UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione e implementazione di un software per la gestione di una biblioteca

Design and implementation of a software for the management of a library

Relatore

Prof. Domenico Ursino

Candidato

Alessandro Stella

ANNO ACCADEMICO 2023-2024

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Il seguente studio illustra dettagliatamente lo sviluppo di un software gestionale, partendo dalla fase di ideazione fino alla sua effettiva implementazione. I software gestionali rappresentano un'importante risorsa per la gestione efficiente di una vasta gamma di attività, sia in ambito professionale che non. Grazie all'automazione dei processi, questi sistemi permettono di eliminare l'uso di tecnologie obsolete, ridurre i margini di errore e ottimizzare i tempi operativi.

Alla base dello sviluppo di tali sistemi vi è la progettazione del software, che deve necessariamente tener conto delle metodologie e degli strumenti propri dell'Ingegneria del Software. L'Ingegneria del Software, infatti, è una disciplina che abbraccia ogni aspetto del ciclo di vita del software, dalla definizione dei requisiti alla manutenzione del sistema una volta implementato. Questa disciplina include non solo i processi tecnici di sviluppo, ma anche la gestione dei progetti, l'uso degli strumenti di supporto e l'applicazione di metodi e teorie volte a garantire la qualità e la sicurezza del software.

La presente tesi si compone di sette capitoli, ognuno dei quali tratta un aspetto specifico dello sviluppo del software:

- -) Nel capitolo 1 verranno presentati il contesto di riferimento ed i requisiti di sistema, con particolare attenzione ai servizi che il sistema deve offrire e ai vincoli operativi
- -) Nel capitolo 2 verrà descritta la modellazione dell'interazione tra l'utente e il sistema attraverso l'uso di diagrammi dei casi d'uso
- -) Nel capitolo 3 saranno introdotti i modelli strutturali del software, evidenziando l'organizzazione del sistema tramite diagrammi delle classi
- -) Nel capitolo 4 verrà descritta la progettazione della componente applicativa, con focus sugli strumenti e le metodologie dell'Ingegneria del Software.
- -) Nel capitolo 5 sarà illustrata l'implementazione del software, ovvero il processo di traduzione del progetto in un programma eseguibile.
- -) Nel capitolo 6 sarà presentata l'analisi dei test condotti per verificare la correttezza e affidabilità del sistema, con particolare enfasi sulla gestione dei bug
- -) Nel capitolo 7 verranno discusse le conclusioni della tesi.

CAPITOLO 1

Descrizione del contesto di riferimento e specifica dei requisiti

In questo primo capitolo verrà esaminata la fase iniziale dello sviluppo del software oggetto di questa tesi. Tale fase è cruciale per raccogliere quante più informazioni possibili, al fine di pianificare in modo ottimale lo sviluppo del programma e prevenire eventuali errori futuri. Questi, se presenti in un contesto professionale in cui il software è commissionato da un cliente, potrebbero tradursi in un inutile spreco di risorse sia economiche che temporali. Il punto di partenza è rappresentato dalla descrizione del contesto di riferimento, con l'obiettivo di definire chiaramente gli scopi e le funzionalità del software in base alle richieste e alle esigenze dello stakeholder committente. Successivamente, si procederà con la definizione dei requisiti del sistema, che definiscono i servizi che il software dovrà offrire e i vincoli operativi entro i quali dovrà funzionare. Infine, verrà presentata una tabella che metterà in relazione le specifiche del sistema con i casi d'uso, fornendo una panoramica chiara delle funzionalità e delle interazioni previste.

1.1 Descrizione del contesto di riferimento

Il software sviluppato nell'ambito di questo progetto è destinato alla gestione integrata di una biblioteca e si pone l'obiettivo di automatizzare, semplificare e migliorare l'efficienza delle varie attività che caratterizzano il funzionamento di tale ambiente. L'aumento della domanda di servizi digitali richiede una soluzione software che possa centralizzare e ottimizzare la gestione di risorse fisiche e digitali, permettendo una fruizione più agevole da parte degli utenti e un'amministrazione più efficace delle risorse disponibili.

Una delle principali funzionalità del sistema è la gestione delle prenotazioni dei posti in aula studio, che consente agli studenti di riservare postazioni di lavoro in modo facile e veloce attraverso un'interfaccia intuitiva. Tale funzionalità si rivela particolarmente utile in contesti ad alta affluenza, dove è necessario monitorare e regolare l'uso degli spazi per garantire un'equa distribuzione e ottimizzare l'utilizzo delle risorse. Parallelamente, il software si occupa della gestione dei prestiti di libri, sia fisici che digitali, permettendo agli utenti di accedere al catalogo della biblioteca, di verificare la disponibilità dei volumi e procedere con la prenotazione.

In aggiunta, il sistema consente di tracciare in maniera precisa le operazioni degli utenti e del personale della biblioteca. I dipendenti possono accedere al software per monitorare il flusso di risorse, elaborare nuove acquisizioni di libri, accedere allo storico delle prenotazioni e verificare le richieste di prestito in tempo reale. L'interfaccia del software è progettata per essere user-friendly e accessibile a tutti i livelli di competenza informatica, garantendo un'esperienza d'uso fluida per ogni categoria di utente: studenti, personale e amministratori.

L'elemento chiave del progetto è l'automazione delle operazioni ripetitive, che non solo riduce il carico di lavoro per i dipendenti della biblioteca, ma minimizza anche la possibilità di errori umani, garantendo maggiore precisione e affidabilità nella gestione delle risorse. Inoltre, il sistema è dotato di funzionalità per la gestione degli utenti e delle licenze del personale, permettendo di controllare l'accesso a determinate funzionalità in base ai ruoli e alle responsabilità assegnate.

1.2 Glossario

Nella Tabella 1.1 viene fornita la descrizione in linguaggio naturale del glossario dei termini utilizzati nel programma. Il glossario contiene tutti i concetti fondamentali dello scenario di riferimento con la relativa spiegazione.

Glossario		
Termine	Tipo	Significato
Libro	Tecnico	Supporto di lettura che può essere inserito, eliminato o dato in prestito.
Titolo	Tecnico	Titolo del supporto di lettura.
Categoria	Tecnico	Categoria alla quale appartiene il supporto di lettura.
Studente	Tecnico	Una parte che può prenotare libri o posti in biblioteca.
Posto	Tecnico	Una delle 100 parti a disposizione degli studenti nella biblioteca.
Prenotazione Posto	Operazione	Azione che permette di prenotare un posto.
Prenotazione Libro	Operazione	Azione che permette di prenotare un libro.
Dipendente	Tecnico	Una parte che lavora al'interno della biblioteca.
Licenza	Tecnico	Mansione che svolge un dipendente.

Tabella 1.1: Glossario dei termini utilizzati all'interno del software

1.3 Specifica dei requisiti

La specifica e l'analisi dei requisiti rappresentano una fase cruciale nello sviluppo di un sistema software, poiché stabiliscono con precisione le funzionalità, i vincoli operativi, le prestazioni e le interfacce che il sistema dovrà possedere per rispondere efficacemente alle esigenze del committente. Questa fase culmina nella redazione di un documento di Specifica dei Requisiti Software (SRS), che deve essere completo, accurato, coerente, privo di ambiguità e, nel caso in cui il software sia stato commissionato da uno stakeholder, comprensibile tanto per quest'ultimo quanto per lo sviluppatore. È essenziale notare che l'obiettivo di questa fase è definire "cosa" il sistema dovrà fare, evitando di descrivere "come" lo farà. Un SRS ben redatto rappresenta il punto di convergenza di tre prospettive distinte: lo stakeholder, l'utente finale e lo sviluppatore. Esso funge anche da riferimento per la validazione del prodotto finale e costituisce un requisito fondamentale per garantire la qualità del software. Un errore in questa fase si tradurrà in errori nel sistema finale, con conseguenti ripercussioni negative sui costi e sui tempi di sviluppo. L'ingegneria dei requisiti (Requisite Engineering - RE) è la fase iniziale del processo di sviluppo che si occupa di identificare, analizzare, documentare e verificare i requisiti del sistema. Sebbene sia una delle prime fasi, è cruciale perché fornisce una visione globale delle funzioni che il software deve implementare e dei benefici che offrirà.

I requisiti si suddividono in due categorie principali: requisiti funzionali e requisiti non funzionali. Queste due categorie verranno esaminate in dettaglio nelle prossime sottosezioni.

1.3.1 Requisiti funzionali

I requisiti funzionali definiscono i servizi specifici che il sistema deve fornire, specificando come il software deve rispondere a determinati input e gestire varie situazioni operative.

Di seguito sono riportati i principali requisiti funzionali che caratterizzano il software da realizzare nella presente tesi, divisi in quattro macrogruppi:

- Gestione studenti:
  - iscrizione degli studenti che desiderano utilizzare i servizi della biblioteca, come il prestito di libri e la prenotazione di posti;
  - visualizzazione dei dati relativi agli studenti registrati;
  - rimozione degli studenti non più iscritti.
- Gestione libri:
  - inserimento di libri e altri materiali nell'elenco degli articoli disponibili;
  - rimozione dei suddetti articoli di lettura dall'elenco dei supporti disponibili.
- Gestione dipendenti:
  - inserimento di nuovi dipendenti;
  - visualizzazione dei dati relativi ai dipendenti;
  - rimozione dei dipendenti dall'elenco del personale della biblioteca.
- Gestione prenotazioni:
  - gestione del prestito degli articoli da lettura;
  - gestione delle prenotazioni dei posti in aula studio;
  - monitoraggio delle scadenze dei prestiti;
  - monitoraggio delle prenotazioni in aula studio.

1.3.2 Requisiti non funzionali

I requisiti non funzionali rappresentano vincoli legati alle performance del sistema e alle modalità di erogazione dei servizi. Questi includono restrizioni temporali, conformità a standard e vincoli di sviluppo. I requisiti non funzionali realativi al sistema da analizzare sono i seguenti:

- implementazione del sistema: il sistema sarà sviluppato utilizzando il linguaggio Python 3;
- libro non disponibile: il sistema non deve permettere il prestito di un supporto di lettura non disponibile o già prenotato;
- posto non disponibile: il sistema non deve permettere la prenotazione di un posto già occupato;
- interfaccia grafica intuitiva e user-friendly.

1.4 Matrice di mapping dei requisiti

Il rapporto tra i casi d'uso e i requisiti funzionali è cruciale nella fase di analisi e progettazione di un sistema software. I casi d'uso rappresentano infatti un mezzo per descrivere le interazioni tra gli attori esterni (utenti o sistemi) e il sistema stesso, fornendo una visione chiara e strutturata di come il sistema debba rispondere a specifiche esigenze degli utenti. Ogni caso d'uso corrisponde a uno o più requisiti funzionali, descrivendo le azioni che il sistema deve essere in grado di eseguire per soddisfare tali requisiti. In altre parole, i requisiti funzionali definiscono cosa il sistema deve fare, mentre i casi d'uso descrivono come il sistema interagirà con gli utenti per soddisfare queste funzioni. Questo legame tra requisiti e casi d'uso permette di assicurare che tutte le funzionalità richieste siano correttamente implementate, riducendo il rischio di incomprensioni o lacune durante lo sviluppo. Nel prossimo capitolo saranno illustrati dettagliatamente i casi d'uso relativi allo sviluppo del software oggetto di questo studio, mentre nella Figura 1.1 viene raffigurata la Matrice di mapping dei requisiti: tale matrice è importante per capire se ad ogni requisito è collegato almeno un caso d'uso, in modo tale che tutti i requisiti vengano rispettati.

	RF1	RF2	RF3	RF4	RF5	RF6	RF7	RF8	RF9	RF10	RF11	RF12	RF13	RF14	RF15
InserisciStudente	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Inserisci Libro	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Inserisci Dipendente	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐
Inserisci Prenotazione Libro	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐
Inserisci Prenotazione Posto	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐
Visualizza Studente	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Visualizza Libro	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Visualizza Dipendente	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐
Visualizza Prenotazione	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☑	☐	☐
Rimuovi Studente	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
Rimuovi Libro	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐	☐	☐	☐
Rimuovi Dipendente	☐	☐	☐	☐	☐	☐	☐	☐	☑	☐	☐	☐	☐	☐	☐
Rimuovi Prenotazione	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☑	☑

Figura 1.1: Matrice di mapping tra requisiti e casi d'uso :
RF1-inserimento studente, RF2-visualizzazione studente, RF3-rimozione studente, RF4-inserimento libro, RF5-visualizzazione libro, RF6-rimozione libro, RF7-inserimento dipendente, RF8-visualizzazione dipendente, RF9-rimozione dipendente, RF10-inserimento prenotazione libro, RF11-inserimento prenotazione posto, RF12-visualizzazione prenotazione libro, RF13-visualizzazione prenotazione posto, RF14-rimozione prenotazione libro, RF15-rimozione prenotazione posto.

CAPITOLO 2

Diagramma dei casi d'uso

Dopo aver definito in modo esaustivo i requisiti del sistema, in questo capitolo si procederà a descrivere i casi d'uso, ovvero gli scenari che potrebbero verificarsi durante l'interazione con il software. I casi d'uso costituiscono una metodologia per rappresentare le interazioni tra utenti e sistema attraverso l'impiego di modelli grafici e testi strutturati.

Originariamente introdotti nel metodo Objectory, i casi d'uso sono oggi uno strumento essenziale del linguaggio UML (Unified Modeling Language), un linguaggio standard per la modellazione di sistemi orientati agli oggetti. Nella loro struttura più semplice, un caso d'uso identifica gli attori coinvolti nell'interazione con il sistema e assegna un nome al tipo di interazione. A questa descrizione iniziale possono essere aggiunte ulteriori informazioni che dettagliano l'interazione stessa, utilizzando sia descrizioni testuali sia modelli grafici, come i diagrammi di sequenza o i diagrammi di stato UML. Tali diagrammi forniscono una rappresentazione dinamica e dettagliata del comportamento del sistema in risposta a specifici stimoli provenienti dagli attori esterni.

Il diagramma dei casi d'uso è generalmente il primo strumento grafico utilizzato nel processo di sviluppo software, in particolare durante la fase di analisi dei requisiti. Questo tipo di diagramma è finalizzato a esprimere un comportamento, che può essere sia desiderato sia offerto dal sistema, e si focalizza su due elementi principali:

- *gli attori: ossia, chi o cosa interagisce con il sistema;*
- *i casi d'uso: ossia, quali azioni o funzionalità l'attore può svolgere.*

In sostanza, il diagramma dei casi d'uso si concentra sui diversi "modi" in cui il sistema può essere utilizzato e sulle funzionalità che esso offre agli utenti finali. Grazie alla sua semplicità e alla sua capacità di rappresentare graficamente le interazioni chiave, esso costituisce una base solida per la definizione dei requisiti funzionali e guida lo sviluppo delle fasi successive nel ciclo di vita del software.

2.1 Gestione degli studenti

Il diagramma dei casi d'uso riguardante la gestione degli studenti è mostrato in Figura 2.1. I casi d'uso in questione sono:

- InserisciStudente;
- VisualizzaStudente;
- RicercaStudente;
- RimuoviStudente.

