UNIVERSITÀ
DEGLI STUDI
FIRENZE

Corso di laurea triennale in ingegneria informatica

Ingegneria del software

PIATTAFORMA DI GESTIONE PER LA PRENOTAZIONE DI VISITE MEDICHE

Autori: Gianmarco De Laurentiis, Lorenzo Fedi
Docente: Vicario Enrico

Febbraio 2025

Indice

1 Introduzione

1.1 Obiettivo e descrizione del progetto

Il nostro progetto è incentrato sulla creazione di una piattaforma avanzata per la gestione di visite mediche, progettata per soddisfare le esigenze tanto dei pazienti quanto dei medici (libero professionisti o di strutture sanitarie). Questa piattaforma offre un ambiente virtuale dinamico e interattivo in cui pazienti e medici possono connettersi, collaborare e organizzare visite personalizzate in vari ambiti di specializzazione. Indipendentemente dal tipo di visita o dal contesto clinico, la nostra piattaforma offre un'opportunità per i pazienti di ricevere assistenza in modo personalizzato e per i medici di condividere le proprie competenze in modo flessibile e remunerativo. Gli attori principali nella nostra piattaforma sono i pazienti, alla ricerca di supporto sanitario su misura, e i medici, che offrono le proprie competenze in diverse specializzazioni. La piattaforma facilita l'incontro tra domanda e offerta di visite mediche, semplificando il processo di prenotazione e pagamento attraverso un sistema di transazioni online sicure.

- I medici possono creare annunci per visite (con la possibilità di modificarli o cancellarli in un secondo momento), specificando dettagli cruciali quali specializzazione, orario, modalità (online o in presenza) e tariffa. Inoltre, hanno la facoltà di visualizzare gli annunci pubblicati e consultare il calendario delle visite prenotate dai pazienti.
- I pazienti possono cercare annunci di visite disponibili, prenotare la visita secondo le proprie esigenze e gestire le proprie prenotazioni in modo comodo e intuitivo.

Di seguito viene riportato il codice sorgente dell'elaborato e cliccando sopra la scritta è possibile accedere al repository di github del progetto.

SOURCE CODE ON GITHUB

1.2 Architettura e pratiche utilizzate

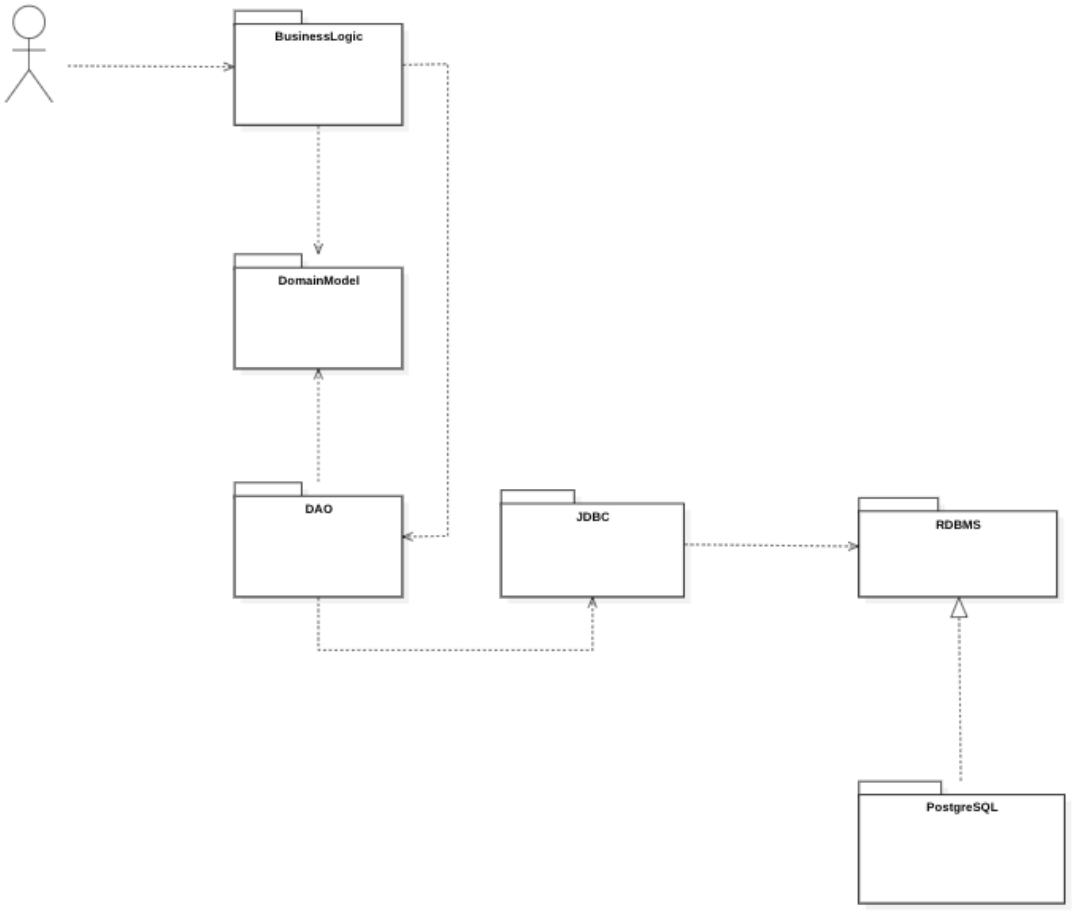

Figura 3: Diagramma delle dipendenze

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 *PostgreSQL* tramite *JDBC (Java DataBase Connectivity)*.

