le classi concrete estendono la classe base astratta *State*.

3.1.4 Tags package

Questo package gestisce i tag associati alle lezioni, come "Online", "Level", "Subject" e "Zone". Ogni tipo di tag è rappresentato da una classe separata (come *TagIsOnline, TagLevel, TagSubject e TagZone*) che estende la classe astratta *Tag*. Questi tag consentono una categorizzazione dettagliata delle lezioni in base a criteri come la modalità online/offline, il livello di difficoltà, il soggetto e la zona geografica.

3.1.5 Search package

Il package *domainModel.Search* nel progetto gestisce la funzionalità di ricerca delle lezioni con criteri specifici. Utilizza il pattern *decorator* (vedi sezione 2.3.1) per consentire la composizione flessibile delle query di ricerca. Ecco un'analisi delle classi all'interno del package:

- `Search Interface`: Questa interfaccia definisce il contratto per gli oggetti di ricerca. L'implementazione di base restituisce una query SQL di base che seleziona tutte le lezioni disponibili.

- `SearchConcrete Class`: Questa classe implementa l'interfaccia Search e fornisce una query di base per selezionare le lezioni con lo stato 'Available'.
- `BaseDecoratorSearch Class`: Questa classe astratta implementa l'interfaccia Search e fornisce un costruttore che accetta un oggetto Search decorato. Questa classe è il componente base per tutti i decoratori.
- `DecoratorSearch-FILTER`: Queste classi, come si vede da 13, sono tutti decoratori che estendono *BaseDecoratorSearch*. Queste classi aggiungono delle clausole alla query di ricerca base per filtrare le lezioni secondo il loro criterio per cui sono state progettate. La query verrà modificata per includere solo le lezioni interessate. I decoratori sono: *DecoratorSearchSubject, DecoratorSearchStartTime, DecoratorSearchPrice, DecoratorSearchLevel* e *DecoratorSearchIsOnline*.

In generale, questo sistema di classi consente di costruire query di ricerca complesse combinando diversi criteri. Ad esempio, se desideri trovare lezioni online (modalità di insegnamento), di un certo livello e in un certo orario, puoi combinare gli oggetti decorati per creare una query composta. Il pattern decorator ti permette di estendere in modo dinamico le funzionalità di ricerca senza dover modificare il codice esistente.

3.2 businessLogic

La *businessLogic* è il package che si occupa della gestione dei dati. Fornisce i metodi per creare, modificare e eliminare gli elementi del domainModel, in sostanza fornisce le API del gestionale.

3.2.1 PeopleController, StudentsController e TutorsController

La classe astratta *PeopleController* fornisce metodi generici per gestire le persone nel sistema, sia tutor che studenti. Le sottoclassi *StudentsController e TutorsController* specializzano l'implementazione per studenti e tutor, rispettivamente, estendendo questa classe e fornendo implementazioni specifiche per l'aggiunta, la cancellazione e il recupero dei dati relativi a studenti e tutor.

3.2.2 LessonsController

La classe *LessonsController* gestisce le operazioni legate alle lezioni nel sistema. Essa comunica con il database e gestisce l'aggiunta, la modifica, la cancellazione e il recupero delle lezioni. Oltre a ciò, permette l'assegnazione di tag alle lezioni e ne gestisce la rimozione.
Nello snippet poco sotto si vede come nel metodo `addLesson()` viene effettuato un controllo sull'esistenza del tutor associato alla lezione. Successivamente, nello snippet 3, viene verificato che il tutor non stia cercando di inserire una nuova lezione in un orario in cui è già impegnato. Questa procedura evita sovrapposizioni di lezioni e garantendo che il tutor sia disponibile per condurre la lezione richiesta:

Snippet 2: Metodo "addLessons()" di *LessonsController*

```
public int addLesson(...) throws Exception{
    Tutor tutor = tutorsController.getPerson(tutorCF);
    // @throws Exception If the tutor is not found
    if (tutor == null)
        throw new IllegalArgumentException("Tutor␣not␣found"
            );
    ...
}
```