Vediamo più in dettaglio le informazioni relative a ciascun caso d'uso:

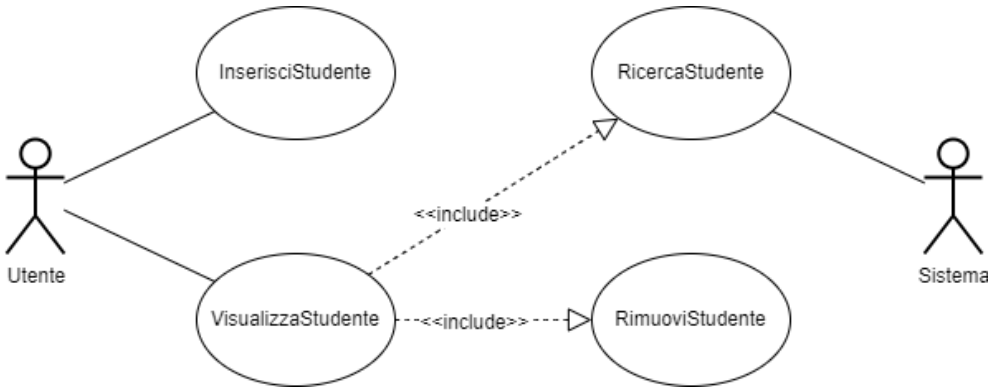

Figura 2.1: Diagramma dei casi d'uso relativo alla gestione degli studenti

► InserisciStudente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: nessuna;
- postcondizioni: lo studente viene inserito nel registro degli studenti;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole inserire un nuovo studente all'interno del registro degli studenti;
  - l'attore inserisce tutti i dati relativi allo studente;
  - il sistema salva il nuovo studente all'interno del registro degli studenti;
- sequenza degli eventi alternativa: nessuna.

► VisualizzaStudente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: lo studente oggetto della visualizzazione deve essere presente nel registro degli studenti;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole controllare i dati di uno studente registrato;
  - include (RicercaStudente);
  - il sistema mostra le informazioni riguardanti lo studente selezionato;
  - l'utente può rimuovere lo studente selezionato se lo desidera;
- sequenza degli eventi alternativa: nessuna.

► RicercaStudente:

- attori primari: sistema;
- attori secondari: nessuno;
- precondizioni: lo studente oggetto della ricerca deve essere presente nel registro degli studenti;

- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole leggere le informazioni di uno studente registrato;
  - il sistema cerca il codice identificativo dello studente selezionato e mostra i dati di quest'ultimo se è presente nel registro;
- sequenza degli eventi alternativa: nessuna.

▶ RimuoviStudente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: lo studente che si vuole rimuovere deve essere presente nel registro degli studenti;
- postcondizioni: lo studente è rimosso dal registro;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole rimuovere uno studente dal registro degli studenti;
  - il sistema visualizza sullo schermo le informazioni dello studente;
  - l'utente avvia la procedura di eliminazione;
  - il sistema rimuove le informazioni dello studente dal registro;
- sequenza degli eventi alternativa: nessuna.

2.2 Gestione della libreria

Il diagramma dei casi d'uso riguardante la gestione della libreria è mostrato in Figura 2.2. I casi d'uso in questione sono:

- InserisciLibro;
- VisualizzaLibro;
- RicercaLibro;
- RimuoviLibro.

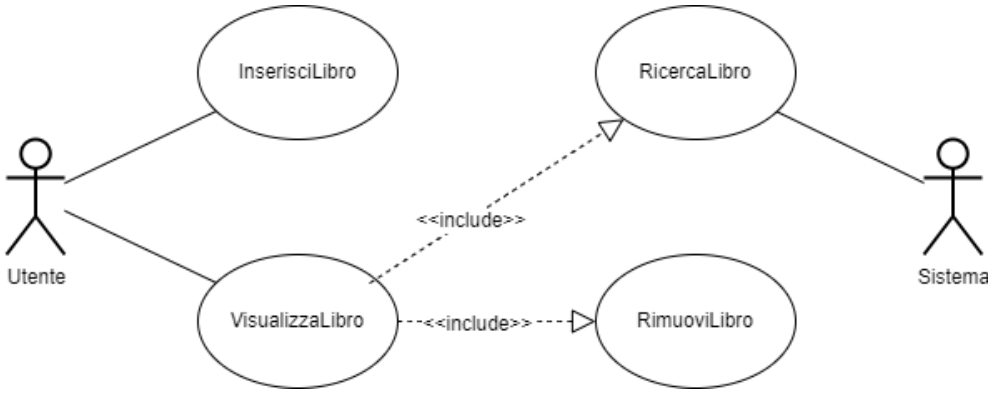

Figura 2.2: Diagramma dei casi d'uso relativo alla gestione della libreria

Più in dettaglio, le informazioni relative a ciascun caso d'uso sono le seguenti:

▶ InserisciLibro:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: nessuna;
- postcondizioni: il libro viene inserito nel registro dei libri disponibili;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole inserire un nuovo libro all'interno del registro dei libri;
  - l'attore inserisce tutti i dati relativi al libro;
  - il sistema salva il nuovo libro all'interno del registro dei libri;
- sequenza degli eventi alternativa: nessuna.

▶ VisualizzaLibro:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: il libro oggetto della visualizzazione deve essere presente nel registro dei libri disponibili;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole controllare la scheda di un libro registrato;
  - include (RicercaLibro);
  - il sistema mostra le informazioni riguardanti il libro selezionato;
  - l'utente può rimuovere il libro selezionato se lo desidera;
- sequenza degli eventi alternativa: nessuna.

▶ RicercaLibro:

- attori primari: sistema;
- attori secondari: nessuno;
- precondizioni: il libro oggetto della ricerca deve essere presente nel registro dei libri disponibili;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole controllare la scheda di un libro registrato;
  - il sistema cerca il codice identificativo del libro selezionato e mostra i dati di quest'ultimo se è presente nel registro;
- sequenza degli eventi alternativa: nessuna.

▶ RimuoviLibro:

- attori primari: utente;
- attori secondari: nessuno;

- precondizioni: il libro che si vuole rimuovere deve essere presente nel registro dei libri;
- postcondizioni: il libro è rimosso dal registro;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole rimuovere un libro dal registro dei libri disponibili;
  - il sistema visualizza sullo schermo la scheda del libro selezionato;
  - l'utente avvia la procedura di eliminazione;
  - il sistema rimuove le informazioni del libro dal registro;
- sequenza degli eventi alternativa: nessuna.

2.3 Gestione dei dipendenti

Il diagramma dei casi d'uso riguardante la gestione dei dipendenti è mostrato in Figura 2.3. I casi d'uso in questione sono:

- InserisciDipendente;
- VisualizzaDipendente;
- RicercaDipendente;
- RimuoviDipendente.

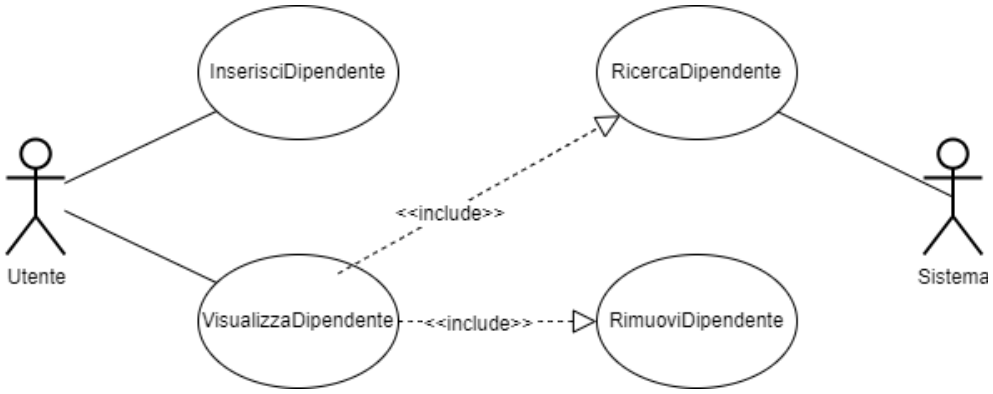

Figura 2.3: Diagramma dei casi d'uso relativo alla gestione dei dipendenti

La descrizione dei singoli casi d'uso è di seguito specificata:

► InserisciDipendente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: nessuna;
- postcondizioni: il dipendente viene inserito nel registro dei dipendenti;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole inserire un nuovo dipendente all'interno del registro dei dipendenti;

  - l'attore inserisce tutti i dati relativi al dipendente;
  - il sistema salva il nuovo dipendente all'interno del registro dei dipendenti;
- sequenza degli eventi alternativa: nessuna.

▶ VisualizzaDipendente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: il dipendente oggetto della visualizzazione deve essere presente nel registro dei dipendenti;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole controllare i dati di un dipendente registrato;
  - include (RicercaDipendente);
  - il sistema mostra le informazioni riguardanti il dipendente selezionato;
  - l'utente può rimuovere il dipendente selezionato se lo desidera;
- sequenza degli eventi alternativa: nessuna.

▶ RicercaDipendente:

- attori primari: sistema;
- attori secondari: nessuno;
- precondizioni: il dipendente oggetto della ricerca deve essere presente nel registro dei dipendenti;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole leggere le informazioni di un dipendente registrato;
  - il sistema cerca il codice identificativo del dipendente selezionato e mostra i dati di quest'ultimo se è presente nel registro;
- sequenza degli eventi alternativa: nessuna.

▶ RimuoviDipendente:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: il dipendente che si vuole rimuovere deve essere presente nel registro dei dipendenti;
- postcondizioni: il dipendente è rimosso dal registro;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole rimuovere un dipendente dal registro dei dipendenti;
  - il sistema visualizza sullo schermo le informazioni del dipendente;
  - l'utente avvia la procedura di eliminazione;
  - il sistema rimuove le informazioni del dipendente dal registro;
- sequenza degli eventi alternativa: nessuna.

2.4 Gestione delle prenotazioni

Il software gestisce due tipi di prenotazioni che possono essere effettuate dall'utente:

- il prestito di un libro;
- la prenotazione di un posto in aula studio.

2.4.1 Prestito di un libro

Il diagramma dei casi d'uso riguardante il prestito di un libro è mostrato in Figura 2.4. I casi d'uso in questione sono:

- InserisciPrestitoLibro;
- VisualizzaPrestitoLibro;
- RicercaPrestitoLibro;
- RimuoviPrestitoLibro.

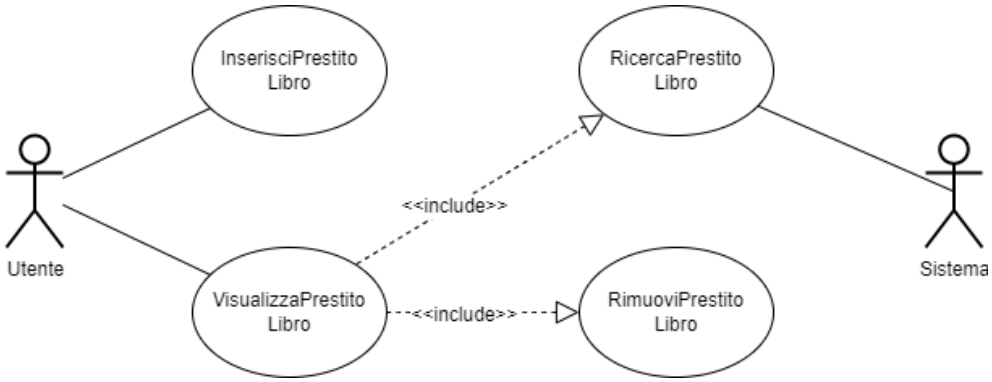

Figura 2.4: Diagramma dei casi d'uso relativo al prestito di un libro

▶ InserisciPrestitoLibro:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: lo studente che vuole prenotare un libro deve essere registrato e il libro in questione deve essere disponibile;
- postcondizioni: il libro non è più disponibile fino al termine del prestito;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando uno studente vuole prendere in prestito un libro dalla biblioteca;
  - l'utente controlla se lo studente sia registrato e se il libro selezionato sia disponibile;
  - il sistema rende tale libro non disponibile per un altro prestito;
- sequenza degli eventi alternativa: nessuna.

▶ VisualizzaPrestitoLibro:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: uno o più studenti devono aver preso in prestito uno o più libri;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole constatare quali supporti dei lettura sono stati presi in prestito;
  - include (RicercaPrestitoLibro);
  - il sistema mostra quali prestiti sono in corso;
  - l'utente può disdire il prestito selezionato se lo desidera;
- sequenza degli eventi alternativa: nessuna.

▶ RicercaPrestitoLibro:

- attori primari: sistema;
- attori secondari: nessuno;
- precondizioni: uno o più studenti devono aver preso in prestito uno o più libri;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole constatare quali libri sono stati presi in prestito;
  - il sistema cerca il codice identificativo del prestito selezionato e mostra i dati di quest'ultimo se è effettivamente in corso;
- sequenza degli eventi alternativa: nessuna.

▶ RimuoviPrestitoLibro:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: il prestito da disdire deve essere effettivamente in corso ;
- postcondizioni: il libro precedentemente in prestito torna disponibile;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole disdire il prestito dei un libro da parte di uno studente;
  - il sistema visualizza sullo schermo i dati del prestito selezionato;
  - l'utente avvia la procedura di disdetta;
  - il sistema rende il libro nuovamente disponibile;
- sequenza degli eventi alternativa: nessuna.

2.4.2 Prenotazione di un posto in biblioteca

Il diagramma dei casi d'uso riguardante la prenotazione di un posto in biblioteca è mostrato in Figura 2.5. I casi d'uso in questione sono:

- InserisciPrenotazionePosto;
- VisualizzaPrenotazionePosto;

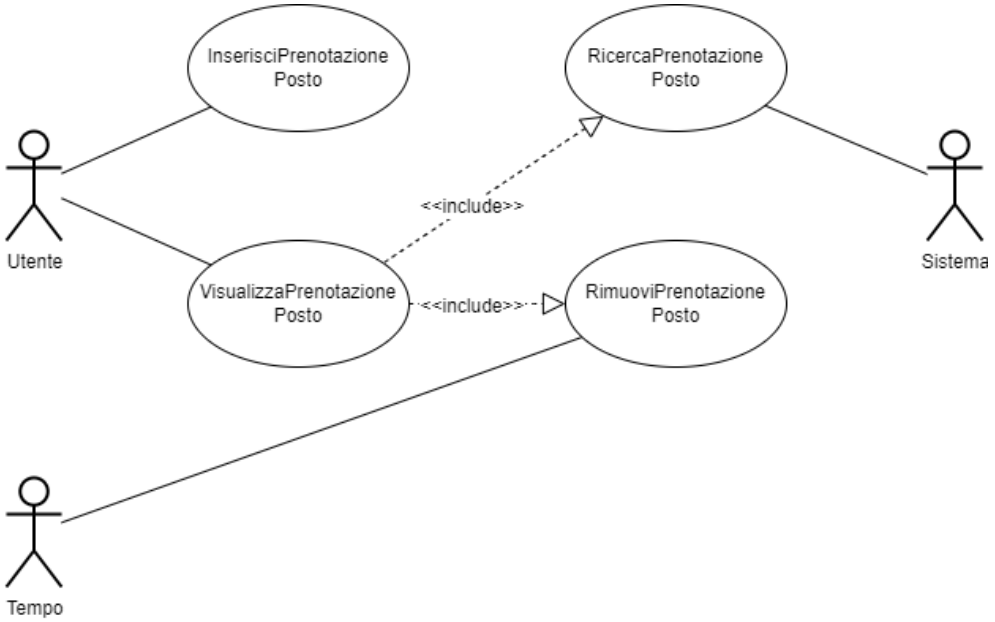

Figura 2.5: Diagramma dei casi d'uso relativo alla prenotazione di un posto in biblioteca

- RicercaPrenotazionePosto;
- RimuoviPrenotazionePosto.