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

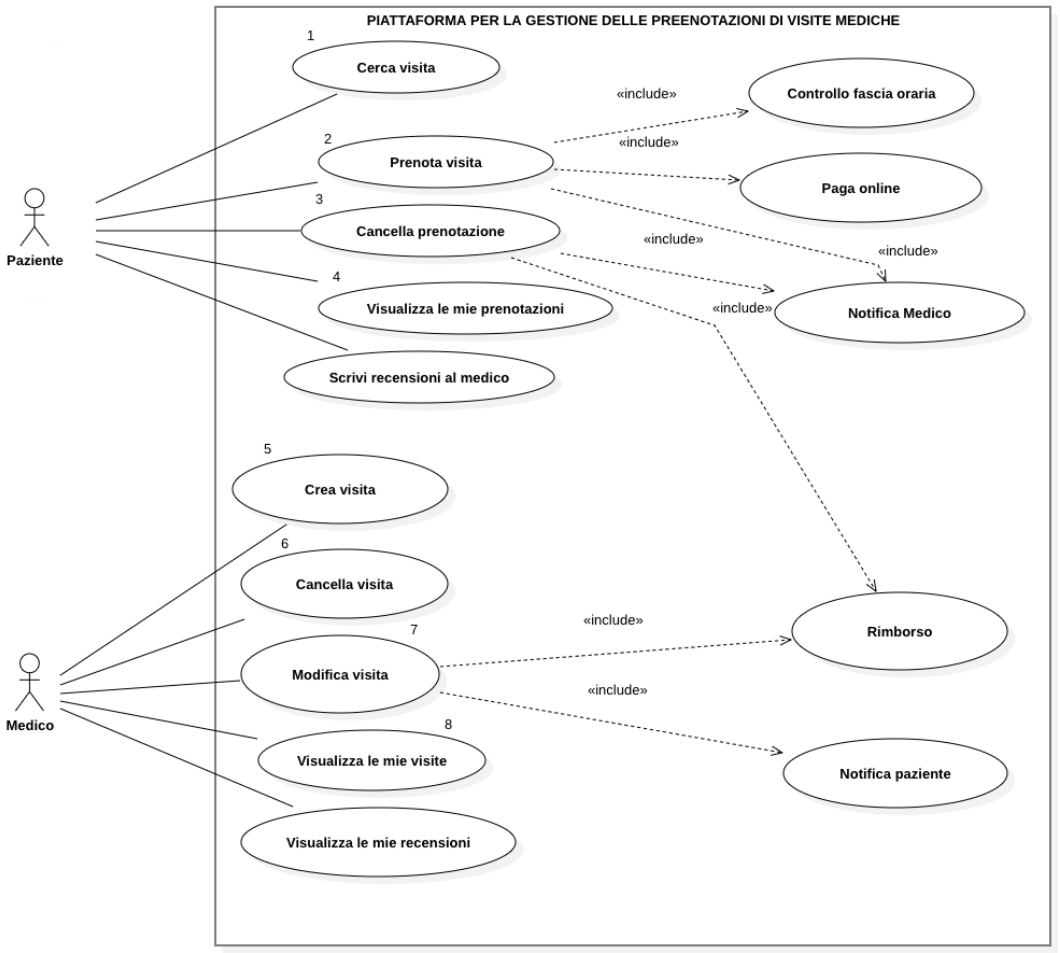

Figura 4: Diagramma dei casi d'uso

All'interno del sistema sono definiti due attori principali: Pazienti e Medici. 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 alle recensioni, ai pagamenti e alle notifiche siano stati inseriti nel diagramma, al momento non sono stati implementati nella

versione attuale del gestionale, ma sono stati progettati come possibili estensioni future del sistema.

I pazienti hanno la possibilità di effettuare diverse azioni nel sistema. Possono eseguire ricerche per trovare visite mediche disponibili, prenotarle, annullare le prenotazioni effettuate e visualizzare un elenco delle visite a cui parteciperanno. D'altra parte, i medici hanno la possibilità di creare nuovi slot per le visite, cancellarli o modificarli in un secondo momento. Inoltre, hanno accesso allo storico completo delle visite, 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 visite mediche 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 medici e pazienti con esso.

Use case 1	**Cerca Visita**
Description	Il paziente cerca la visita di suo interesse
Level	User goal
Actors	Paziente
Basic course	1. Il paziente inserisce i dati della ricerca 2. Vengono visualizzati i risultati di ricerca
Alternative course	2. La ricerca non produce nessun risultato

Figura 5: Cerca visita

Use case 2	**Prenota visita**
Description	Il paziente visualizza la visita e si prenota
Level	User goal
Actors	Paziente
Basic course	1. Il paziente seleziona una visita disponibile 2. Il paziente prenota 3. Il paziente paga la visita
Alternative course	1. Il paziente non può prenotarsi perché ha un'altra visita in quell'orario
Post-conditions	1. Il medico viene notificato dell'avvenuta prenotazione. 2. La visita appena prenotata appare nella sezione "Le mie prenotazioni" del paziente. 3. La visita appena prenotata appare nel calendario del medico 4. La visita non appare più nelle ricerche della piattaforma e il suo stato passa a "Prenotato"

Figura 6: Prenota visita

Use case 3	**Cancella prenotazione**
Description	Il paziente cancella la prenotazione ad una visita.
Level	User goal
Actors	Paziente
Basic course	1. Il paziente va nella sezione “Le mie prenotazioni”. 2. Il paziente seleziona la prenotazione desiderata. 3. Il paziente cancella la prenotazione. 4. Il paziente viene reindirizzato alla sezione “Le mie prenotazioni”.
Alternative course	3. Il paziente non può cancellare la visita perché troppo vicina alla data di essa.
Pre-conditions	Deve esistere almeno una visita alla quale il paziente è prenotato.
Post-conditions	1. Il medico viene notificato dell’avvenuta cancellazione 2. Il paziente viene rimborsato 3. La prenotazione scompare dalla sezione “Le mie prenotazioni” del paziente. 4. La visita scompare dal calendario del medico e il suo stato torna a “Disponibile”

Figura 7: Cancella prenotazione

Use case 4	**Visualizza le mie prenotazioni**
Description	Il paziente visualizza le visite a cui è prenotato.
Level	User goal
Actors	Paziente
Basic course	1. Il paziente 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 prenotazioni (del paziente)

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

Figura 9: Crea visita

Use case 6	**Cancella visita**
Description	Il medico cancella una visita.
Level	User goal
Actors	Medico
Basic course	1. Il medico va nella sezione “il mio Calendario”. 2. Il medico seleziona una visita. 3. Il medico cancella la visita selezionata.
Alternative course	3. Il medico non può cancellarla perché troppo vicina ad essa.
Pre-conditions	Il medico deve avere almeno una visita nel calendario.
Post-conditions	1. Se la visita è nello stato “Prenotata”, allora il paziente viene notificato dell’avvenuta cancellazione e rimborsato. 2. Lo stato della lezione permuta in “Cancellata”. 3. La visita non compare più nella sezione “le mie Prenotazioni” del paziente.

Figura 10: Cancella visita

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

Figura 11: Modifica visita

Use case 8	Visualizza le mie visite
Description	Il medico visualizza lo storico delle sue visite.
Level	User goal
Actors	Medico
Basic course	1. Il medico va nella sezione “il mio Calendario”. 2. Viene mostrato l’elenco delle visite.
Alternative course	2. Non è presente nessuna visita nello storico.

Figura 12: Visualizza le mie visite (storico del medico)

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 *VisitsController* per la gestione delle visite mediche, *PatientsController* per i pazienti e *DoctorsController* per i medici.
  - È presente anche una classe astratta *PeopleController* che viene implementata nei controllori dei pazienti e dei medici 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 alla sezione 2.3.1), consentendo l’applicazione di una serie di filtri. Questa approfondita

strategia di ricerca rende il processo di individuazione delle lezioni estremamente user-friendly e altamente efficiente per i pazienti

  - `state`: Questo package è dedicato alla gestione dinamica del comportamento di una visita 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 visita può esistere in quattro stati distinti: *available*, *booked*, *cancelled* e *completed*. Per dettagli specifici sul funzionamento di questi stati, si rimanda alla sezione 2.3.2
  - `tags`: Questo package consente l'associazione di tag specifici alle visite, fornendo dettagli quali il livello di urgenza, il soggetto, la zona geografica e la modalità della visita (online o in presenza). Tali informazioni extra permettono una categorizzazione avanzata delle visite, agevolando la ricerca e la selezione dei pazienti. In sostanza, il package *domainModel.Tags* si sposa armoniosamente con il pattern Decorator, arricchendo le visite con informazioni specifiche che possono essere approfondite nella sezione 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. 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.