Snippet 3: Metodo "addLessons()" di *LessonsController*

```
public int addLesson(...) throws Exception{
    ...
    //throws IllegalArgumentException If the tutor is
        already occupied in the given time range

    for (Lesson l : this.lessonDAO.getAll()) {
        if (l.getTutorCF().equals(tutorCF)) {
            if ((l.getStartTime().isBefore(endTime) || l.
                getStartTime().equals(endTime))
                    && (l.getEndTime().isAfter(startTime) ||
                        l.getEndTime().equals(startTime)))
                throw new RuntimeException("...");
        }
    }
    ...
}
```

3.2.3 StateController

La classe *StateController* gestisce gli stati delle lezioni, consentendo di prenotare una lezione, cancellare una prenotazione, segnare una lezione come completata o eliminarla . Interagisce con la classe *LessonsController* per ottenere e modificare le informazioni sullo stato delle lezioni. Questa classe gestisce anche la validità delle operazioni, garantendo che un'azione possa essere eseguita solo se la lezione è nello stato appropriato.
Per esempio, nel metodo `deleteBooking()` vegnono effettuati vari controlli, descritti nei commenti nello snippet di codice sottostante:

Snippet 4: Metodo "deleteBooking()" di *StateController*

```
public void deleteBooking(String studentCF, int idLesson)
    throws Exception {
    // Retrieve the lesson associated with the given ID.
    Lesson lesson = lessonsController.getLesson(idLesson);
    // If the lesson does not exist, throw an exception.
    if (lesson == null) {
        throw new IllegalArgumentException("The␣given␣lesson
            ␣ID␣does␣not␣exist.");
    }
```

```
        // Retrieve the student associated with the given Fiscal
            Code.
        Student student = studentsController.getPerson(studentCF
            );
        // If the student does not exist, throw an exception.
        if (student == null) {
            throw new IllegalArgumentException("The␣given␣
                student␣does␣not␣exist.");
        }

        // Check if the lesson is in the 'Booked' state.
        if (!Objects.equals(lesson.getState(), "Booked")) {
            // If the lesson is not in the 'Booked' state,
                cancelation is not allowed.
            throw new RuntimeException("The␣booking␣cannot␣be␣
                canceled␣because␣it␣is␣not␣in␣a␣'Booked'␣state."
                );
        }

        // Check if the student associated with the booking
            attempt is the actual booker.
        if (Objects.equals(lesson.getStateExtraInfo(), studentCF
            )) {
            // If the student is the actual booker, change the
                lesson state to 'Available'.
            Available av = new Available();
            this.lessonDAO.changeState(idLesson, av);
        } else {
            // If the student is not the actual booker, throw an
                 exception.
            throw new RuntimeException("Student␣" + studentCF +
                "␣is␣not␣booked␣for␣lesson␣#" + idLesson + ".");
        }
    }
```

3.2.4 TagsController

La classe *TagsController* gestisce i tag associati alle lezioni. Permette la creazione di nuovi tag e la loro associazione a specifiche lezioni. La classe fornisce anche la possibilità di rimuovere i tag, garantendo la coerenza dei dati nel sistema.

3.3 dao

Questo package implementa il pattern DAO, descritto dettagliatamente nella sezione 2.3.3 del documento. Le interfacce definite all'interno hanno l'unico scopo di dichiarare i metodi che le classi concrete dei DAO dovranno implementare. Le classi concrete, a loro volta, saranno utilizzate dai Controllers, come descritto nella sezione 3.2 del testo.

Per la gestione del database, è stata adottata la libreria SQLite. Sono state sviluppate dunque classi DAO concrete che stabiliscono e gestiscono la connessione con il database SQLite. In particolare, la classe *Database* (consultare la sezione 3.4) si occupa di gestire la connessione. Le altre classi nel package sono responsabili di eseguire le operazioni CRUD sul database e di ricostruire gli oggetti Java quando richiesto.