La descrizione dei singoli casi d'uso è di seguito specificata:

► InserisciPrenotazionePosto:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: lo studente che vuole prenotare un posto in aula studio deve essere registrato ed il posto in questione deve essere disponibile;
- postcondizioni: il posto non è più disponibile fino al termine della prenotazione;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando uno studente vuole prenotare un posto in biblioteca;
  - l'utente controlla se lo studente è registrato e se il posto in questioone è disponibile;
  - il sistema rende tale posto non disponibile per un altra prenotazione;
- sequenza degli eventi alternativa: nessuna.

► VisualizzaPrenotazionePosto:

- attori primari: utente;
- attori secondari: nessuno;
- precondizioni: uno o più studenti devono aver prenotato uno o più posti in aula studio;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole constatare quali posti sono stati prenotati;
  - include (RicercaPrenotazionePosto);
  - il sistema mostra quali prenotazioni sono in corso;
  - l'utente può disdire la prenotazione selezionata se lo desidera;
- sequenza degli eventi alternativa: nessuna.

► RicercaPrenotazionePosto:

- attori primari: sistema;
- attori secondari: nessuno;
- precondizioni: uno o più studenti devono aver prenotato uno o più posti in biblioteca;
- postcondizioni: nessuna;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole constatare quali posti sono stati prenotati;
  - il sistema cerca il codice identificativo della prenotazione selezionata e mostra i dati di quest'ultima se è effettivamente in corso;
- sequenza degli eventi alternativa: nessuna.

► RimuoviPrenotazionePosto:

- attori primari: utente;

- attori secondari: tempo;
- precondizioni: la prenotazione da disdire deve essere effettivamente in corso;
- postcondizioni: il posto precedentemente prenotato torna disponibile;
- sequenza degli eventi principale:
  - il caso d'uso inizia quando l'utente vuole disdire la prenotazione di un posto in biblioteca da parte di uno studente;
  - il sistema visualizza sullo schermo i dati della prenotazione selezionata;
  - l'utente avvia la procedura di disdetta;
  - il sistema rende il posto nuovamente disponibile;
- sequenza degli eventi alternativa: nessuna:
  - trascorse tre ore la prenotazione giunge al termine;
  - lo studente lascia il posto a lui riservato e il sistema lo rende nuovamente disponibile.

CAPITOLO 3

Diagramma delle classi

Dopo aver identificato e analizzato i casi d'uso, abbiamo estratto da essi le classi fondamentali che costituiranno il sistema. Successivamente abbiamo realizzato i diagrammi che rappresentano i comportamenti principali del software. Nell'ambito dell'ingegneria del software, i diagrammi delle classi offrono una rappresentazione statica del sistema, fornendo una visione d'insieme delle entità e delle relazioni che esistono tra esse. Questi diagrammi definiscono i tipi di oggetti presenti nel sistema e le relazioni che li legano, offrendo una visione di alto livello dell'applicazione. La modellazione mediante diagrammi delle classi può essere eseguita utilizzando quasi tutti i metodi orientati agli oggetti, il che la rende uno strumento versatile e fondamentale nel processo di progettazione.

La prima fase dello sviluppo di un modello software consiste generalmente nell'osservazione del mondo reale e nell'identificazione degli oggetti essenziali, che vengono successivamente rappresentati come classi. Queste ultime sono rappresentate graficamente come rettangoli contenenti il nome della classe stessa, i suoi attributi e le operazioni. Per descrivere un'associazione tra due o più classi, si utilizza una linea che collega i rettangoli corrispondenti, indicando la relazione esistente tra le entità.

I diagrammi delle classi sono simili ai modelli semantici dei dati, ampiamente utilizzati nella progettazione di database. Questi modelli, infatti, rappresentano le entità dei dati, gli attributi associati a ciascuna entità, e le relazioni tra di esse. La loro struttura permette di descrivere, in maniera chiara e precisa, la logica organizzativa dei dati di un sistema, facilitando così la comprensione e la progettazione del software. I diagrammi delle classi, oltre a fornire una rappresentazione visiva delle componenti chiave del sistema, svolgono un ruolo centrale nella definizione e comprensione delle relazioni tra le classi e nella specifica delle operazioni che possono essere eseguite su di esse.

Per la realizzazione del software, è stato adottato il pattern architetturale MVC (Model-View-Controller), la cui struttura sarà approfondita nel capitolo successivo.

3.1 Home

Il modulo Home (Figura 1.1) rappresenta la schermata principale del sistema e funge da punto di accesso per le diverse funzionalità del software. In questo modulo vengono gestite le interfacce utente e le operazioni iniziali, offrendo un'interazione fluida e intuitiva. Esso costituisce l'elemento chiave per l'avvio dell'esperienza dell'utente, permettendo di accedere rapidamente ai dati di interesse, siano essi relativi a studenti, libri, prenotazioni o informazioni sui membri del personale.

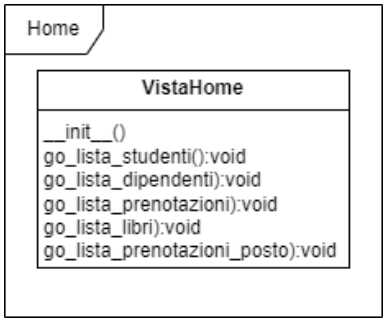

Figura 3.1: Diagramma delle classi relativo alla home page

3.2 Studenti

Il modulo Studenti (Figura 1.2) si occupa della gestione degli studenti registrati e della memorizzazione delle loro informazioni. Al suo interno troviamo i model Studente e Lista Studenti, i controller Controllore Studente e Controllore Lista Studenti e le view Vista Studente, Vista Lista Studenti e Vista Inserisci Studente.

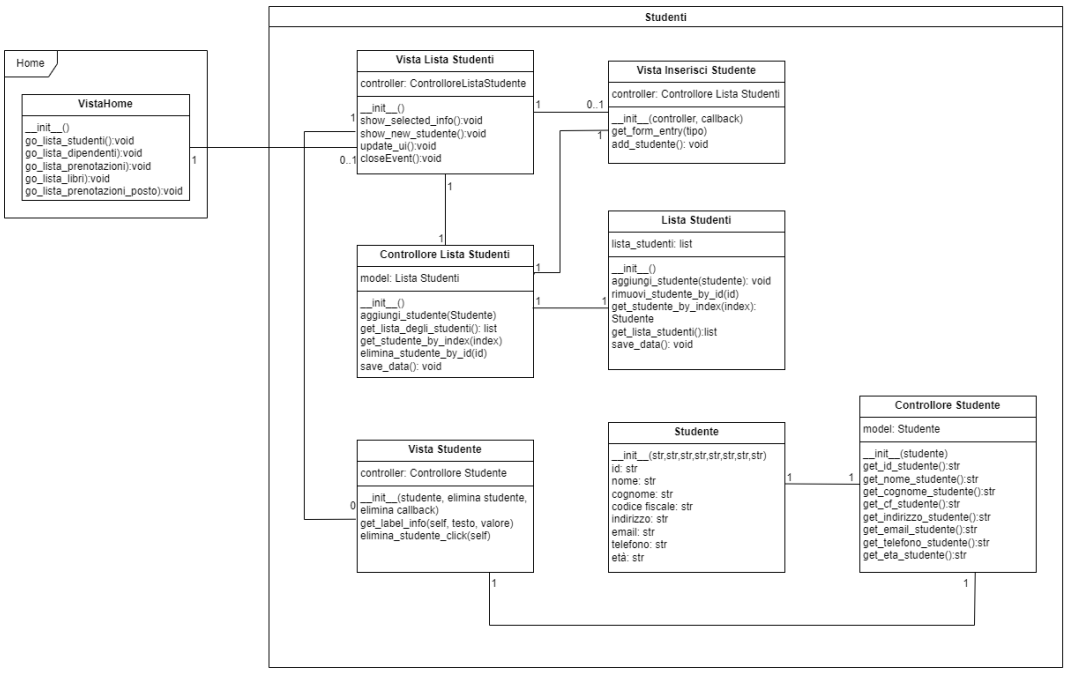

Figura 3.2: Diagramma delle classi relativo al modulo di gestione degli studenti

3.3 Dipendenti

Il modulo Dipendenti (Figura 1.3) è dedicato alla gestione del personale della biblioteca e delle informazioni relative ai dipendenti registrati. Al suo interno troviamo i model Dipendente e Lista Dipendenti, i controller Controllore Dipendente e Controllore Lista Dipendenti e le view Vista Dipendente, Vista Lista Dipendenti e Vista Inserisci Dipendente.

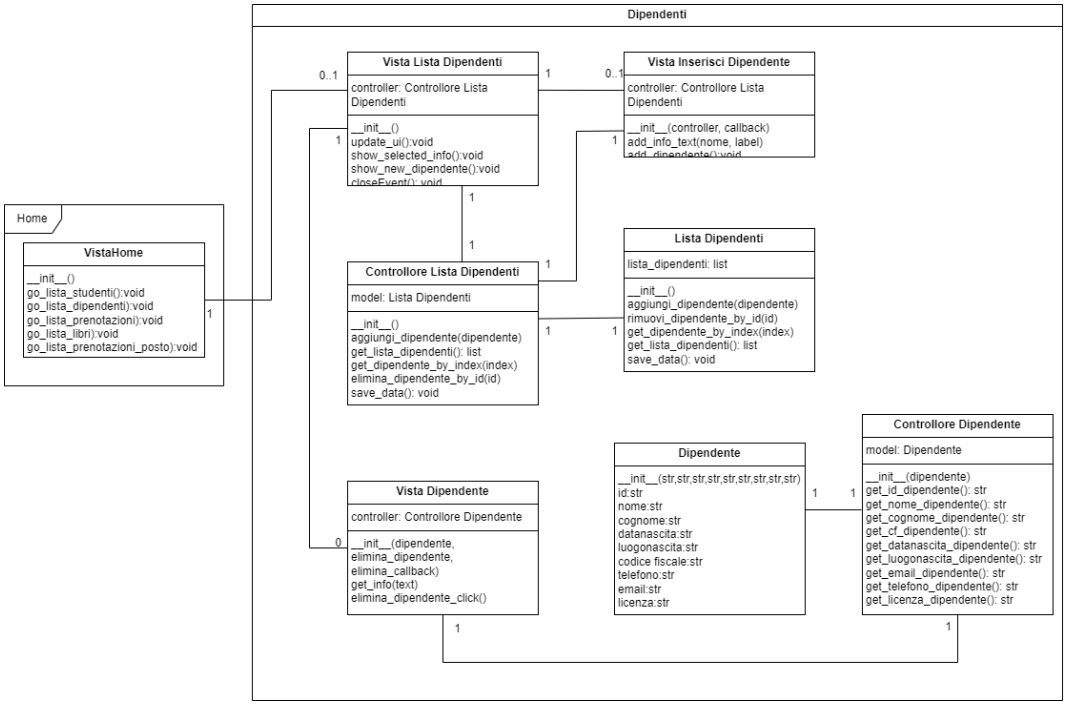

Figura 3.3: Diagramma delle classi relativo al modulo di gestione dei dipendenti

3.4 Libreria

Il modulo Libreria (Figura 1.4) rappresenta la collezione di libri disponibili nel sistema e le relative operazioni di gestione. Esso contiene informazioni dettagliate su ogni libro, tra cui titolo, genere e disponibilità. Al suo interno troviamo i model Libro e Lista Libri, i controller Controllore Libro e Controllore Lista Libro e le view Vista Libro, Vista Lista Libri e Vista Inserisci Libro.

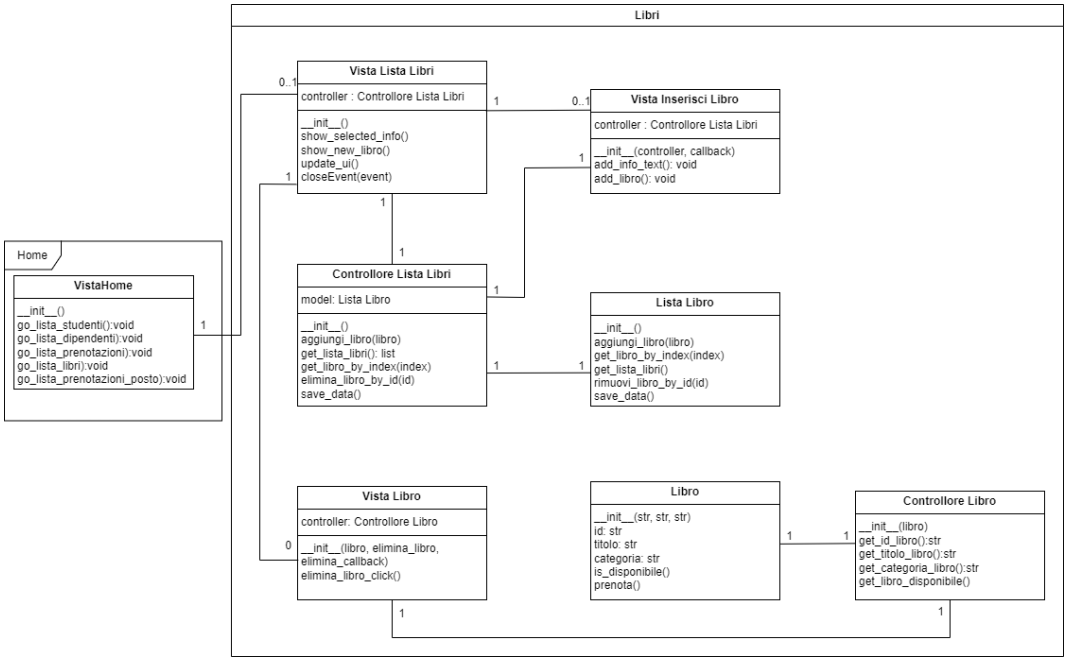

Figura 3.4: Diagramma delle classi relativo al modulo di gestione della libreria

3.5 Prenotazione dei posti

Il modulo Prenotazione dei Posti (Figura 1.5) è progettato per la gestione delle postazioni di studio all'interno della biblioteca. Gli studenti possono verificare la disponibilità dei posti e prenotarli per una determinata fascia oraria. Al suo interno troviamo i model Prenotazione Posto e Lista Prenotazione Posto, i controller Controllore Prenotazioni Posto e Controllore Lista Prenotazioni Posto e le view Vista Prenotazione Posto, Vista Lista Prenotazioni Posto e Vista Inserisci Prenotazioni Posto.