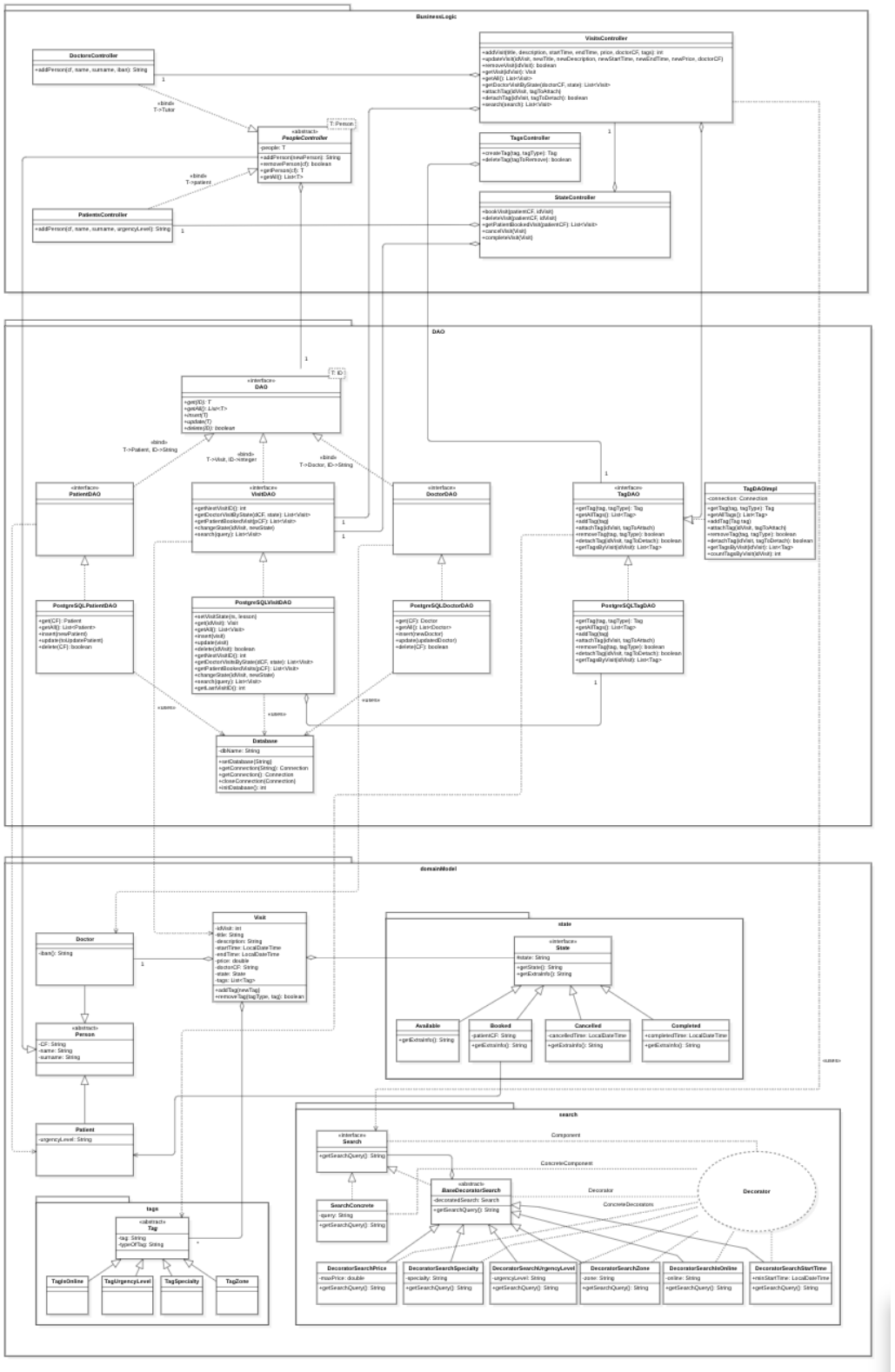

Figura 13: Diagramma delle classi

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

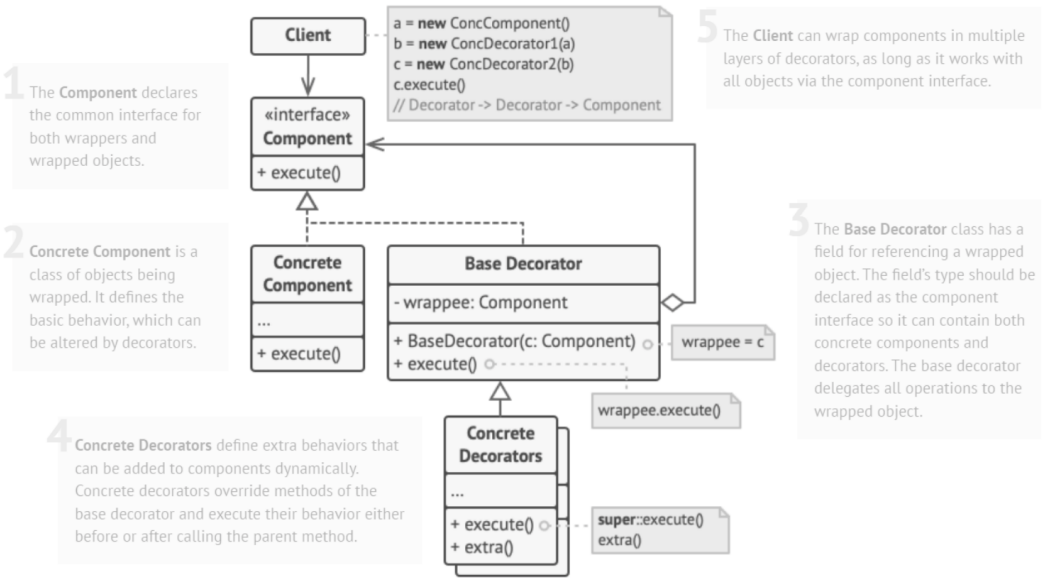

Figura 14: Pattern Decorator Generalizzato
(1)

Per ottimizzare e semplificare la ricerca delle visite per i pazienti, 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 visite tramite diversi filtri, tra cui *specializzazione del medico, livello di urgenza, 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 visite mediche di cardiologia a Firenze. Questo obiettivo può essere raggiunto combinando la classe di ricerca di base, nota come SearchConcrete, con i decoratori DecoratorSearchSpecialty e DecoratorSearchZone. Questa concatenazione di decoratori consente agli studenti di effettuare ricerche altamente specifiche, affinando le opzioni in base alle loro esigenze uniche (si rimanda al diagramma delle classi in figura 13).

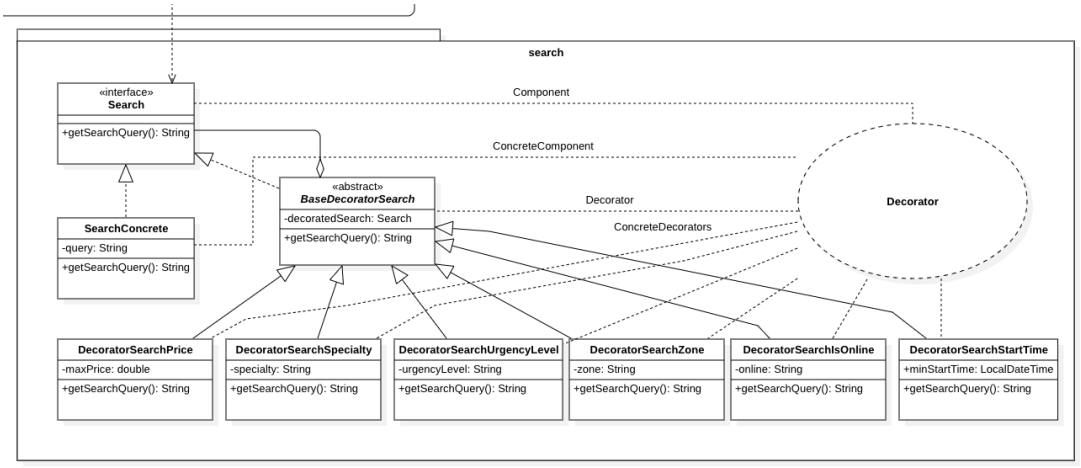

Figura 15: Pattern Decorator

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

2.3.2 State pattern

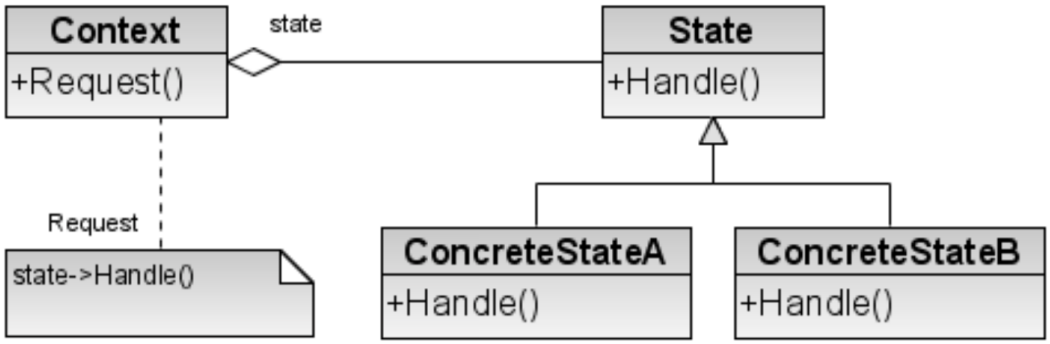

Figura 16: Pattern State Generalizzato
(3)

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.