Nel seguente snippet viene mostrato un DAO concreto che gestisce le lezioni, in particolare viene mostrato il metodo con il quale si ottiene una specifica lezione:

Snippet 5: Metodo "get()" di *SQLiteLessonDAO*

```
public class SQLiteLessonDAO implements LessonDAO{
    private final TagDAO tagDAO;

    public SQLiteLessonDAO(TagDAO tagDAO){
        this.tagDAO = tagDAO;
    }
...

   // Get a specific lesson
    @Override
    public Lesson get(Integer idLesson) throws Exception {
        Connection con = Database.getConnection();
        Lesson lesson = null;
        PreparedStatement ps = con.prepareStatement("SELECT␣
            *␣FROM␣lessons␣WHERE␣idLesson␣=␣?");
        ps.setInt(1, idLesson);
        ResultSet rs = ps.executeQuery();

        if(rs.next()){
            lesson = new Lesson( // This recreates the
                attributes of the lesson
                    rs.getInt("idLesson"),
                    ...
                    LocalDateTime.parse(rs.getString("
```

```
                        startTime")),
                    ...
            );

            // This recreates the state of the lesson
            this.setLessonState(rs, lesson);
            // This recreates the tags of the lesson
            List<Tag> lessonTags = this.tagDAO.
                getTagsByLesson(lesson.getIdLesson());
            for (Tag t: lessonTags){
                lesson.addTag(t);
            }
        }
        rs.close();
        ps.close();
        Database.closeConnection(con);
        return lesson;
    }
...
}
```

3.4 Connessione al DB

La connessione al database è stata realizzata con JDBC ed è utilizzata dalle classi DAO concrete (vedi sezione 3.3):

Snippet 6: Connessione al DB grazie a JDBC

```
public class Database {
    private static String dbName = "main.db";
    ...
    public static Connection getConnection(String dbName)
        throws SQLException {
        return DriverManager.getConnection("jdbc:sqlite:" +
            dbName);
    }
    ...
}
```

4 Test

L'obiettivo principale dei test è garantire il corretto funzionamento delle funzionalità implementate. Questo processo di verifica avviene sia a livello di singole unità (test di unità) che durante l'uso dell'applicativo (test funzionali).

I test sono organizzati nella directory ***src/test/java***, seguendo la struttura del progetto principale (e la convenzione dei progetti Maven). Questa organizzazione è cruciale per mantenere la coerenza e facilitare la manutenzione del codice di test. I nomi dei test rispettano le convenzioni descritte in (7).

Per l'implementazione dei test, è stato utilizzato JUnit. Abbiamo sviluppato sia test di unità che test funzionali. I test di unità mirano a verificare il comportamento delle singole unità del codice, garantendo che ciascuna parte funzioni come previsto in isolamento. D'altra parte, i test funzionali sono stati progettati per verificare le funzionalità del sistema secondo i casi d'uso specificati. Questi test funzionali coincidono con le funzionalità del package businessLogic e sono stati dettagliatamente spiegati nella sezione 4.2.

4.1 domainModel

In questa cartella troviamo gli unit test per `Lesson` e `State`.

- **LessonTest.java**:

  1. *testLessonCreationWithValidDates()*: Questo test verifica che la creazione di una lezione con date valide non generi eccezioni. Assicura che il costruttore di Lesson gestisca correttamente le date di inizio e fine.
  2. *testLessonCreationWithInvalidDates()*: Questo test verifica che la creazione di una lezione con una data di fine precedente a quella di inizio generi un'eccezione del tipo IllegalArgumentException.
  3. *testGetters()*: Questo test verifica i metodi getter di Lesson per assicurarsi che restituiscano i valori corretti per i diversi attributi della lezione.
  4. *testSetState()*: Questo test verifica il metodo setState() di Lesson, che imposta lo stato della lezione. Assicura che lo stato e le informazioni extra siano impostati correttamente.
  5. *testTagOperations()*: Questo test verifica i metodi di gestione dei tag di Lesson. Verifica che i tag possano essere aggiunti correttamente alla lezione, rimossi correttamente e che l'operazione di rimozione generi un'eccezione se il tipo di tag non è valido. Verifica anche che il tentativo di rimuovere un tag che non esiste restituisca false.