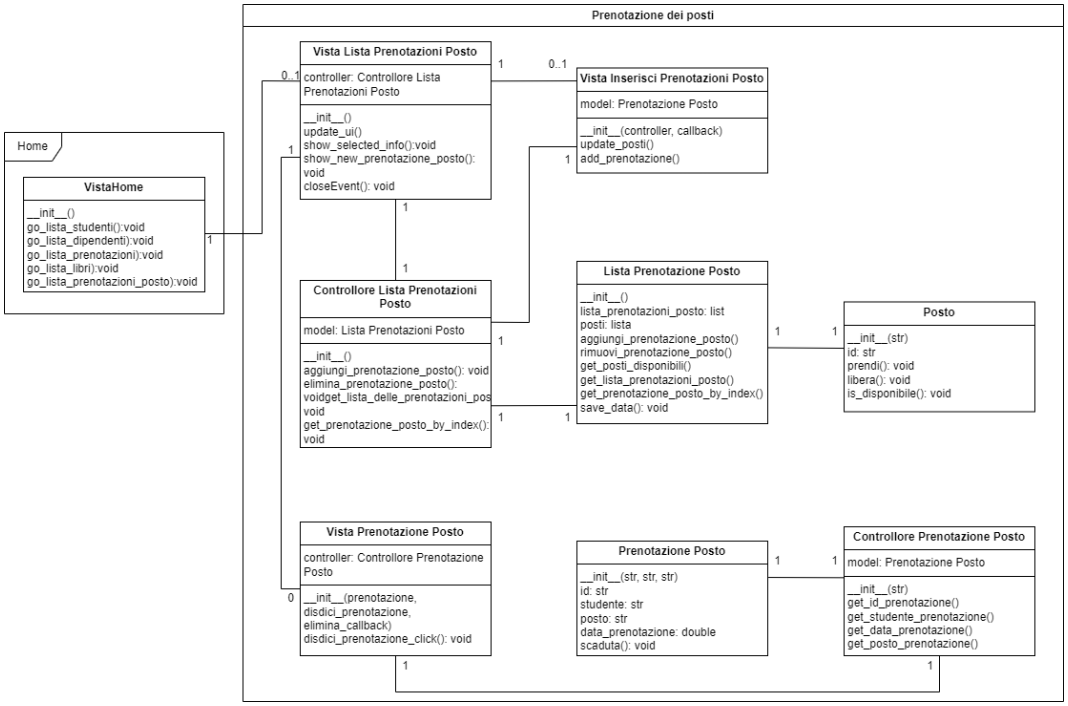

Figura 3.5: Diagramma delle classi relativo al modulo di prenotazione dei posti

3.6 Prenotazione dei libri

Il modulo Prenotazione dei Libri (Figura 1.6)gestisce il processo di prenotazione e di restituzione di un libro da parte degli utenti. Al suo interno troviamo i model Prenotazione e Lista Prenotazioni, i controller Controllore Prenotazione e Controllore Lista Prenotazioni e le view Vista Prenotazione, Vista Lista Prenotazioni e Vista Inserisci Prenotazione.

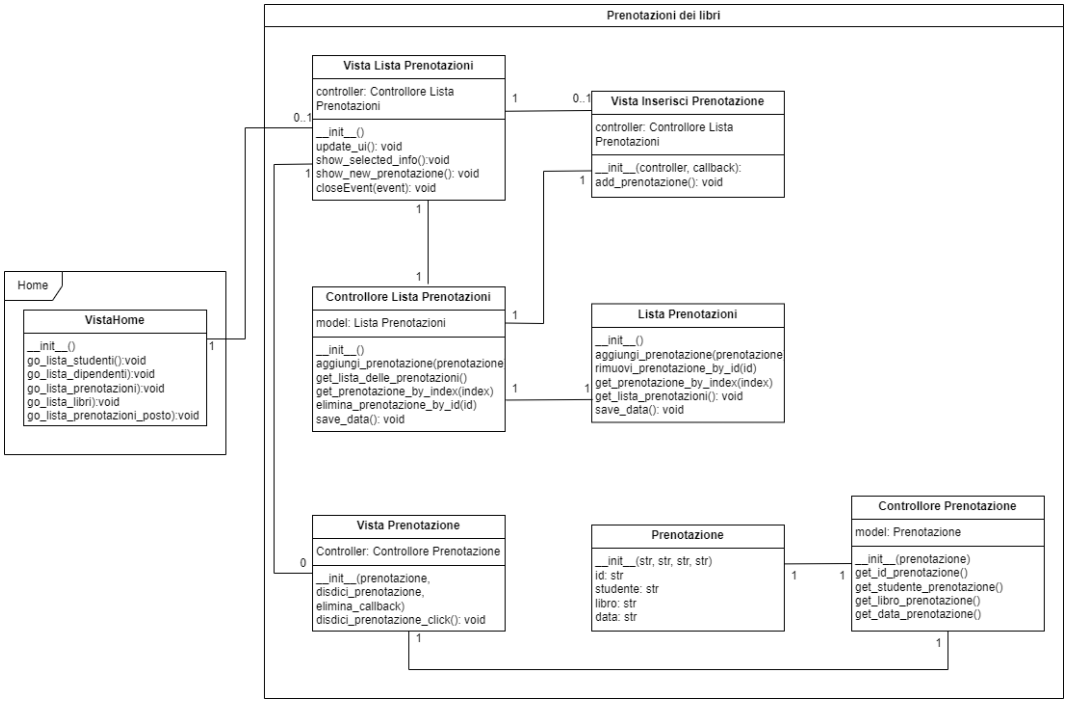

Figura 3.6: Diagramma delle classi relativo al modulo di prenotazione dei libri

3.7 Diagramma delle classi complessivo

Il diagramma complessivo delle classi fornisce una rappresentazione unificata di tutte le classi principali del sistema e delle relazioni esistenti tra esse. Questo diagramma offre una visione globale dell'architettura del software, evidenziando le interazioni tra le varie classi. Esso è mostrato in figura 1.7.

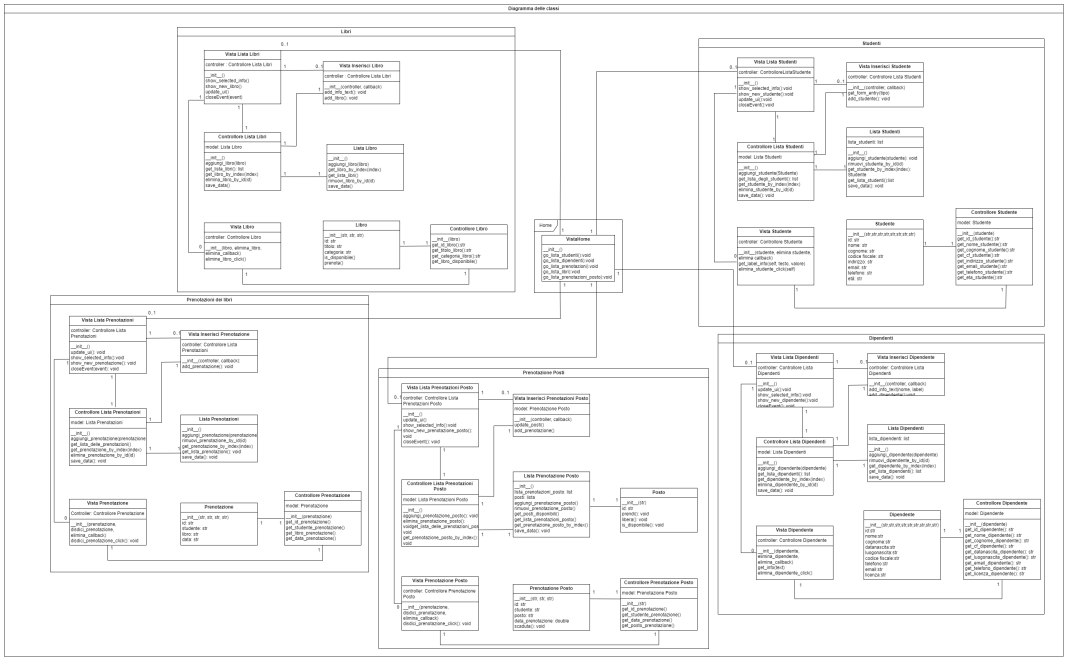

Figura 3.7: Diagramma delle classi complessivo

CAPITOLO 4

Progettazione della componente applicativa

In questo capitolo si illustreranno i modelli architetturali utilizzati nel progetto, soffermandosi in particolare sull'architettura MVC, sui diagrammi di sequenza e sui diagrammi di attività, al fine di fornire una visione completa dell'organizzazione e del flusso di lavoro del software.

4.1 Progettazione dell'architettura

Per la realizzazione del software, è stato adottato il pattern architetturale MVC (Model-View-Controller), una scelta strategica che si basa sulla suddivisione del sistema in tre componenti logiche ben distinte: model, view e controller. Il motivo principale di questa decisione risiede nella capacità del pattern di favorire la modularità e la manutenibilità del codice, consentendo di apportare modifiche o di aggiungere nuove funzionalità in una parte del sistema senza influenzare le altre. Questa separazione delle responsabilità facilita lo sviluppo e l'evoluzione del software, poiché ogni componente può essere modificato indipendentemente dalle altre, rendendo il sistema flessibile e adattabile a cambiamenti futuri. Un pattern è una soluzione progettuale collaudata per problemi ricorrenti nello sviluppo del software. In altre parole, rappresenta una linea guida che offre una struttura organizzata e riutilizzabile per affrontare determinate sfide di progettazione, garantendo efficienza e chiarezza nella costruzione del sistema. Il pattern MVC, in particolare, si distingue per la sua capacità di separare le operazioni effettuate software, distribuendo le responsabilità tra tre elementi fondamentali:

- *Model*: il model rappresenta il cuore dei dati e della logica di business del sistema. In questa componente vengono gestiti e manipolati i dati dell'applicazione, comprese le operazioni di accesso, aggiornamento e gestione delle informazioni persistenti, spesso interfacciandosi con un database. Il model è indipendente dalla presentazione e può essere riutilizzato in diverse interfacce utente, poiché si occupa esclusivamente della gestione e del trattamento dei dati.

- *View*: la view si occupa della presentazione visiva delle informazioni agli utenti. Essa riceve i dati dal model e li trasforma in un formato comprensibile e facilmente interpretabile. La view non contiene logica di business, ma si limita a mostrare l'interfaccia grafica e a fornire agli utenti un punto di interazione con il sistema. Il grande vantaggio di questa separazione è la possibilità di modificare o sostituire la view senza influire sui

dati o sulle logiche sottostanti, garantendo, così, una maggiore flessibilità nella gestione dell'interfaccia utente.

- *Controller*: il controller agisce come intermediario tra il model e la view. È responsabile della gestione delle richieste degli utenti e della loro interpretazione, decidendo quali operazioni eseguire sui dati e quale vista restituire. Il controller riceve gli input dall'utente, li elabora e invia i risultati al Model, per l'aggiornamento dei dati, o alla view, per la presentazione. Questa componente assicura che la logica di business e la presentazione rimangano disaccoppiate, gestendo il flusso complessivo dell'applicazione.

La scelta di utilizzare l'architettura MVC si è rivelata efficace per garantire una struttura ordinata e modulare, dove le tre componenti possono evolvere autonomamente. Ciò significa, ad esempio, che è possibile aggiungere o modificare una nuova view per migliorare l'interfaccia utente senza dover intervenire sui dati gestiti dal model o sulle logiche di interazione gestite dal controller, incrementando, così, la scalabilità e la manutenibilità del sistema nel tempo.

4.2 Diagrammi di sequenza

I diagrammi di sequenza, nel linguaggio UML, sono strumenti fondamentali per modellare le interazioni tra attori e oggetti di un sistema, nonché per rappresentare le interazioni interne tra gli oggetti stessi. Grazie alla ricca sintassi di UML, è possibile rappresentare diversi tipi di interazione, catturando la dinamica di un sistema in relazione a specifici casi d'uso. Come suggerisce il nome, un diagramma di sequenza visualizza la sequenza temporale delle interazioni durante l'esecuzione di un caso d'uso o di una sua istanza. Gli oggetti e gli attori coinvolti nel processo sono disposti orizzontalmente nella parte superiore del diagramma, ciascuno con una linea tratteggiata verticale che rappresenta la loro "linea di vita", ovvero il periodo in cui l'istanza dell'oggetto partecipa al flusso di esecuzione. Le interazioni tra oggetti sono indicate da frecce annotate, le quali rappresentano chiamate, parametri e valori di ritorno, esplicitando la natura dello scambio tra gli elementi. La sequenza degli eventi è ordinata dall'alto verso il basso, con l'avanzare della lettura che corrisponde allo sviluppo cronologico delle operazioni. Un rettangolo, disegnato sulla linea di vita di un oggetto, denota l'intervallo in cui l'oggetto è attivamente coinvolto in un'operazione. Questa struttura consente di rappresentare in modo chiaro e intuitivo il flusso delle interazioni, facilitando la comprensione del comportamento del sistema per attori e sviluppatori e supportando la validazione delle specifiche dei requisiti funzionali attraverso una rappresentazione visiva delle chiamate tra oggetti.

Nelle Figure 1.1 - 1.5 vengono riportati i diagrammi di sequenza per la gestione degli studenti, dei dipendenti, della libreria, dei libri e dei posti.