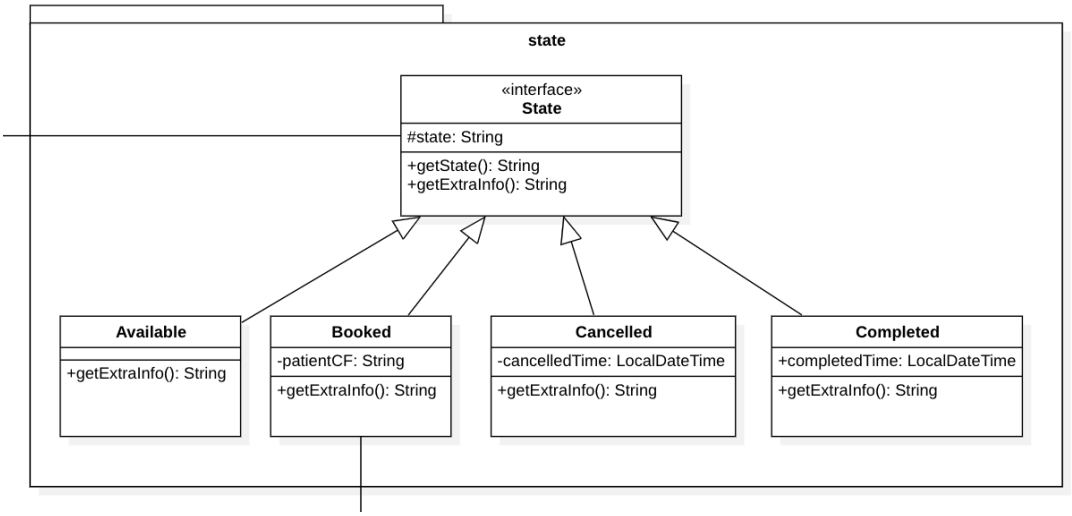

Figura 17: Pattern State

Come si vede in figura 17 il *context* è rappresentato dalla classe *Visit* (in cui ci sarà un campo state di tipo "State" e 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 visita:

1. `Available`: Rappresenta lo stato in cui una visita è disponibile per la prenotazione.
2. `Booked`: Indica che una visita è stata prenotata da un paziente.
3. `Cancelled`: Segnala che una visita è stata cancellata.
4. `Completed`: Indica che una visita 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 visite, 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 visita può essere adattato in modo dinamico, rispondendo ai cambiamenti di stato senza complicare eccessivamente la logica dell'applicazione.

2.3.3 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 PostgreSQL, 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).

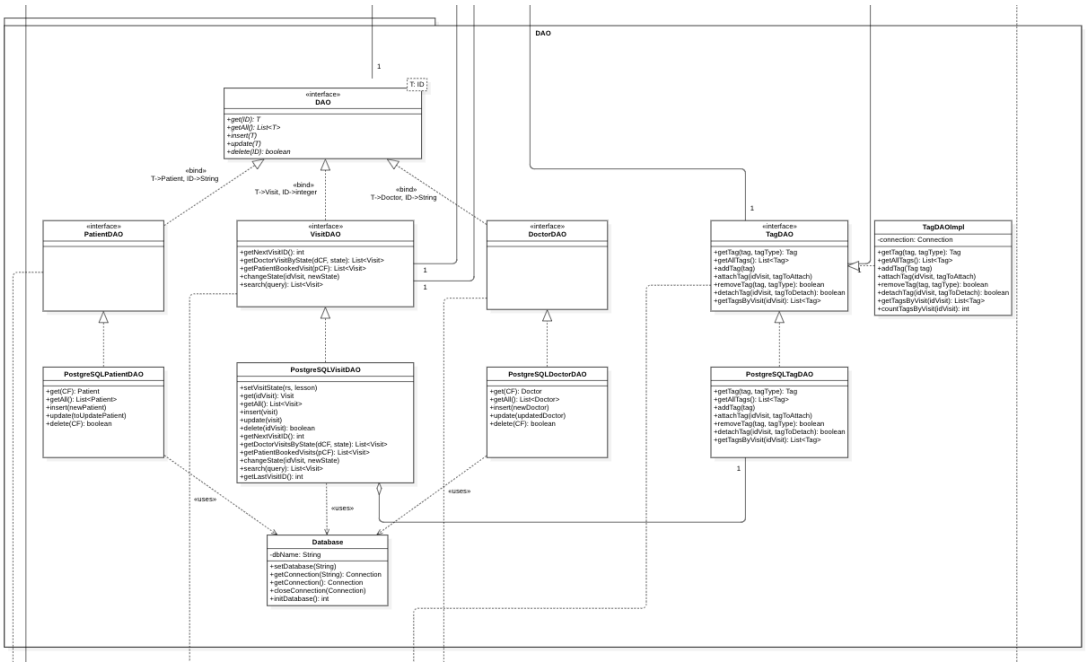

Figura 18: DAO

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 18, 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. PatientDAO, VisitDAO, DoctorDAO 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 ulteriori metodi specifici per ciascuna entità.

Nel dettaglio, l'interfaccia TagDAO contiene operazioni specializzate relative alla gestione dei tag, come l'associazione dei tag con le visite 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 PostgreSQLPatientDAO, PostgreSQLDoctorDAO, PostgreSQLTag-

DAO e PostgreSQLVisitDAO, implementano le interfacce specifiche. Ogni classe concreta è responsabile della gestione delle operazioni di accesso al database PostgreSQL per l'entità corrispondente. Per esempio, PostgreSQLTagDAO si occupa dell'associazione tra i tag e le visite, mentre PostgreSQLVisitDAO gestisce gli stati delle visite in modo separato.

2.3.4 Tags

Il package *Tags* svolge un ruolo cruciale nella categorizzazione e classificazione avanzata delle visite, contribuendo a una gestione più efficace delle informazioni associate alle visite stesse. Questo package consente l'associazione di tag specifici alle visite, fornendo dettagli come il livello di urgenza, la specializzazione, la zona geografica e la modalità della visita (online o in presenza). Queste informazioni extra arricchiscono il contesto delle visite, facilitando la ricerca e la selezione dei pazienti interessati. Ovviamente la classe *Visit* nel package domainModel ha una lista di oggetti Tag come attributo.