- **StateTest.java**:

  1. *testAvailableState()*: Questo test verifica che lo stato "Available" sia creato correttamente e che l'informazione extra sia nulla.
  2. *testCompletedState()*: Questo test verifica che lo stato "Completed" sia creato correttamente con un timestamp specifico come informazione extra.
  3. *testCancelledState()*: Questo test verifica che lo stato "Cancelled" sia creato correttamente con un timestamp specifico come informazione extra.
  4. *testBookedState()*: Questo test verifica che lo stato "Booked" sia creato correttamente con il codice fiscale dello studente come informazione extra.

Snippet 7: Esempio di unit test di Lesson

```
@Test
void testTagOperations() {
    // Verifica i metodi di gestione dei tag
    TagZone tagZone = new TagZone("Firenze");
    TagSubject tagSubject = new TagSubject( "Matematica"
        );

    // Aggiungi un tag
    lesson.addTag(tagZone);
    lesson.addTag(tagSubject);
    Assertions.assertTrue(lesson.getTags().contains(
        tagZone));

    // Rimuovi un tag
    lesson.removeTag("Zone", "Firenze");
    Assertions.assertFalse(lesson.getTags().contains(
        tagZone));

    // Verifica che il metodo removeTag lanci un'
        eccezione quando si cerca di rimuovere un tag
        con un tagType non valido
    Assertions.assertThrows(IllegalArgumentException.
        class, () -> lesson.removeTag("InvalidType", "
        Tag1"));

    // Tentativo di rimuovere un tag che non esiste
    boolean removed = lesson.removeTag("Zone", "Tag2");
    Assertions.assertFalse(removed);
}
```

4.2 businessLogic

I test inclusi nel package domainModel svolgono una duplice funzione: da un lato, verificano il corretto funzionamento delle operazioni principali specificate nei diagrammi dei casi d'uso e nei template dei casi d'uso (come descritto nella sezione 2.1 del documento); dall'altro, costituiscono i test funzionali del sistema. Questi test controllano attentamente una serie di operazioni significative, mirando a coprire gli scenari previsti nell'ambito delle funzionalità dell'applicazione.

Ora verranno descritti solo alcuni dei test effettuati, solo quelli fondamentali:

- **LessonsControllerTest.java**:
  1. *testGetLessonByID()*: Questo test verifica se è possibile ottenere una lezione dal database utilizzando l'ID della lezione. Verifica che i dettagli della lezione recuperata corrispondano ai valori inseriti durante la configurazione del database di test.
  2. *testInsertLesson()*: Questo test verifica se è possibile inserire una nuova lezione nel database. Verifica anche se l'ID della lezione assegnato è valido e se la lezione è stata inserita correttamente nel database.
  3. *testUpdateLesson()*: Questo test verifica se è possibile aggiornare i dettagli di una lezione nel database e se le modifiche sono state applicate correttamente.
  4. *testDeleteLesson()*: Questo test verifica se è possibile eliminare una lezione dal database utilizzando l'ID della lezione. Verifica anche se la lezione è stata effettivamente eliminata dal database.
  5. *testAttachTagToLesson()*: Questo test verifica se è possibile allegare un nuovo tag a una lezione esistente nel database. Il test verifica se il tag è stato allegato correttamente alla lezione.
  6. *testDetachTagFromLesson()*: Questo test verifica se è possibile staccare un tag da una lezione esistente nel database. Il test verifica se il tag è stato rimosso correttamente dalla lezione.
  7. *ecc...*
- **TutorsControllerTest.java**:
  1. *when_AddingNewTutor_Expect_Success()*: Questo test verifica se è possibile aggiungere un nuovo tutor al database. Verifica se il tutor è stato aggiunto correttamente e se è possibile recuperarlo dal database.
  2. *when_AddingAlreadyExistingTutor_Expect_Exception()*: Questo test verifica se il sistema gestisce correttamente il caso in cui si cerca di aggiungere un tutor già esistente nel database.