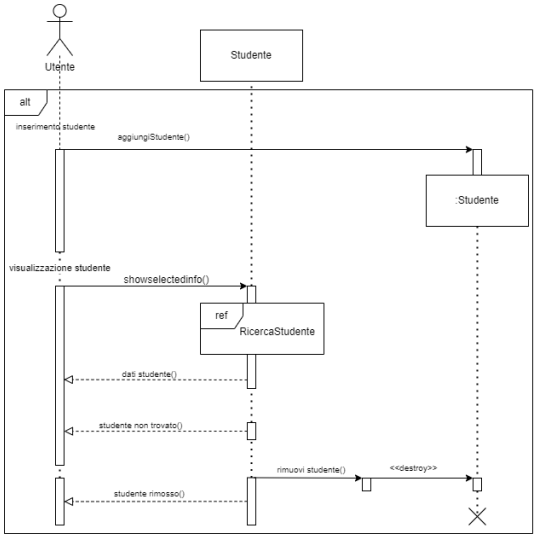

Figura 4.1: Diagramma di sequenza relativo alla gestione degli studenti

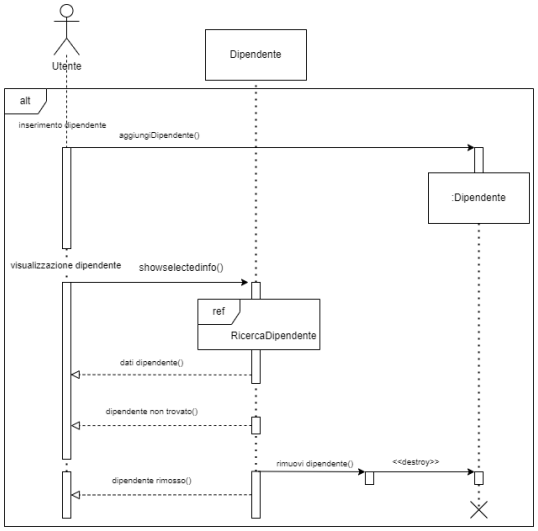

Figura 4.2: Diagramma di sequenza relativo alla gestione dei dipendenti

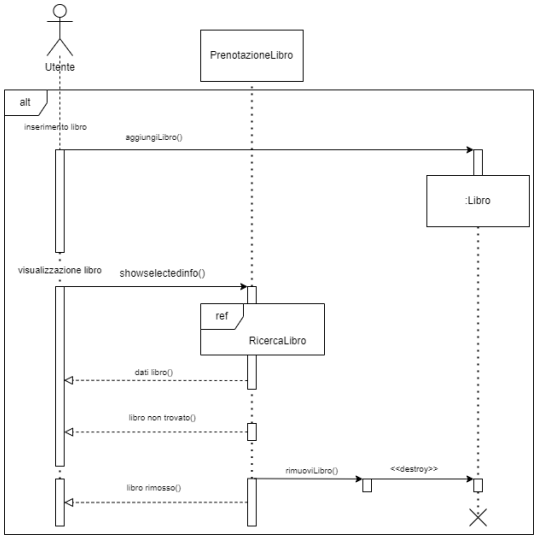

Figura 4.3: Diagramma di sequenza relativo alla gestione della libreria

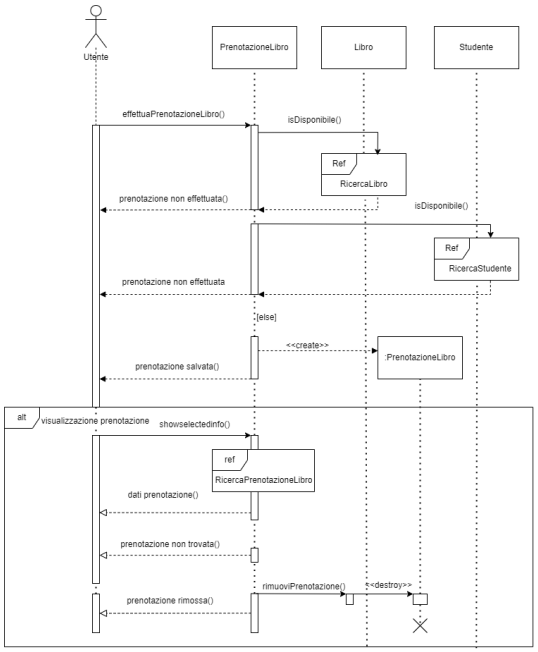

Figura 4.4: Diagramma di sequenza relativo alla prenotazione di un libro

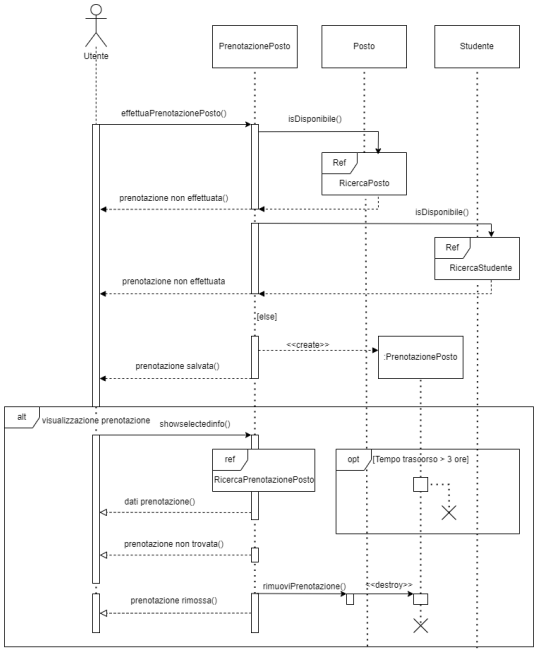

Figura 4.5: Diagramma di sequenza relativo alla prenotazione di un posto

4.3 Diagrammi di attività

Il diagramma di attività è un tipo di diagramma UML particolarmente utile per descrivere processi attraverso rappresentazioni grafiche in cui i nodi corrispondono alle attività e gli archi definiscono l'ordine di esecuzione. Ogni attività è rappresentata graficamente con un rettangolo dai bordi smussati, all'interno del quale è riportata una descrizione dell'operazione da svolgere. Il flusso di esecuzione viene indicato tramite frecce orientate, che definiscono la sequenza temporale delle attività. All'interno del diagramma, l'inizio del processo è rappresentato con un simbolo specifico, così come il termine, garantendo chiarezza nella lettura dell'intero flusso. È inoltre possibile rappresentare attività parallele, le quali vengono indicate tramite un segmento da cui si diramano frecce divergenti per ciascuna attività che può essere eseguita in simultanea. Quando il processo richiede delle decisioni, ossia quando l'esecuzione di un'attività dipende da una scelta condizionale, il punto decisionale viene rappresentato con un rombo, dal quale partono i flussi alternativi corrispondenti alle possibili opzioni. Il ricongiungimento dei flussi, al termine di percorsi alternativi o paralleli, è rappresentato graficamente da un segmento in cui le frecce convergono, completando così la descrizione del processo. Questo tipo di diagramma risulta particolarmente efficace per rappresentare in modo dettagliato e intuitivo i processi complessi, offrendo una visione d'insieme chiara e immediata sulla sequenza e sulle dipendenze tra le attività.

Nelle Figure 1.6 - 1.14 vengono presentati i diagrammi di attività relativi al nostro sistema.

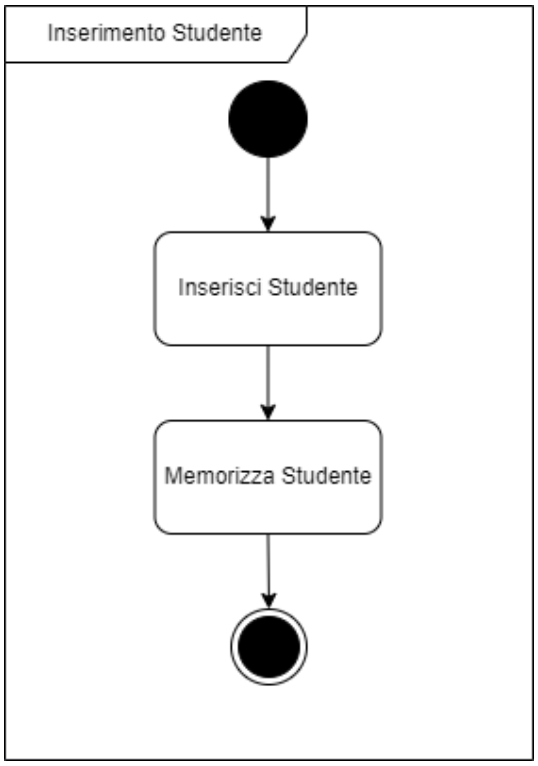

Figura 4.6: Diagramma di attività relativo all'inserimento di uno studente

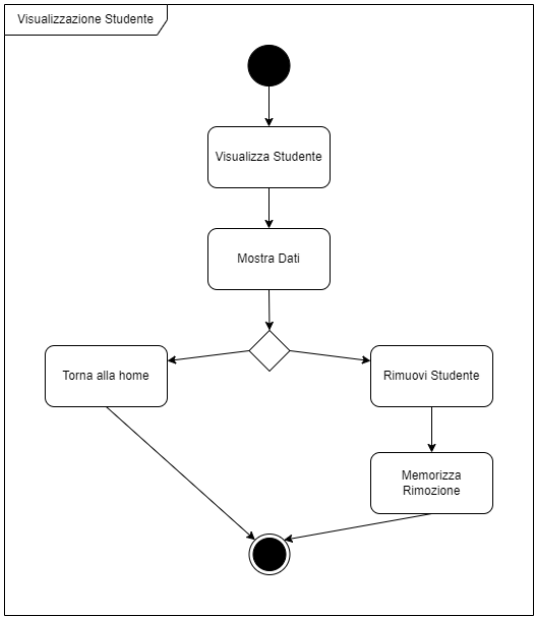

Figura 4.7: Diagramma di attività relativo alla visualizzazione dei dati di uno studente

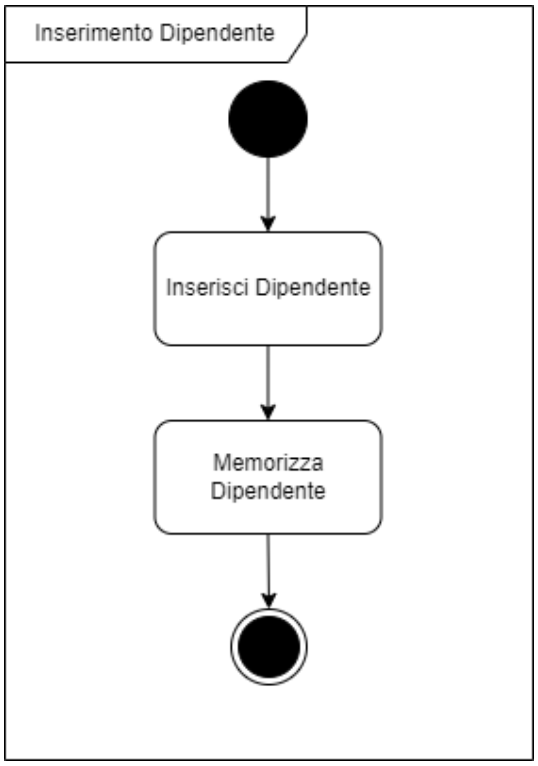

Figura 4.8: Diagramma di attività relativo all'inserimento di un dipendente

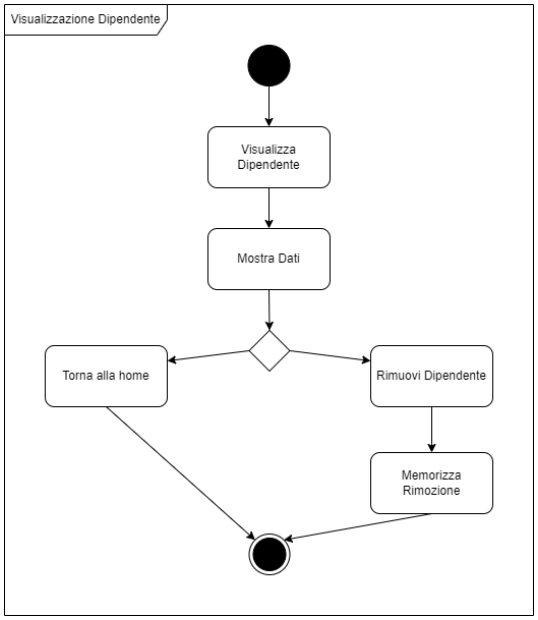

Figura 4.9: Diagramma di attività relativo alla visualizzazione dei dati di un dipendente

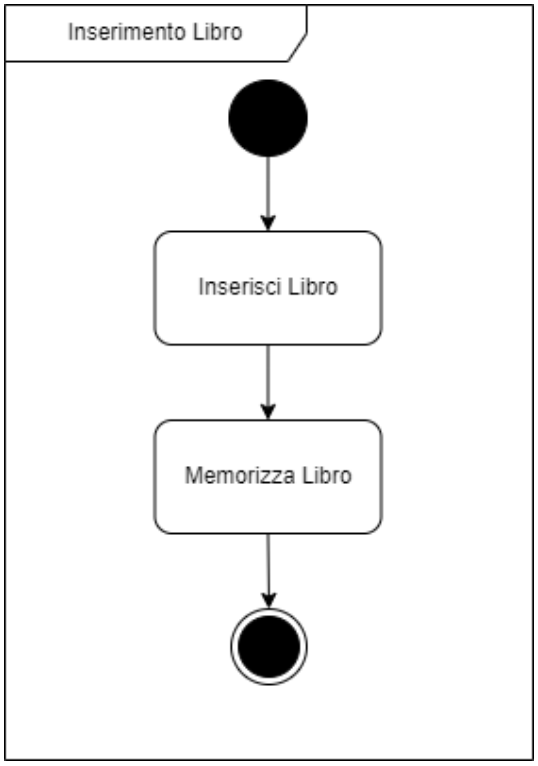

Figura 4.10: Diagramma di attività relativo all' inserimento di un libro

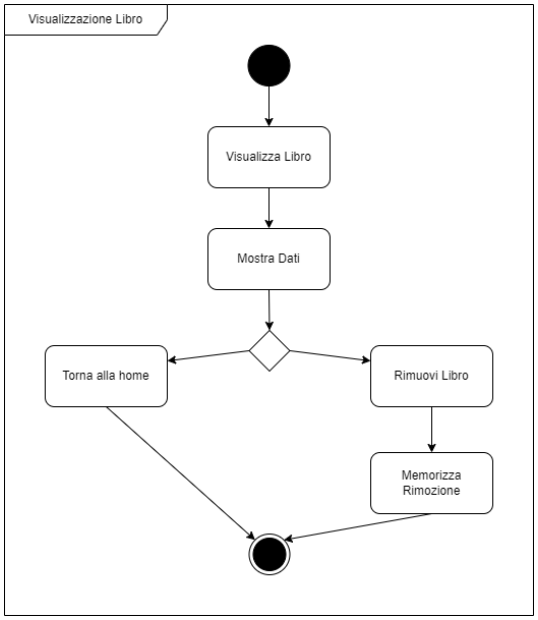

Figura 4.11: Diagramma di attività relativo alla visualizzazione dei dati di un libro