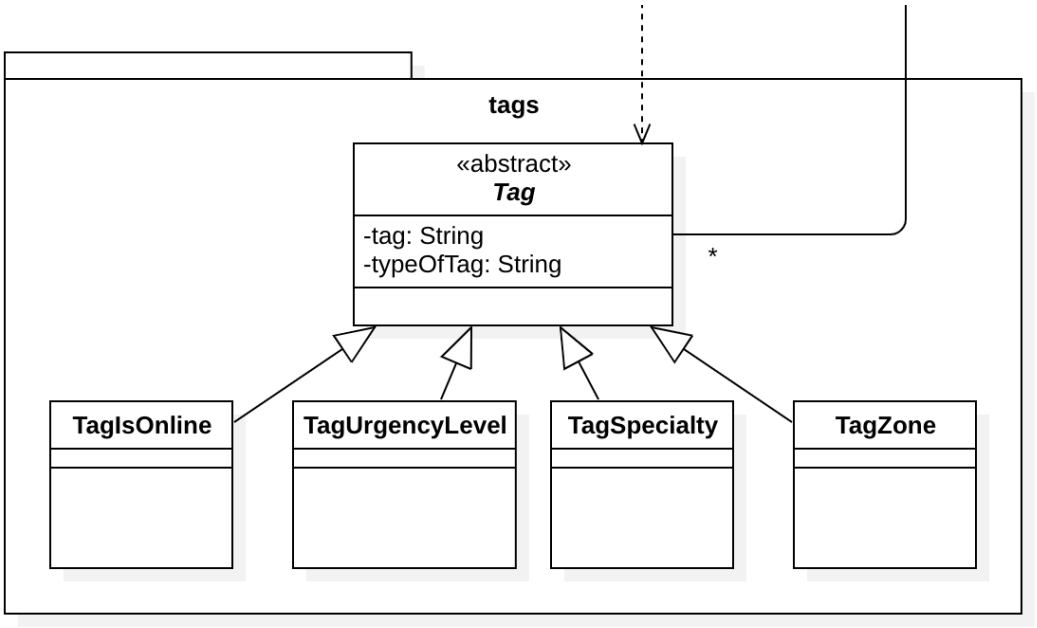

Figura 19: Package Tags

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, TagUrgencyLevel, TagSpecialty, 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 Visit. 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 20 è rappresentata la struttura del database relazionale implementato utilizzando il linguaggio SQL con il motore PostgreSQL. Il DB è progettato per gestire diverse entità chiave nel contesto delle visite mediche: doctor (`Doctors`), patient (`Patients`), visit (`Visits`), tag (`Tags`) e la tabella di associazione `VisitsTags`.
Ora verranno analizzate e spiegate nel dettaglio alcune scelte implementative, senza specificare quelle ovvie, ricavabili dal diagramma ER.

- **Tabella "Visits"**: *Foreign key* = doctorCF. Questa scelta di collegare questo attributo alla *Primary key* della tabella `Doctors`, permette di verificare l'integrità referenziale; ogni visita è associata a un medico esistente, evitando l'orfanità dei dati e garantendo che ogni visita sia gestita da un medico valido.
- **Tabella "VisitsTags"**: *Foreign key 1* = tag, tagType, *Foreign key 2* = idVisit. Si tratta di una tabella di collegamento che stabilisce una relazione molti a molti tra le visite e i tags. Ha un ruolo fondamentale nella gestione del sistema di visite mediche. Ogni riga in questa tabella indica che una determinata visita è associata a un certo tag. La tabella visitsTags serve a consentire l'associazione flessibile e dinamica di uno o più tag a ciascuna visita. In altre parole, permette di categorizzare ogni visita in base a vari criteri come la specializzazione, la modalità (online o presenziale), la zona geografica e così via. Alcuni aspetti degni di nota:

  1. *Flessibilità nell'associazione dei tag*: Le visite possono avere diversi tag, e questi possono variare ampiamente. Alcune visite potrebbero essere associate solo a una specializzazionee, mentre altre potrebbero avere multiple associazioni come specializzazione, modalità e zona geografica. Questa flessibilità è fondamentale per gestire una vasta gamma di visite.
  2. *Facilita la ricerca e la selezione*: Grazie ai tag, i pazienti possono facilmente cercare visite basate su criteri specifici. Ad esempio, un paziente potrebbe cercare tutte le visite di ortopedia a Milano. La tabella VisitsTags facilita questo processo consentendo di identificare rapidamente le visite 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 "Patients"**: da notare la presenza degli attributi *state (TEXT NOT NULL)* e *stateExtraInfo (TEXT)*. Questo approccio elegante integra perfettamente lo stato della visita con le informazioni sul paziente 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 visite nel sistema. Per maggiori informazioni si rimanda alla sezione 2.3.3

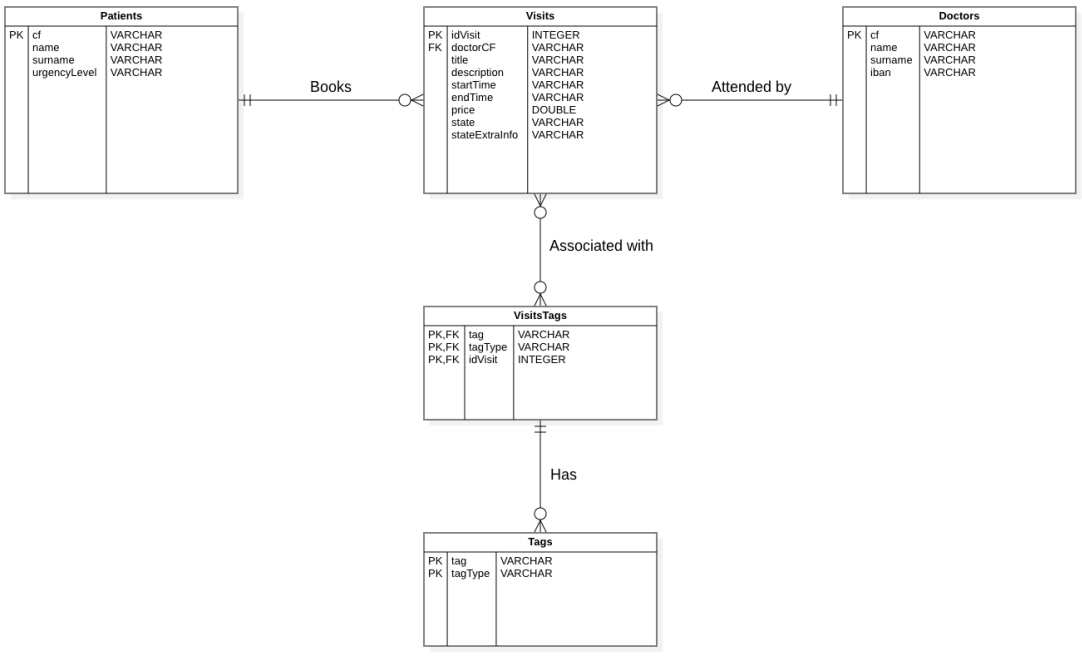

Figura 20: Diagramma Entity-Relationship del database

Qui di seguito viene fornito uno Snippet di codice che mostra la definizione di una delle tabelle presente nel diagramma:

Snippet 1: Esempio di definizione delle tabelle

```
/* Definisce la tabella visitsTags, che rappresenta la
    relazione molti-a-molti tra visite e tag. */

-- drop table if already exists
DROP TABLE IF EXISTS visitsTags
```

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

Perché è interessante?

- Permette di associare più tag a una visita senza duplicare dati nelle tabelle principali.
- Utilizza chiavi esterne con CASCADE, garantendo che, se una visita o un tag viene eliminato, la relazione venga aggiornata automaticamente.
- La chiave primaria composta (tag, tagType, idVisit) previene duplicati, assicurando che una visita non abbia lo stesso tag più volte.

3 Implementazione

Il progetto è stato realizzato in Java con l'uso dell'IDE IntelliJ, 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 in figura (21), e in questa sezione d'implementazione verranno descritti i contenuti di **src/main/java**.