3. *when_RemovingExistingTutor_Expect_True()*: Questo test verifica se è possibile rimuovere un tutor esistente dal database. Verifica se la rimozione è avvenuta con successo.
4. *when_RemovingNonExistingTutor_Expect_False()*: Questo test verifica se il sistema restituisce false quando si cerca di rimuovere un tutor non esistente dal database.
5. *ecc...*

Snippet 8: Esempio di unit test in LessonsControllerTest

```
@Test
void testInsertLesson() throws Exception {
    List<Tag> tags = new ArrayList<>();
    TagSubject ts = new TagSubject("English");
    tags.add(ts);
    int lessonId = lessonsController.addLesson("English␣
        Lesson", "English␣basics", LocalDateTime.now(),
        LocalDateTime.now(), 40.0, "tutor2", tags);

    // Assicurati che lessonId sia un valore valido
    assertTrue(lessonId > 0);

    // Assicurati che la lezione sia stata inserita
        correttamente nel database
    Lesson insertedLesson = lessonsController.getLesson(
        lessonId);
    assertNotNull(insertedLesson);
}
```

4.3 dao

Questo package contiene test specifici per le implementazioni concrete delle classi DAO. I test sono progettati per assicurare la persistenza corretta delle modifiche nel database e per rilevare errori derivanti da operazioni non valide. L'obiettivo principale è garantire l'integrità e l'affidabilità delle operazioni delle classi DAO nel sistema.

Il seguente snippet mostra il test *testDeleteLesson()* del file **SQLiteLessonDAOTest.java**. In sintesi, il test verifica che il metodo `delete()` di *SQLiteLessonDAO* elimini correttamente una lezione dal database e che l'elenco delle lezioni rimanenti abbia la dimensione prevista dopo l'eliminazione. Questo aiuta a garantire che l'operazione di eliminazione sia stata eseguita correttamente senza causare errori nel sistema.

Snippet 9: Esempio di test del SQLiteLessonDAOTest.java

```
@Test
void testDeleteLesson() throws Exception {
    // Test the delete method to delete a lesson
    boolean result = lessonDAO.delete(1);
    Assertions.assertTrue(result);
    List<Lesson> lessons = lessonDAO.getAll();
    Assertions.assertEquals(1, lessons.size());
}
```

4.4 Risultati dei test

Tutti i test implementati sono stati eseguiti senza errori. Durante l'esecuzione dei test, nessuna eccezione è stata sollevata, confermando il corretto funzionamento delle funzionalità testate. I risultati positivi dei test forniscono una conferma affidabile dell'integrità del sistema e della robustezza delle operazioni implementate, come documentato nelle figure 20 e 21:

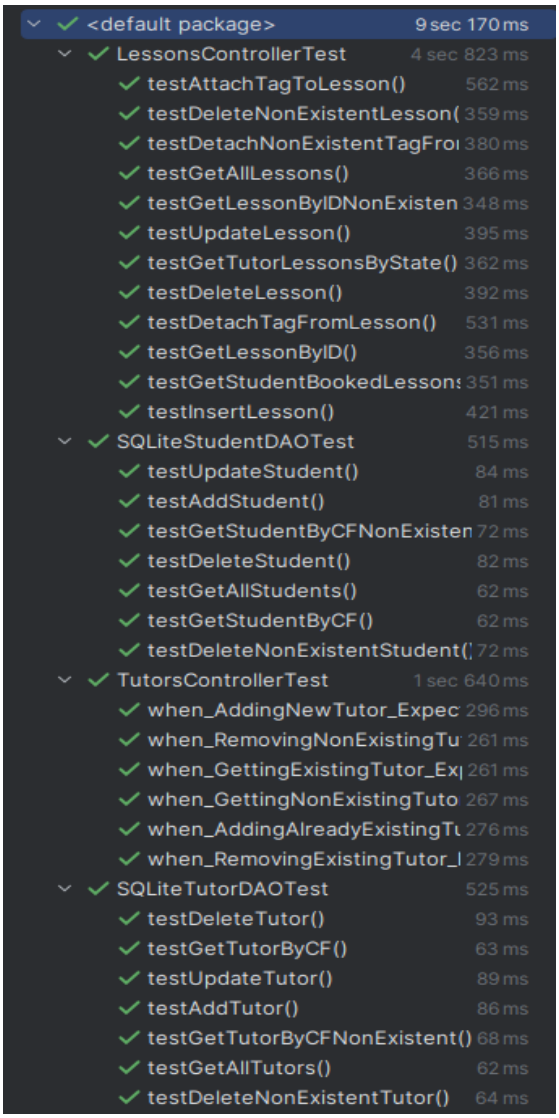

Figura 20: Risultati dei test, parte 1

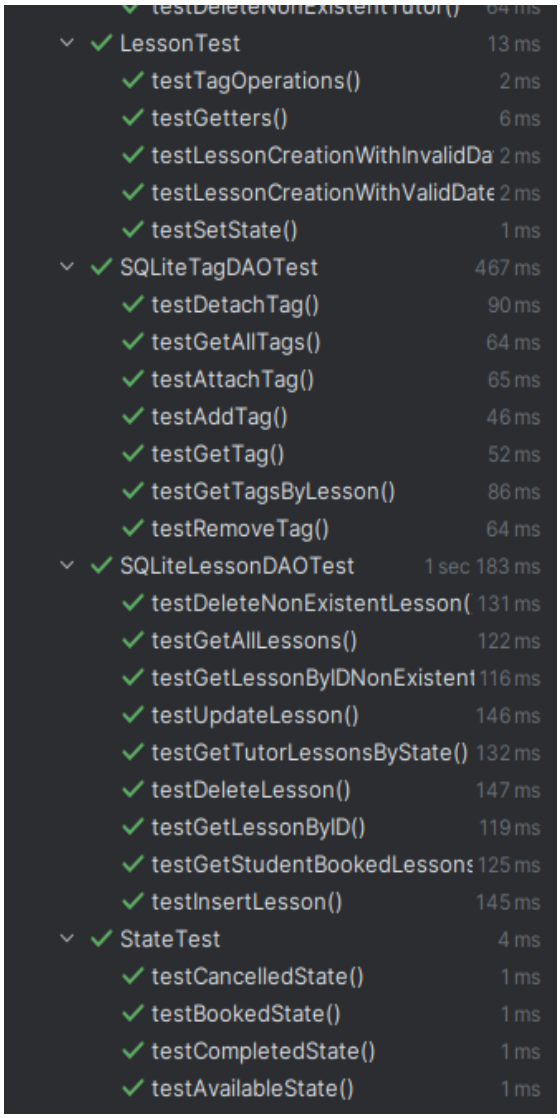

Figura 21: Risultati dei test, parte 2

Riferimenti bibliografici

[1] https://refactoring.guru/design-patterns/decorator

[2] https://it.wikipedia.org/wiki/Decorator

[3] https://it.wikipedia.org/wiki/State_pattern#/media/File:State_design_pattern.png

[4] https://it.wikipedia.org/wiki/State_pattern

[5] https://it.wikipedia.org/wiki/Data_Access_Object

[6] https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[7] https://medium.com/@stefanovskyi/unit-test-naming-conventions-dd9208eadbea