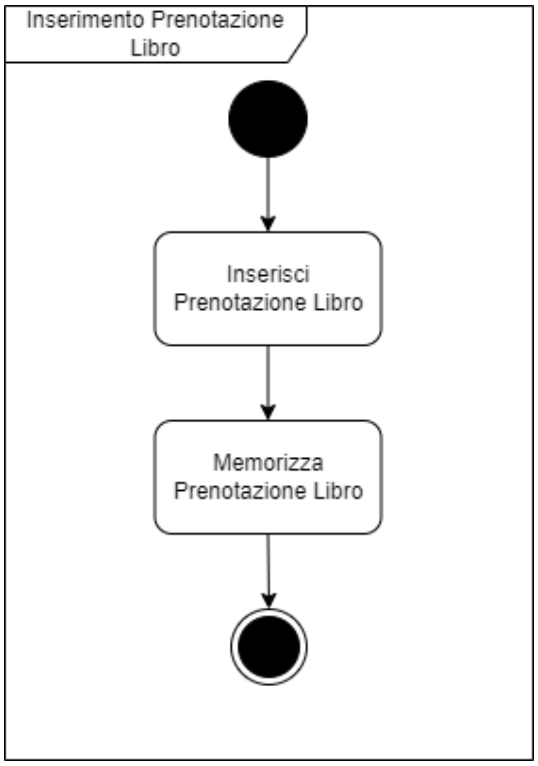

Figura 4.12: Diagramma di attività relativo alla prenotazione di un libro

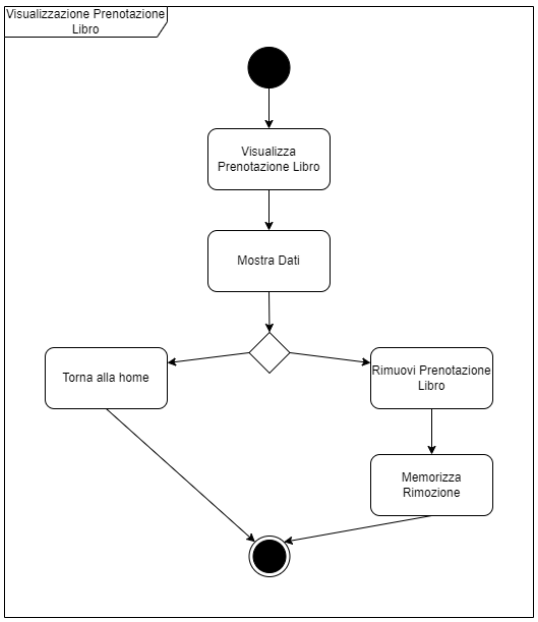

Figura 4.13: Diagramma di attività relativo alla visualizzazione della prenotazione di un libro

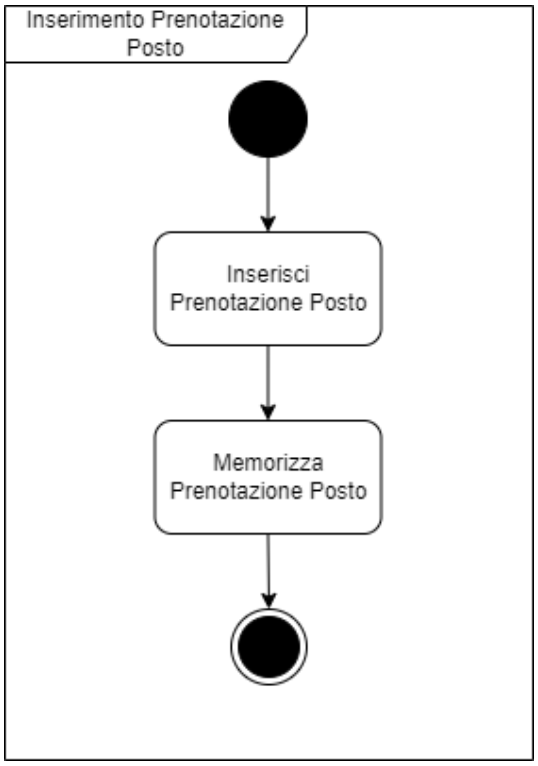

Figura 4.14: Diagramma di attività relativo alla prenotazione di un posto

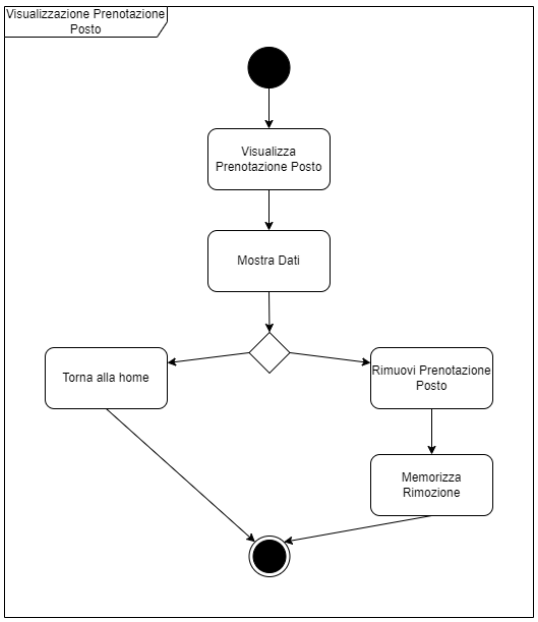

Figura 4.15: Diagramma di attività relativo alla visualizzazione della prenotazione di un posto

CAPITOLO 5

Implementazione

Nell'ingegneria del software, la fase di implementazione rappresenta quel momento fondamentale del processo in cui viene sviluppato un sistema software eseguibile, seguendo le specifiche progettuali stabilite. È importante notare come le attività di progettazione e implementazione risultino strettamente connesse e interdipendenti, poiché le decisioni prese in fase di progettazione influenzano profondamente la qualità e la manutenibilità del codice prodotto, mentre il feedback derivato dall'implementazione può condurre a revisioni e affinamenti delle scelte progettuali iniziali. In questo capitolo, verrà illustrato in dettaglio come la fase di progettazione sia stata affrontata e sviluppata, specificamente nel contesto del software oggetto della presente tesi, evidenziando le scelte tecniche e metodologiche adottate per garantire l'efficacia e la modularità del sistema.

5.1 Codice relativo alla gestione degli studenti

Questa sezione descrive le parti del codice dedicate alla gestione degli studenti iscritti alla biblioteca. Si illustreranno le principali funzioni per l'aggiunta, la cancellazione e la ricerca degli studenti nel sistema, oltre ai dettagli sui metodi utilizzati per la validazione dei dati e l'interazione con il database per garantire un'efficiente gestione degli utenti.

La classe `ListaStudenti`(Listato 1.1) rappresenta il "model" della lista degli studenti. Il costruttore carica la lista di studenti da un file pickle se esiste già, altrimenti crea una lista vuota, dopodiché i quattro metodi della classe gestiscono le operazioni di aggiunta, rimozione e ricerca degli studenti e il salvataggio della lista degli studenti dopo ogni operazione.

```
class ListaStudenti():
def __init__(self):
   super(ListaStudenti, self).__init__()
   self.lista_studenti = []
   if
   os.path.isfile('listastudenti/data/lista_studenti_salvata.pickle'):
      with open
      ('listastudenti/data/lista_studenti_salvata.pickle', 'rb')

      as f:
         self.lista_studenti = pickle.load(f)

def aggiungi_studente(self, studente):
   self.lista_studenti.append(studente)

def rimuovi_studente_by_id(self, id):
   def is_selected_studente(studente):
      if studente.id == id:
         return True
      return False
   self.lista_studenti.remove(list(filter(is_selected_studente,

   self.lista_studenti))[0])
```

```
def get_studente_by_index(self, index):
   return self.lista_studenti[index]

def get_lista_studenti(self):
   return self.lista_studenti

def save_data(self):
   with open
   ('listastudenti/data/lista_studenti_salvata.pickle', 'wb')

   as handle:
      pickle.dump(self.lista_studenti, handle,

      pickle.HIGHEST_PROTOCOL)
```

Listato 5.1: Classe `ListaStudenti`

La classe `ControlloreListaStudenti` rappresenta il controller che fa da intermediario tra la classe `ListaStudenti` e le classi `VistaListaStudenti` e `VistaInserisciStudente`, che implementano l'interfaccia grafica del software. Al suo interno vengono definite cinque funzioni che richiamano i rispettivi metodi nel modello `ListaStudenti`.

La classe `VistaListaStudenti` rappresenta la vista principale per quanto riguarda la lista degli studenti. I metodi al suo interno sono i seguenti:

- `show-selected-info(self)`, che apre una finestra con i dati dello studente selezionato;
- `show-new-studente(self)`, che apre una finestra per inserire un nuovo studente;
- `update-ui(self)`, che aggiorna la vista con i dati attuali della lista degli studenti;
- `closeEvent(self, event)`, che salva i dati alla chiusura della finestra.

La classe `VistaInserisciStudente`, mostrata nel Listato 1.2, implementa l'operazione di aggiunta di un nuovo studente. Al suo interno sono presenti due metodi.

```
   def get_form_entry(self, tipo):
   self.v_layout.addWidget(QLabel(tipo))
   current_text_edit = QLineEdit(self)
   self.v_layout.addWidget(current_text_edit)
   self.info[tipo] = current_text_edit

def add_studente(self):
   nome = self.info["Nome"].text()
   cognome = self.info["Cognome"].text()
   cf = self.info["Codice Fiscale"].text()
   indirizzo = self.info["Indirizzo"].text()
   email = self.info["Email"].text()
   telefono = self.info["Telefono"].text()
   eta = self.info["Età"].text()
   if nome == "" or cognome == "" or cf == "" or indirizzo ==
   "" or email == "" or telefono == "" or eta == "":
      QMessageBox.critical(self, 'Errore', 'Per favore, inserisci

      tutte le informazioni richieste',

      QMessageBox.Ok, QMessageBox.Ok)
   else:
   self.controller.aggiungi_studente(Studente((nome+cognome).lower(),
   nome, cognome, cf, indirizzo, email, telefono, eta))
      self.callback()
      self.close()
```

Listato 5.2: Classe `VistaInserisciStudente`

La classe `Studente` rappresenta un singolo studente con i suoi attributi, che vengono gestiti dal "controller" della classe `Studente`, rappresentato dalla classe `ControlloreStudente`(Listato 1.3), al cui interno sono definiti i metodi per ottenere gli attributi dello studente selezionato dal modello `Studente`:

```
class ControlloreStudente():
def __init__(self, studente):
   self.model = studente
```

```
def get_id_studente(self):
   return self.model.id

def get_nome_studente(self):
   return self.model.nome

def get_cognome_studente(self):
   return self.model.cognome

def get_cf_studente(self):
   return self.model.cf

def get_indirizzo_studente(self):
   return self.model.indirizzo

def get_email_studente(self):
   return self.model.email

def get_telefono_studente(self):
   return self.model.telefono

def get_eta_studente(self):
   return self.model.eta
```

Listato 5.3: Classe `ControlloreStudente`

La classe `VistaStudente` implementa una finestra che mostra i dati di un singolo studente e permette di eliminarlo. I metodi al suo interno sono due:

- `get-label-info`(self, testo, valore), che crea una riga di testo con le informazioni di uno specifico attributo dello studente;
- `elimina-studente-click(self)`, che elimina lo studente e chiude la finestra.

5.2 Codice relativo alla gestione dei dipendenti

In questa sezione illustreremo il codice utilizzato per la gestione dei dipendenti. La classe `ListaDipendenti`, illustrata nel Listato 1.4, è il modello principale che gestisce la lista dei dipendenti. Questo modello si occupa di caricare i dati dei dipendenti da un file pickle al momento dell'inizializzazione, aggiungere nuovi dipendenti, rimuovere un dipendente specifico usando il suo id, fornire accesso alla lista completa dei dipendenti o a un dipendente specifico tramite il suo indice e salvare la lista aggiornata nel file pickle.

```
class ListaDipendenti():
def __init__(self):
   super(ListaDipendenti, self).__init__()
   self.lista_dipendenti = []
   if os.path.isfile('listadipendenti/data/
   lista_dipendenti_salvata.pickle'):
      with open('listadipendenti/data/
      lista_dipendenti_salvata.pickle', 'rb') as f:
         self.lista_dipendenti = pickle.load(f)

def aggiungi_dipendente(self, dipendente):
   self.lista_dipendenti.append(dipendente)

def rimuovi_dipendente_by_id(self, id):
   def is_selected_dipendente(dipendente):
      if dipendente.id == id:
         return True
      return False
   self.lista_dipendenti.remove(list(filter(is_selected_dipendente,

   self.lista_dipendenti))[0])

def get_dipendente_by_index(self, index):
   return self.lista_dipendenti[index]

def get_lista_dipendenti(self):
   return self.lista_dipendenti

def save_data(self):
   with open('listadipendenti/data/lista_dipendenti_salvata.pickle',

   'wb') as handle:
      pickle.dump(self.lista_dipendenti,

      handle, pickle.HIGHEST_PROTOCOL)
```

Listato 5.4: Classe `ListaDipendenti`

La classe `ControlloreListaDipendenti` è un controller che fornisce un'interfaccia di accesso alle funzionalità di `ListaDipendenti` per la gestione dei dipendenti, mentre la classe `VistaListaDipendenti`, mostrata nel Listato 1.5, implementa la principale interfaccia grafica per la lista dei dipendenti. All'interno del costruttore viene definita una vista della lista dei dipendenti e i pulsanti "Apri", per visualizzare i dettagli di un dipendente selezionato, e "Nuovo", per aggiungere un nuovo dipendente, mentre i metodi utilizzati aggiornano la lista dei dipendenti dopo un'operazione e salvano i dati al momento della chiusura della finestra.

```
class VistaListaDipendenti(QWidget):
def __init__(self, parent=None):
   super(VistaListaDipendenti, self).__init__(parent)

   self.controller = ControlloreListaDipendenti()

   h_layout = QHBoxLayout()
   self.list_view = QListView()
   self.update_ui()
   h_layout.addWidget(self.list_view)

   buttons_layout = QVBoxLayout()
   open_button = QPushButton('Apri')
   open_button.clicked.connect(self.show_selected_info)
   buttons_layout.addWidget(open_button)
   new_button = QPushButton("Nuovo")
   new_button.clicked.connect(self.show_new_dipendente)
   buttons_layout.addWidget(new_button)
   buttons_layout.addStretch()
   h_layout.addLayout(buttons_layout)

   self.setLayout(h_layout)
   self.resize(600,300)
   self.setWindowTitle('Lista Dipendenti')

def update_ui(self):
   self.listview_model = QStandardItemModel(self.list_view)
   for dipendente in self.controller.get_lista_dipendenti():
      item = QStandardItem()
      item.setText(dipendente.nome + " " + dipendente.cognome)
      item.setEditable(False)
      font = item.font()
      font.setPointSize(18)
      item.setFont(font)
      self.listview_model.appendRow(item)
   self.list_view.setModel(self.listview_model)

def show_selected_info(self):
   if(len(self.list_view.selectedIndexes()) > 0):
      selected = self.list_view.selectedIndexes()[0].row()
      dipendente_selezionato =
      self.controller.get_dipendente_by_index(selected)
      self.vista_dipendente =
      VistaDipendente(dipendente_selezionato,
      self.controller.elimina_dipendente_by_id, self.update_ui)
      self.vista_dipendente.show()

def show_new_dipendente(self):
   self.vista_inserisci_dipendente =
   VistaInserisciDipendente(self.controller, self.update_ui)
   self.vista_inserisci_dipendente.show()

def closeEvent(self, event):
   self.controller.save_data()
```

Listato 5.5: Classe `VistaListaDipendenti`

La classe `VistaInserisciDipendente` implementa l'operazione di aggiunta di un nuovo dipendente. Al suo interno sono presenti due metodi:

- `add-info-text(self, nome, label)`, che aggiunge un campo di testo all'interfaccia per raccogliere informazioni sul dipendente;
- `add-dipendente(self)`, che aggiunge un nuovo dipendente, verifica che i campi da inserire non siano vuoti e aggiunge il dipendente alla lista, aggiornando l'interfaccia principale.