Figura 21: Suddivisione del codice

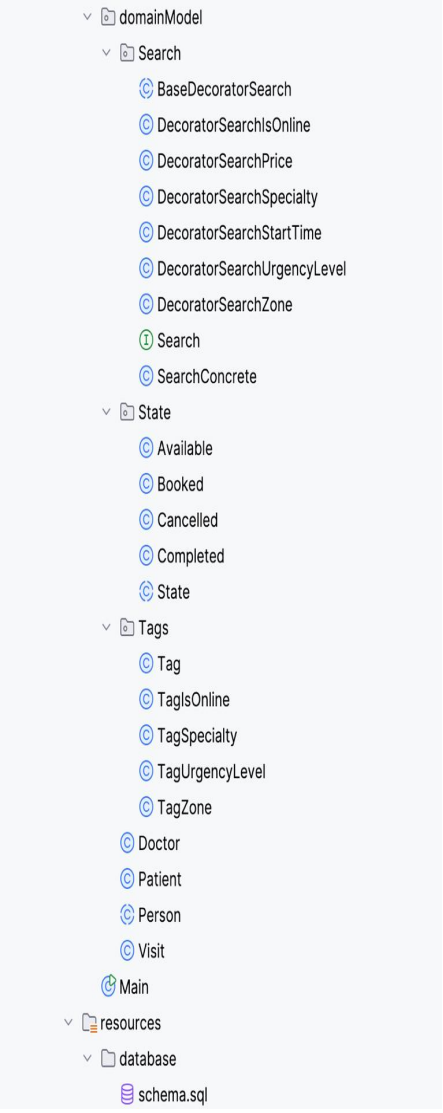

Figura 22: Zoom nel package domainModel

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, Patient, Doctor

Come si nota dal diagramma delle classi, la classe astratta *Person* rappresenta una persona generica con attributi come codice fiscale, nome e cognome. Le classi *Patient* e *Doctor* estendono questa classe per rappresentare rispettivamente pazienti e medici, aggiungendo l'attributo `urgencyLevel` per i pazienti e `iban` per i medici.

3.1.2 Visit

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

3.1.3 State package

Questo package gestisce gli stati delle visite, 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 del paziente che ha prenotato, il momento in cui è stata cancellata o completata la visita. Tutte e quattro le classi concrete estendono la classe base astratta *State*.

3.1.4 Tags package

Questo package gestisce i tag associati alle visite, come "Online", "urgencyLevel", "Specialty" e "Zone". Ogni tipo di tag è rappresentato da una classe separata (come *TagIsOnline, TagUrgencyLevel, TagSpecialty e TagZone*) che estende la classe astratta *Tag*. Questi tag consentono una categorizzazione dettagliata delle visite in base a criteri come la modalità online/offline, il livello di urgenza, la specializzazione e la zona geografica.

3.1.5 Search package

Il package *domainModel.Search* nel progetto gestisce la funzionalità di ricerca delle visite 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 visite disponibili.
- `SearchConcrete Class`: Questa classe implementa l'interfaccia Search e fornisce una query di base per selezionare le visite 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 visite secondo il loro criterio per cui sono state progettate. La query verrà modificata per includere solo le visite interessate. I decoratori sono: *DecoratorSearchSpecialty, DecoratorSearchStartTime, DecoratorSearchPrice, DecoratorSearchUrgencyLevel e DecoratorSearchIsOnline*.

In generale, questo sistema di classi consente di costruire query di ricerca complesse combinando diversi criteri. Ad esempio, se desideri trovare visite online (modalità di visita), di 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, PatientsController e DoctorsController

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

3.2.2 VisitsController

La classe *VisitsController* gestisce le operazioni legate alle visite nel sistema. Essa comunica con il database e gestisce l'aggiunta, la modifica, la cancellazione e il recupero delle visite. Oltre a ciò, permette l'assegnazione di tag alle visite e ne gestisce la rimozione.
Nello snippet poco sotto si vede come nel metodo `addVisit()` viene effettuato un controllo sull'esistenza del medico associato alla visita. Successivamente, nello snippet 3, viene verificato che il medico non stia cercando di inserire una nuova visita in un orario in cui è già impegnato.

Questa procedura evita sovrapposizioni di visite e garantisce che il medico sia disponibile per condurre la visita richiesta:

Snippet 2: Metodo "addVisit()" di *VisitsController*

```
/*Questo snippet evidenzia un controllo essenziale nel
    processo di aggiunta di una visita. Prima di creare una
    nuova visita, il sistema verifica che il medico
    specificato esista nel database.*/

public int addVisit(...) throws Exception{
    Doctor doctor = doctorsController.getPerson(doctorCF);
    // @throws Exception If the doctor is not found
    if (doctor == null)
        throw new IllegalArgumentException("doctor not found
            ");
    ...
}
```

Perché è interessante?

- Evita errori legati a riferimenti non validi a medici inesistenti.
- Mostra come viene gestita l'eccezione con IllegalArgumentException.

Snippet 3: Metodo "addVisit()" di *VisitsController*

```
/*Qui viene implementata una verifica per evitare che un
    medico abbia due visite sovrapposte nello stesso orario.
    */

public int addVisit(...) throws Exception{
    ...
    //throws IllegalArgumentException If the doctor is
        already occupied in the given time range

    for (Visit v : this.visitDAO.getAll()) {
        if (v.getDoctorCF().equals(doctorCF)) {
            if ((v.getStartTime().isBefore(endTime) || v.
                getStartTime().equals(endTime))
                    && (v.getEndTime().isAfter(startTime) ||
```

```
                        v.getEndTime().equals(startTime)))
                throw new RuntimeException("...");
            }
        }
        ...
}
```

Perché è interessante?

- Dimostra l'uso di una validazione lato server per garantire la consistenza dei dati.
- Mostra come il sistema impedisca conflitti di prenotazione.

3.2.3 StateController

La classe *StateController* gestisce gli stati delle visite, consentendo di prenotare una visita, cancellare una prenotazione, segnare una visita come completata o eliminarla . Interagisce con la classe *VisitsController* per ottenere e modificare le informazioni sullo stato delle visite. Questa classe gestisce anche la validità delle operazioni, garantendo che un'azione possa essere eseguita solo se la visita 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*

```
/*Questo metodo gestisce la cancellazione di una
    prenotazione verificando prima che la visita e il
    paziente esistano.*/

public void deleteBooking(String patientCF, int idVisit)
    throws Exception {
    // Retrieve the visit associated with the given ID.
    Visit visit = visitsController.getVisit(idVisit);
    // If the visit does not exist, throw an exception.
    if (visit == null) {
        throw new IllegalArgumentException("The␣given␣visit␣
            ID␣does␣not␣exist.");
    }

    // Retrieve the patient associated with the given Fiscal
        Code.
```

```
    Visit visit = visitsController.getPerson(patientCF);
    // If the visit does not exist, throw an exception.
    if (visit == null) {
        throw new IllegalArgumentException("The␣given␣
            patient␣does␣not␣exist.");
    }

    // Check if the visit is in the 'Booked' state.
    if (!Objects.equals(visit.getState(), "Booked")) {
        // If the visit 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 patient associated with the booking
        attempt is the actual booker.
    if (Objects.equals(visit.getStateExtraInfo(), patieentCF
        )) {
        // If the patient is the actual booker, change the
            visit state to 'Available'.
        Available av = new Available();
        this.visitDAO.changeState(idVisit, av);
    } else {
        // If the visit is not the actual booker, throw an
            exception.
        throw new RuntimeException("Patient␣" + patientCF +
            "␣is␣not␣booked␣for␣visit␣#" + idVisit + ".");
    }
}
```

Perché è interessante?

- Mostra l'importanza della validazione dei dati prima di eseguire modifiche critiche.
- Evita errori dovuti a tentativi di cancellare prenotazioni inesistenti.