La classe `Dipendente` rappresenta un singolo dipendente con le sue proprietà, che vengono gestite dal "controller" della classe `Dipendente`, rappresentato dalla classe `ControlloreDipendent`

al cui interno sono definiti i metodi per ottenere gli attributi del dipendente selezionato dal modello `Dipendente`.

La classe `VistaDipendente` (Listato 1.6) implementa la finestra per visualizzare le informazioni di un singolo dipendente selezionato nella lista; essa visualizza i dettagli di un dipendente, definisce un pulsante "Elimina" che rimuove il dipendente selezionato dalla lista e aggiorna la list-view.

```
class VistaDipendente(QWidget):
def __init__(self, dipendente, elimina_dipendente, elimina_callback,

parent=None):
   super(VistaDipendente, self).__init__(parent)
   self.controller = ControlloreDipendente(dipendente)
   self.elimina_dipendente = elimina_dipendente
   self.elimina_callback = elimina_callback

   v_layout = QVBoxLayout()

   label_nome = QLabel(self.controller.get_nome_dipendente()

   + " " + self.controller.get_cognome_dipendente())
   font_nome = label_nome.font()
   font_nome.setPointSize(30)
   label_nome.setFont(font_nome)
   v_layout.addWidget(label_nome)

   v_layout.addItem(QSpacerItem(20, 40, QSizePolicy.Minimum,

   QSizePolicy.Expanding))

   v_layout.addWidget(self.get_info("Codice Fiscale:
   {}".format(self.controller.get_cf_dipendente())))
   v_layout.addWidget(self.get_info("data Nascita:
   {}".format(self.controller.get_datanascita_dipendente())))
   v_layout.addWidget(self.get_info("Luogo Nascita:
   {}".format(self.controller.get_luogonascita_dipendente())))
   v_layout.addWidget(self.get_info("Email:
   {}".format(self.controller.get_email_dipendente())))
   v_layout.addWidget(self.get_info("Telefono:
   {}".format(self.controller.get_telefono_dipendente())))
   v_layout.addWidget(self.get_info("Licenza:
   {}".format(self.controller.get_licenza_dipendente())))

   v_layout.addItem(QSpacerItem(20, 40, QSizePolicy.Minimum,

   QSizePolicy.Expanding))

   btn_elimina = QPushButton("Elimina")
   btn_elimina.clicked.connect(self.elimina_dipendente_click)
   v_layout.addWidget(btn_elimina)

   self.setLayout(v_layout)
   self.setWindowTitle(self.controller.get_nome_dipendente())

def get_info(self, text):
   label = QLabel(text)
   font = label.font()
   font.setPointSize(17)
   label.setFont(font)
   return label

def elimina_dipendente_click(self):
   self.elimina_dipendente(self.controller.get_id_dipendente())
   self.elimina_callback()
   self.close()
```

Listato 5.6: Classe `VistaDipendente`

5.3 Codice relativo alla gestione della biblioteca

Di seguito verrà illustrato come sono state implementate le operazioni per la gestione della biblioteca, che includono l'aggiunta, la rimozione e il salvataggio dei libri all'interno della lista dei libri. La classe `ListaLibri` gestisce una lista di libri, con operazioni di aggiunta, rimozione e salvataggio. Il costruttore, mostrato nel Listato 1.7, inizializza la lista dei libri come una lista vuota, controlla se esiste il file `lista-libri-salvata.pickle`, che contiene una versione salvata della lista dei libri e, se esiste, carica i dati dei libri salvati al suo interno, altrimenti carica i dati dei libri iniziali da un file JSON.

```
def __init__(self):
   super(ListaLibri, self).__init__()
   self.lista_libri = []
   if os.path.isfile('listalibri/data/lista_libri_salvata.pickle'):
      with open('listalibri/data/lista_libri_salvata.pickle',

      'rb') as f:
         self.lista_libri = pickle.load(f)
   else:
      with open('listalibri/data/lista_libri_iniziali.json') as f:
         lista_libri_iniziali = json.load(f)
      for libro_iniziale in lista_libri_iniziali:
         self.aggiungi_libro(Libro(libro_iniziale["id"],

         libro_iniziale["titolo"], libro_iniziale["categoria"]))
```

Listato 5.7: Costruttore della classe `ListaLibri`

In questa classe abbiamo inserito 5 metodi in fase di progettazione, che sono i seguenti:

- `aggiungi-libro(self, libro)`, che aggiunge un libro alla lista dei libri, prendendo come input libro, che è un'istanza della classe Libro;
- `get-libro-by-index(self, index)`, che restituisce un libro dalla lista in base al suo indice;
- `get-lista-libri(self)`, che restituisce l'intera lista dei libri;
- `rimuovi-libro-by-id(self, id)`, che rimuove un libro dalla lista dei libri in base all'id fornito;
- `save-data(self)`, che salva lo stato attuale della lista dei libri nel file pickle `lista-libri-salv`

La classe `ControlloreListaLibri` è un "controller" che gestisce le operazioni sulla lista dei libri e opera come intermediario tra la classe `ListaLibri` e le classi `VistaListaLibri` e `VistaInserisciLibro`, che andranno a implementare l'interfaccia grafica del software. Al suo interno vengono richiamati i metodi della classe `ListaLibri` che costituisce il "model".

La classe `VistaListaLibri` implementa la finestra in cui l'utente può vedere e gestire la lista dei libri.

Il costruttore `init` (Listato 1.8) inizializza la finestra principale e un'istanza di `ControlloreListaLi` visualizza la lista di libri in una `QListView`, dove ogni libro viene mostrato con il titolo, e infine configura i pulsanti "Apri", per visualizzare i dettagli del libro selezionato, e "Nuovo", per aggiungere un nuovo libro.

```
 def __init__(self, parent = None):
super(VistaListaLibri, self).__init__(parent)

self.controller = ControlloreListaLibri()

h_layout = QHBoxLayout()
self.list_view = QListView()
self.listview_model = QStandardItemModel(self.list_view)
for libro in self.controller.get_lista_dei_libri():
   item = QStandardItem()
   item.setText(libro.titolo)
   item.setEditable(False)
   font = item.font()
   font.setPointSize(18)
   item.setFont(font)
   self.listview_model.appendRow(item)
self.list_view.setModel(self.listview_model)
h_layout.addWidget(self.list_view)

buttons_layout = QVBoxLayout()
open_button = QPushButton("Apri")
open_button.clicked.connect(self.show_selected_info)
buttons_layout.addWidget(open_button)

new_button = QPushButton("Nuovo")
new_button.clicked.connect(self.show_new_libro)
buttons_layout.addWidget(new_button)
buttons_layout.addStretch()
h_layout.addLayout(buttons_layout)
```

```
        self.setLayout(h_layout)
        self.resize(600, 300)
        self.setWindowTitle('Lista Libri')
```

Listato 5.8: Costruttore della classe `VistaListaLibri`

I metodi utilizzati in questa classe sono quattro:

- `closeEvent(self, event)`, che salva i dati della lista dei libri quando la finestra viene chiusa;
- `show-selected-info(self)`, che mostra una finestra con le informazioni del libro selezionato;
- `show-new-libro(self)`, che apre una finestra per aggiungere un nuovo libro alla lista;
- `update-ui(self)`, che aggiorna la `QListView` con i libri attuali.

La classe `VistaInserisciLibro` consente all'utente di inserire un nuovo libro alla lista. I metodi in questa classe sono due:

- `add-info-text`, che aggiunge un campo di testo all'interfaccia per raccogliere informazioni sul libro;
- `add-libro` (Listato 1.9), che crea un nuovo libro, verifica che i campi da inserire non siano vuoti e aggiunge il libro alla lista, aggiornando l'interfaccia principale.

```
            def add_libro(self):
        for value in self.qlines.values():
           if value.text() == "":
              QMessageBox.critical(self, 'Errore', 'Per favore,

              inserisci tutte le informazioni richieste.',

              QMessageBox.Ok, QMessageBox.Ok)

              return
        self.controller.aggiungi_libro(Libro(
           (self.qlines["titolo"].text() +
           self.qlines["categoria"].text()).lower(),
           self.qlines["titolo"].text(),
           self.qlines["categoria"].text())

        )
        self.callback()
        self.close()
```

Listato 5.9: metodo `add-libro(self)`

La classe `Libro`, mostrata nel Listato 1.10, rappresenta un singolo libro con attributi e metodi associati. Il costruttore `init` inizializza un libro con ID, titolo, categoria e stato di disponibilità, mentre i due attributi controllano e modificano la disponibiltà del libro.

```
     class Libro():
    def __init__(self, id, titolo, categoria):
       super(Libro, self).__init__()
       self.id = id
       self.titolo = titolo
       self.categoria = categoria
       self.disponibile = True

    def is_disponibile(self):
       return self.disponibile

    def prenota(self):
       self.disponibile = False
```

Listato 5.10: Classe `Libro`

La classe `ControlloreLibro` gestisce le operazioni con un singolo libro, mentre la classe `VistaLibro` mostra i dettagli di un libro selezionato e permette di eliminarlo dalla lista dei libri, in particolare:

- il costruttore della classe `VistaLibro` configura la finestra per visualizzare i dati del libro selezionato e aggiunge un pulsante "Elimina" per rimuoverlo dalla lista dei libri;
- `elimina-libro-click(self)` è il metodo che permette la rimozione del libro selezionato.

5.4 Codice relativo alla prenotazione dei libri

In questa sezione illustreremo le classi e i moduli utilizzati per la gestione delle prenotazioni dei libri. La classe `ListaPrenotazioni` (Listato 1.11) costituisce il "model" della lista delle prenotazioni dei libri; il costruttore carica automaticamente una lista di prenotazioni salvata in un file pickle quando viene istanziata, mentre i metodi contenuti nella classe permettono di aggiungere e rimuovere una prenotazione, salvare la lista di prenotazioni attuale su file a aggiornare lo stato del libro prenotato o restituito.

```
class ListaPrenotazioni():
def __init__(self):
   super(ListaPrenotazioni, self).__init__()
   self.lista_prenotazioni = []

   if os.path.isfile('listaprenotazioni/data/

   lista_prenotazioni_salvata.pickle'):
      with open('listaprenotazioni/data/

      lista_prenotazioni_salvata.pickle', 'rb') as f:
         self.lista_prenotazioni = pickle.load(f)



def aggiungi_prenotazione(self, prenotazione):
   self.lista_prenotazioni.append(prenotazione)



def rimuovi_prenotazione_by_id(self, id):
   for prenotazione in self.lista_prenotazioni:
      print(f"Controlla prenotazione: {prenotazione},

      ID: {prenotazione.id}, target ID: {id}")
      if prenotazione.id == id:
         print(f"Ritorna {prenotazione.libro} disponibile")

         if os.path.isfile('listalibri/data/

         lista_libri_salvata.pickle'):
            with open('listalibri/data/lista_libri_salvata.pickle',

            'rb') as f:
               self.lista_libri_salvata = pickle.load(f)
         else:
            self.lista_libri_salvata = []

         # Cerca il libro prenotato nella lista dei libri
         for libro in self.lista_libri_salvata:
            if libro.id == prenotazione.libro.id:
               libro.disponibile = True
               break

self.lista_prenotazioni.remove(prenotazione)

         self.save_data()
         self.salva_lista_libri()



def get_prenotazione_by_index(self, index):
   return self.lista_prenotazioni[index]

def get_lista_prenotazioni(self):
   return self.lista_prenotazioni

def save_data(self):
   with open('listaprenotazioni/data/

   lista_prenotazioni_salvata.pickle', 'wb') as handle:
      pickle.dump(self.lista_prenotazioni, handle,
```

```
            pickle.HIGHEST_PROTOCOL)

    def salva_lista_libri(self):
        with open('listalibri/data/lista_libri_salvata.pickle', 'wb') as f:
            pickle.dump(self.lista_libri_salvata, f,

            pickle.HIGHEST_PROTOCOL)
```

Listato 5.11: Classe `ListaPrenotazioni`

La classe `ControlloreListaPrenotazioni` è un "controller" che gestisce la logica di comunicazione tra le classi `ListaPrenotazioni` e `VistaListaPrenotazioni`, che insieme alla classe `VistaInserisciPrenotazione`, implementano la parte di interfaccia utente riguardante la prenotazioni dei libri. All'interno di queste ultime due classi, i metodi che abbiamo inserito sono i seguenti:

- `update-ui(self)`, che aggiorna la vista con i dati attuali delle prenotazioni;
- `show-selected-info(self)`, che apre una finestra con i dati della prenotazione selezionata;
- `show-new-prenotazione(self)`, che apre una finestra per inserire la nuova prenotazione di un libro;
- `closeEvent(self, event)`, che salva i dati alla chiusura della finestra
- `add-prenotazione(self)`, che aggiunge la nuova prenotazione del libro nella lista delle prenotazioni.