3.2.4 TagsController

La classe *TagsController* gestisce i tag associati alle visite. Permette la creazione di nuovi tag e la loro associazione a specifiche visite. 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 PostgreSQL. Sono state sviluppate dunque classi DAO concrete che stabiliscono e gestiscono la connessione con il database PostgreSQL. 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 visite, in particolare viene mostrato il metodo con il quale si ottiene una specifica visita:

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

```
/*Mostra l'implementazione di una query SQL per recuperare
    una visita dal database.*/

public class PostgreSQLVisitDAO implements VisitDAO{
    private final TagDAO tagDAO;

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

   // Get a specific visit
    @Override
    public Visit get(Integer idVisit) throws SQLException {
    String query = "SELECT␣*␣FROM␣visits␣WHERE␣idVisit␣=␣?"
    try(Connection conn = Database.getConnection();
        PreparedStatement ps = conn.prepareStatement(query))
            {
        ps.seetInt(1, idVisit);
        try(ResultSet rs = ps.executeQuery()){
            if(rs.next()){
               Visit visit = new Visit(
               ....
            }
```

```
        }
    }
    return null;
}
```

Perché è interessante?

- Mostra l'uso del pattern DAO per separare la logica di accesso ai dati.
- Utilizza PreparedStatement per prevenire SQL injection.

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

```
/*Questo snippet mostra la configurazione della connessione
    al database usando JDBC.*/

public class Database {

  private static String URL = "jdbc:postgreSQL://localhost
      :5432/GestionaleVisiteMediche";
  private static final String USER = "postgres";
  private static final String PASSWORD = "1234";

  private Databasee(){}

  public static Connection getConnection() throws
      SQLeException{
  return DriverManager.getConnection(URL, USER, PASSWORD);
  }
}
```

Perché è interessante?

- Definisce un singleton per gestire la connessione.
- Centralizza i parametri di connessione, facilitando eventuali modifiche.

In particolare è bene citare ed osservare come nell'implementazione dell'elaborato siano state inserite due connessioni a due database distinti: uno per il codice sorgente (*GestionaleVisiteMediche*) e uno per lo unit testing (*GestionaleVisiteMedicheTest*).

Questa strategia è stata adoperata con lo scopo di poter garantire un maggiore ordine all'interno del codice per renderlo più "pulito" ma anche per evitare possibili conflitti fra i database stessi garantendo più integrità fra i vari dati.

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 `Visit` e `State`.

- **VisitTest.java**:
    1. *testVisitCreationWithValidDates()*: Questo test verifica che la creazione di una visita con date valide non generi eccezioni. Assicura che il costruttore di Visit gestisca correttamente le date di inizio e fine.
    2. *testVisitCreationWithInvalidDates()*: Questo test verifica che la creazione di una visit 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 Visit per assicurarsi che restituiscano i valori corretti per i diversi attributi della visita.

4. *testSetState()*: Questo test verifica il metodo setState() di Visit, che imposta lo stato della visita. Assicura che lo stato e le informazioni extra siano impostati correttamente.
5. *testTagOperations()*: Questo test verifica i metodi di gestione dei tag di Visit. Verifica che i tag possano essere aggiunti correttamente alla visita, 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 del paziente come informazione extra.

Snippet 7: Esempio di unit test di Visit

```
/*Test unitario per verificare il corretto funzionamento
    delle operazioni sui tag.*/

    @Test
    void testTagOperations() {
        // Verifica i metodi di gestione dei tag
        TagZone tagZone = new TagZone("Firenze");
        TagSpecialty tagSpecialty = new TagSpecialty( "
            Pneumologia");

        // Aggiungi un tag
        visit.addTag(tagZone);
        visit.addTag(tagSpecialty);
        Assertions.assertTrue(visit.getTags().contains(
            tagZone));

        // Rimuovi un tag
        visit.removeTag("Zone", "Firenze");
        Assertions.assertFalse(visit.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, () -> visit.removeTag("InvalidType", "
            Tag1"));

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

```
        Assertions.assertFalse(removed);
    }
```

Perché è interessante?

- Verifica l'aggiunta e la rimozione dei tag.
- Controlla la gestione degli errori con Assertions.assertThrows().

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 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:

- **VisitsControllerTest.java**:
  1. *testGetVisitByID()*: Questo test verifica se è possibile ottenere una visita dal database utilizzando l'ID della visita. Verifica che i dettagli della visita recuperata corrispondano ai valori inseriti durante la configurazione del database di test.
  2. *testInsertVisit()*: Questo test verifica se è possibile inserire una nuova visita nel database. Verifica anche se l'ID della visita assegnato è valido e se la visita è stata inserita correttamente nel database.
  3. *testUpdateVisit()*: Questo test verifica se è possibile aggiornare i dettagli di una visita nel database e se le modifiche sono state applicate correttamente.
  4. *testDeleteVisit()*: Questo test verifica se è possibile eliminare una visita dal database utilizzando l'ID della visita. Verifica anche se la visita è stata effettivamente eliminata dal database.
  5. *testAttachTagToVisit()*: Questo test verifica se è possibile allegare un nuovo tag a una visita esistente nel database. Il test verifica se il tag è stato allegato correttamente alla visita.
  6. *testDetachTagFromVisit()*: Questo test verifica se è possibile staccare un tag da una visita esistente nel database. Il test verifica se il tag è stato rimosso correttamente dalla visita.
  7. *ecc...*
- **DoctorsControllerTest.java**:
  1. *when_AddingNewDoctor_Expect_Success()*: Questo test verifica se è possibile aggiungere un nuovo medico al database. Verifica se il medico è stato aggiunto correttamente e se è possibile recuperarlo dal database.
  2. *when_AddingAlreadyExistingDoctor_Expect_Exception()*: Questo test verifica se il sistema gestisce correttamente il caso in cui si cerca di aggiungere un medico già esistente nel database.

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

Snippet 8: Esempio di unit test in VisitControllerTest

```
/*Test unitario per verificare che una visita venga inserita
     correttamente nel database.*/

    @Test
    void testInsertVisit() throws Exception {
        List<Tag> tags = new ArrayList<>();
        TagSpecialty ts = new TagSpecialty("Cardiologia");
        tags.add(ts);
        int visitId = visitsController.addVisit("Cardiology␣
            check␣up", "first␣visit", LocalDateTime.now(),
            LocalDateTime.now(), 40.0, "doctor2", tags);

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

        // Assicurati che la visita sia stata inserita
            correttamente nel database
        Visit insertedVisit = visitsController.getVisit(
            ivisitId);
        assertNotNull(insertedVisit);
    }
```

Perché è interessante?

- Controlla che il metodo addVisit() funzioni correttamente.
- Verifica che l'ID della visita sia valido e che l'oggetto sia stato recuperato con successo.

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 *testDeleteVisit()* del file **PostgreSQLVisitDAOTest.java**. In sintesi, il test verifica che il metodo `delete()` di *PostgreSQLVisitDAO* elimini correttamente una visita dal database e che l'elenco delle visite 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 PostgreSQLVisitDAOTest.java

```
/*Verifica che il metodo delete() rimuova correttamente una
    visita dal database.*/

    @Test
    void testDeleteVisit() throws Exception {
        // Test the delete method to delete a visit
        boolean result = visitDAO.delete(1);
        Assertions.assertTrue(result);
        List<Visit> visits = visitDAO.getAll();
        Assertions.assertEquals(1, visits.size());
    }
```

Perché è interessante?

- Conferma che la visita venga effettivamente eliminata.
- Utilizza assertEquals() per verificare che il numero di visite nel database sia corretto dopo l'eliminazione.

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.

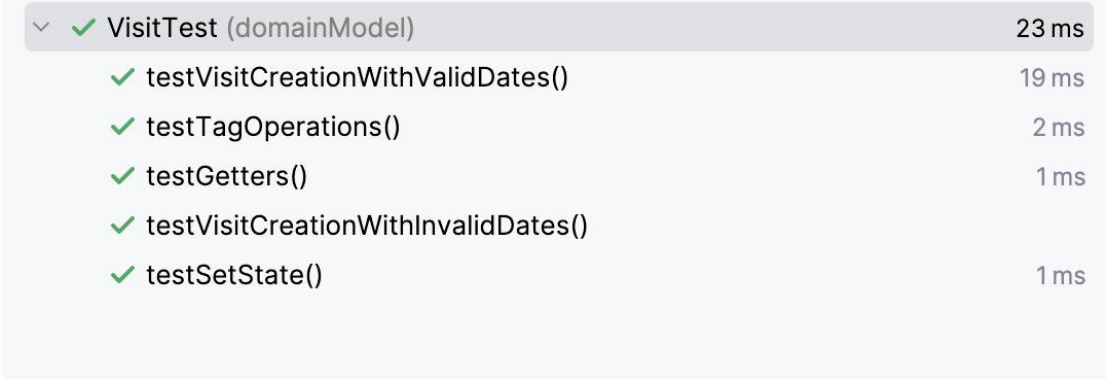

Figura 23: risultati dei test parte 1

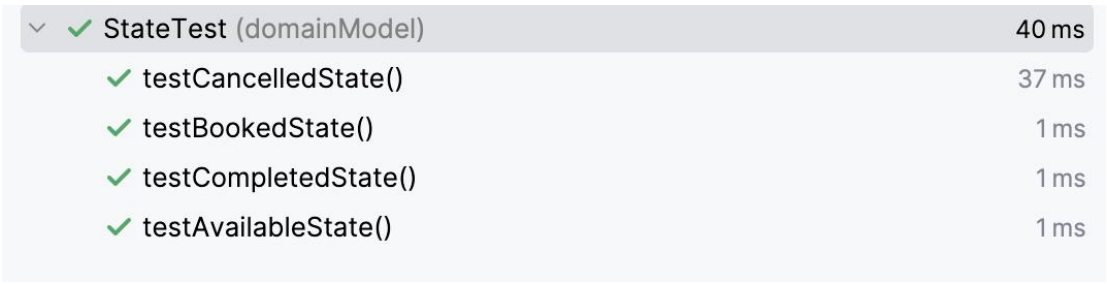

Figura 24: risultati dei test parte 2

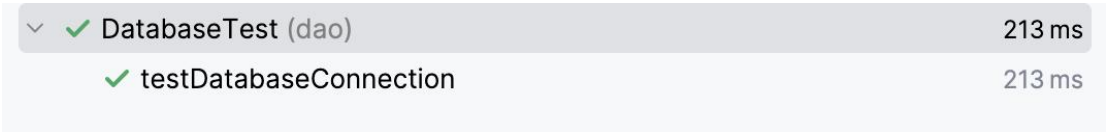

Figura 25: risultati dei test parte 3

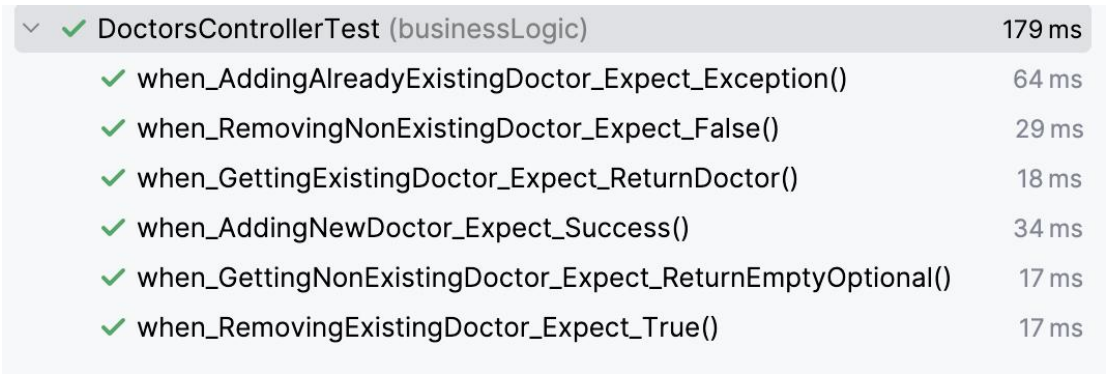

Figura 26: risultati dei test parte 4

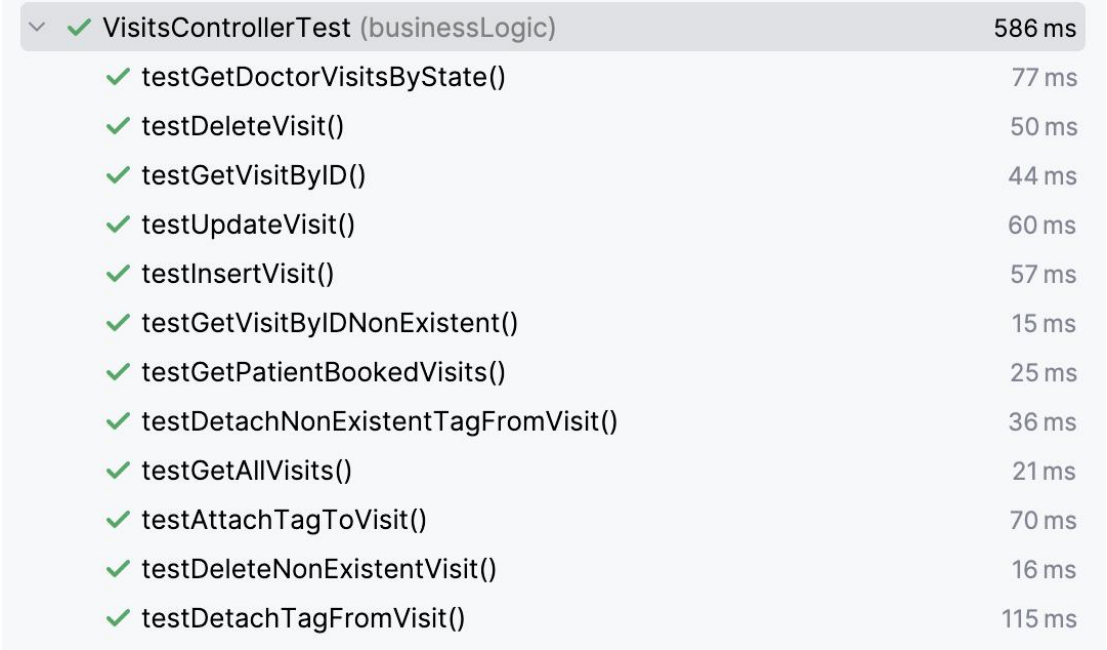

Figura 27: risultati dei test parte 5

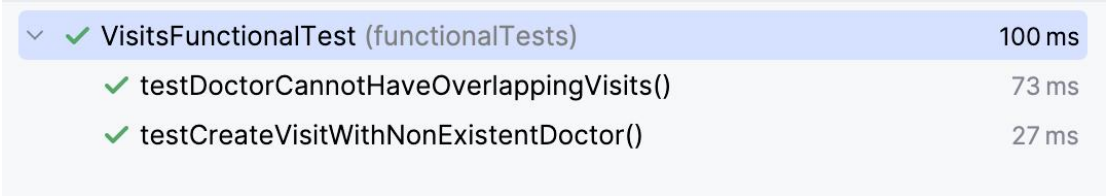

Figura 28: risultati dei test parte 6

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