La classe `Prenotazione` rappresenta una prenotazione individuale, che viene gestita dal controllore `ControllorePrenotazione`, mediante i metodi per ottenere ID, studente, libro e data di prenotazione. Infine La classe `VistaPrenotazione`, illustrata nel Listato 1.12, mostra i dettagli di una singola prenotazione, e include un pulsante "Disdici" per eliminare la prenotazione e rendere di nuovo disponibile il libro.

```
    class VistaPrenotazione(QWidget):
    def __init__(self, prenotazione, disdici_prenotazione,

    elimina_callback, parent=None):
        super(VistaPrenotazione, self).__init__(parent)
        self.controller = ControllorePrenotazione(prenotazione)
        self.disdici_prenotazione = disdici_prenotazione
        self.elimina_callback = elimina_callback

        v_layout = QVBoxLayout()

        label_titolo =
        QLabel(self.controller.get_libro_prenotazione().titolo)
        font_titolo = label_titolo.font()
        font_titolo.setPointSize(30)
        label_titolo.setFont(font_titolo)
        v_layout.addWidget(label_titolo)

        v_layout.addItem(QSpacerItem(20,40,QSizePolicy.Minimum,

        QSizePolicy.Expanding))

        label_studente = QLabel("Studente: {}
        {}".format(self.controller.get_studente_prenotazione().nome,
        self.controller.get_studente_prenotazione().cognome))
        font_studente = label_studente.font()
        font_studente.setPointSize(30)
        label_studente.setFont(font_studente)
        v_layout.addWidget(label_studente)

        label_data = QLabel("data:
        {}".format(self.controller.get_data_prenotazione()))
        font_data = label_data.font()
        font_data.setPointSize(30)
        label_data.setFont(font_data)
        v_layout.addWidget(label_data)

        v_layout.addItem(QSpacerItem(20, 40, QSizePolicy.Minimum,

        QSizePolicy.Expanding))
```

```
        btn_disdici = QPushButton("Disdici")
        btn_disdici.clicked.connect(self.disdici_prenotazione_click)
        v_layout.addWidget(btn_disdici)

        self.setLayout(v_layout)
        self.setWindowTitle(self.
        controller.get_libro_prenotazione().titolo)

    def disdici_prenotazione_click(self):
        self.disdici_prenotazione(self.controller.get_id_prenotazione())
        self.elimina_callback()
        self.close()
```

Listato 5.12: Classe `VistaPrenotazione`

5.5 Codice relativo alla prenotazione dei posti

Di seguito verrà illustrato come sono state implementate le operazioni per la gestione delle prenotazioni dei posti in biblioteca. La classe `ListaPrenotazioniPosto` gestisce l'elenco delle prenotazioni e dei posti. Essa contiene metodi per aggiungere e rimuovere prenotazioni, ottenere la lista dei posti disponibili, eliminare manualmente o automaticamente le prenotazioni scadute (oltre 3 ore dalla creazione), salvare e caricare i dati delle prenotazioni in un file pickle. Il "controller" `ControlloreListaPrenotazioniPosto` fa da interfaccia tra la vista e il modello `ListaPrenotazioniPosto`, permettendo l'accesso ai dati e alle funzionalità della lista delle prenotazioni. La vista relativa alle prenotazioni dei posti è costituita da due classi, ovvero: `VistaListaPrenotazioniPosto` e `VistaInserisciPrenotazioniPosto` (Listato 1.13).

```
    class VistaListaPrenotazioniPosto(QWidget):
    def __init__(self, parent=None):
        super(VistaListaPrenotazioniPosto, self).__init__(parent)
        self.controller = ControlloreListaPrenotazioniPosto()

        h_layout = QHBoxLayout()
        self.list_view = QListView()
        self.update_ui()
        h_layout.addWidget(self.list_view)

        buttons_layout = QVBoxLayout()
        open_button = QPushButton('Apri')
        open_button.clicked.connect(self.show_selected_info)
        buttons_layout.addWidget(open_button)
        book_button = QPushButton("Prenota Posto")
        book_button.clicked.connect(self.show_new_prenotazione_posto)
        buttons_layout.addWidget(book_button)

        buttons_layout.addStretch()
        h_layout.addLayout(buttons_layout)

        self.setLayout(h_layout)
        self.resize(600, 300)
        self.setWindowTitle('Lista Prenotazioni Posto')

    def update_ui(self):
        self.listview_model = QStandardItemModel(self.list_view)

        for prenotazione in
        self.controller.get_lista_delle_prenotazioni_posto():
            item = QStandardItem()
            item.setText(f"Studente: {prenotazione.studente.nome}

            {prenotazione.studente.cognome} -

            Posto {prenotazione.posto.id}")
            item.setEditable(False)
            font = item.font()
            font.setPointSize(18)
            item.setFont(font)
            self.listview_model.appendRow(item)

        self.list_view.setModel(self.listview_model)

    def show_selected_info(self):
        if len(self.list_view.selectedIndexes()) > 0:
            selected = self.list_view.selectedIndexes()[0].row()
            prenotazione =
            self.controller.get_prenotazione_posto_by_index(selected)
            if prenotazione is None:
                print("Errore: prenotazione non trovata.")
                return
            self.vista_prenotazione_posto =
```

```
            VistaPrenotazionePosto(prenotazione,

        self.controller.elimina_prenotazione_posto,
        self.update_ui)
            self.vista_prenotazione_posto.show()

    def show_new_prenotazione_posto(self):
        self.vista_inserisci_prenotazioni_posto =
        VistaInserisciPrenotazioniPosto(self.controller,

        self.update_ui)
        self.vista_inserisci_prenotazioni_posto.show()
        pass

    def closeEvent(self, event):
        try:
            self.controller.save_data()
            print("Dati salvati con successo.")
        except Exception as e:
            print(f"Errore durante il salvataggio dei dati: {e}")
```

Listato 5.13: Classe `VistaListaPrenotazioniPosto`

La singola prenotazione del posto in biblioteca è gestita, sempre con la tecnologia model-view-controller, dalle classi:

- `PrenotazionePosto`, contenente l'ID della prenotazione, la data della prenotazione e una funzione per determinare se la prenotazione è scaduta;
- `ControllorePrenotazionePosto`, che fornisce un'interfaccia per ottenere i dettagli di una singola prenotazione;
- `VistaPrenotazionePosto`, che rappresenta la finestra grafica per visualizzare i dettagli di una singola prenotazione.

5.6 Manuale del sistema

Questa sezione descrive le funzionalità del sistema e guida l'utente nelle principali operazioni. Ogni sottosezione si concentra su un aspetto specifico, presentando i passi da seguire in modo chiaro e lineare.

5.6.1 Gestione degli studenti

Per accedere alla lista degli studenti, è necessario aprire l'applicazione e cliccare sul pulsante "Studenti"; nella finestra principale si caricherà automaticamente la lista degli studenti già salvati nel sistema (Figura 1.1) . La lista mostra i nomi completi degli studenti registrati.

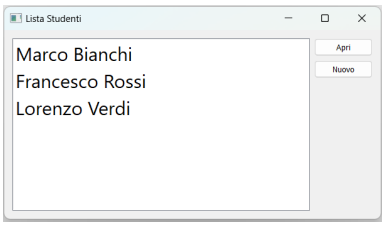

Figura 5.1: `VistaListaStudenti`

Per aggiungere un nuovo studente, cliccare sul pulsante "Nuovo" nella finestra principale. Si aprirà una nuova finestra in cui è necessario inserire tutte le informazioni richieste: nome, cognome, codice fiscale, indirizzo, email, telefono ed età (Figura 1.2). Dopo aver compilato i campi, è necessario cliccare su "OK" per confermare l'aggiunta.

Figura 5.2: `VistaInserisciStudente`

Se si desidera visualizzare i dettagli di uno studente, si deve selezionare il nome nella lista e cliccare su "Apri". Verrà aperta una finestra che mostrerà tutte le informazioni relative allo studente selezionato (Figura 1.3).

Figura 5.3: `VistaStudente`

Per rimuovere uno studente dal sistema, è necessario accedere alla finestra dei dettagli cliccando su "Apri" e poi selezionare "Elimina". Dopo la rimozione, la lista principale verrà aggiornata automaticamente. Infine, il sistema salva i dati automaticamente alla chiusura dell'applicazione. Tutte le informazioni vengono archiviate nel file `lista-studenti-salvata.pickle`, che verrà caricato automaticamente all'apertura successiva del programma.

5.6.2 Gestione dei dipendenti

La gestione dei dipendenti funziona in modo simile a quella degli studenti. Per aggiungere un nuovo dipendente, è necessario accedere alla vista principale che mostra la lista dei dipendenti (Figura 1.4) e cliccare su "Nuovo". Si aprirà una finestra dove sarà possibile inserire le informazioni richieste, come nome, cognome, codice fiscale, ecc. (Figura 1.5). Una volta compilati i campi, si deve cliccare su "OK" per salvare il nuovo profilo.

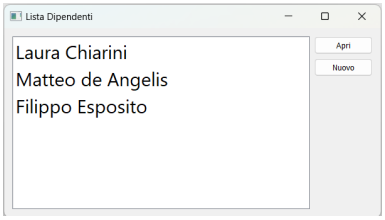

Figura 5.4: `VistaListaDipendenti`

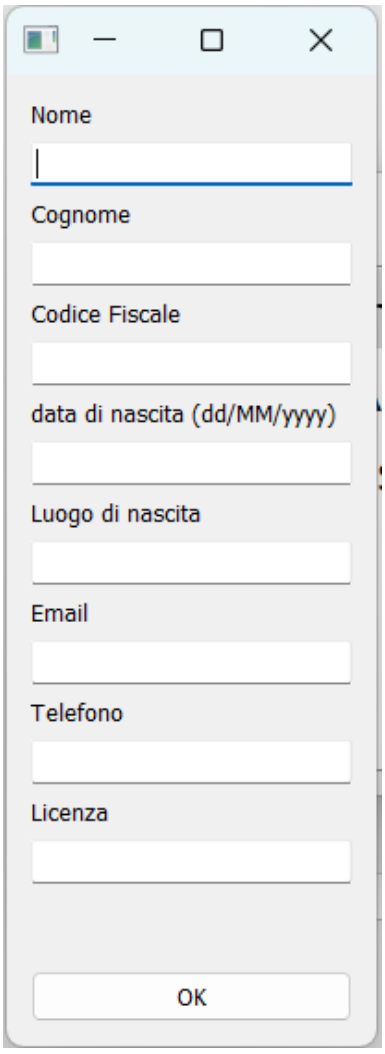

Figura 5.5: `VistaInserisciDipendente`

Per visualizzare i dettagli di un dipendente, è necessario selezionarlo dalla lista e cliccare su "Apri". I dettagli saranno mostrati in una nuova finestra (Figura 1.6). Nel caso si voglia

rimuovere un dipendente, bisogna accedere alla finestra dei dettagli e cliccare su "Elimina". La lista si aggiornerà automaticamente.

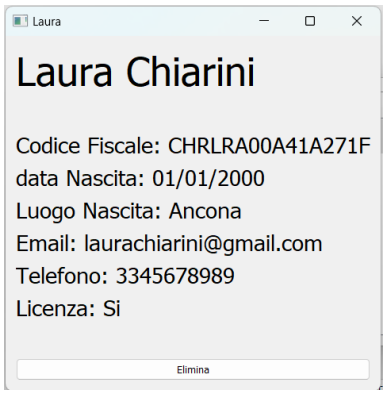

Figura 5.6: `VistaDipendente`

Anche per i dipendenti, il salvataggio dei dati avviene automaticamente alla chiusura del programma, utilizzando un file pickle dedicato.

5.6.3 Gestione della biblioteca

La gestione della biblioteca si svolge nella finestra principale, chiamata `VistaListaLibri` (Figura 1.7). Questa finestra mostra l'elenco di tutti i libri disponibili. È possibile accedere ai dettagli di un libro cliccando su "Apri"; in alternativa aggiungere un nuovo libro cliccando su "Nuovo".

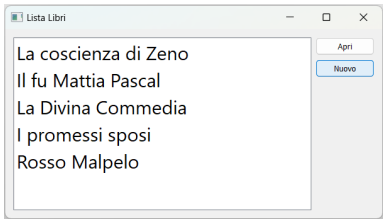

Figura 5.7: `VistaListaLibri`

Quando si sceglie di aggiungere un nuovo libro, viene visualizzata la finestra `VistaInserisciLibr` (Figura 1.8), dove è necessario compilare i dati del libro, come titolo, categoria e stato di disponibilità. Una volta completata l'operazione, si deve cliccare su "OK" per salvare il libro. Se si tenta di salvare un libro con campi vuoti, il sistema mostrerà un messaggio d'errore.

Per eliminare un libro, è necessario accedere ai suoi dettagli (Figura 1.9) tramite il pulsante "Apri" e selezionare "Elimina". Il libro verrà rimosso e la lista verrà aggiornata automaticamente.

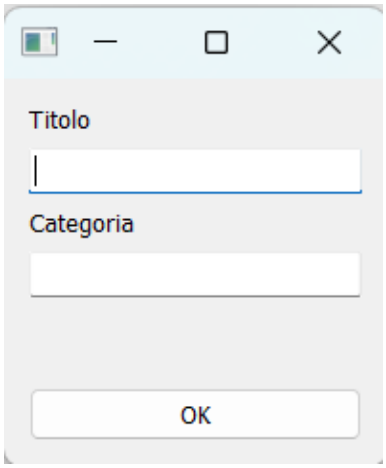

Figura 5.8: `VistaInserisciLibro`

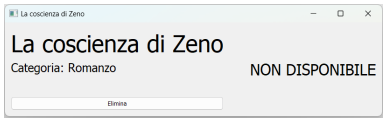

Figura 5.9: `VistaLibro`

Come per gli altri moduli, i dati della libreria vengono salvati automaticamente alla chiusura del programma.

5.6.4 Prenotazione dei libri

La finestra principale dedicata alle prenotazioni, chiamata `VistaListaPrenotazioni` (Figura 1.10), consente di visualizzare tutte le prenotazioni attive. Ogni prenotazione è indicata con il nome dello studente e il titolo del libro.

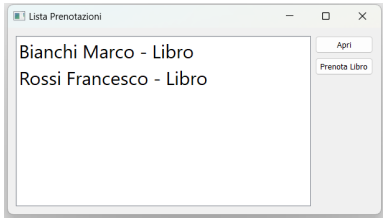

Figura 5.10: `VistaListaPrenotazioni`

Per effettuare una nuova prenotazione, è necessario cliccare su "Prenota Libro". Si aprirà la finestra `VistaInserisciPrenotazione` (Figura 1.11), dove sarà possibile selezionare uno studente e un libro dai menù a discesa. È, inoltre, necessario inserire la data nel formato *dd/MM/yyyy* . Dopo aver completato i campi, è necessario cliccare su "OK". Il sistema registrerà la prenotazione e cambierà lo stato del libro in "non disponibile".

Per visualizzare i dettagli di una prenotazione, selezionarla dalla lista e cliccare su "Apri". Verranno mostrati il titolo del libro, il nome dello studente e la data della prenotazione. Se si vuole annullare una prenotazione, è necessario accedere ai dettagli e cliccare su "Disdici" (Figura 1.12). L'operazione eliminerà la prenotazione e il libro tornerà disponibile.

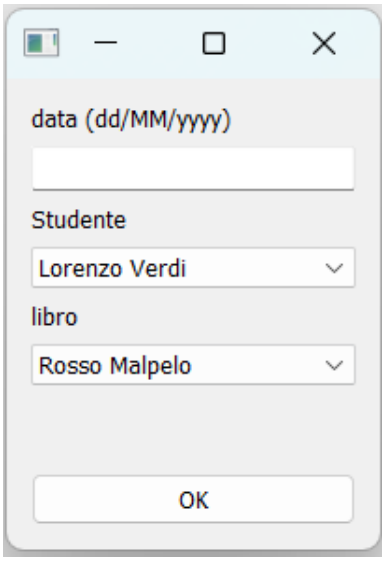

Figura 5.11: `VistaInserisciPrenotazione`

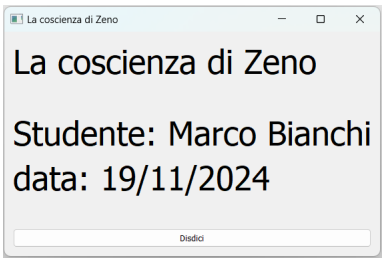

Figura 5.12: `VistaPrenotazione`

Anche in questo caso, i dati vengono salvati automaticamente alla chiusura.

5.6.5 Prenotazione dei posti

La prenotazione dei posti segue una logica simile a quella dei libri. Per prenotare un posto, aprire la finestra `VistaInserisciPrenotazioniPosto` (Figura 1.13), selezionare uno studente e scegliere un posto disponibile. Una volta confermata l'operazione, la prenotazione sarà salvata e il posto sarà segnato come occupato.

Le prenotazioni attive sono visibili nella finestra `VistaListaPrenotazioniPosto` (Figura 1.14), dove sono elencati il nome dello studente e il numero del posto.

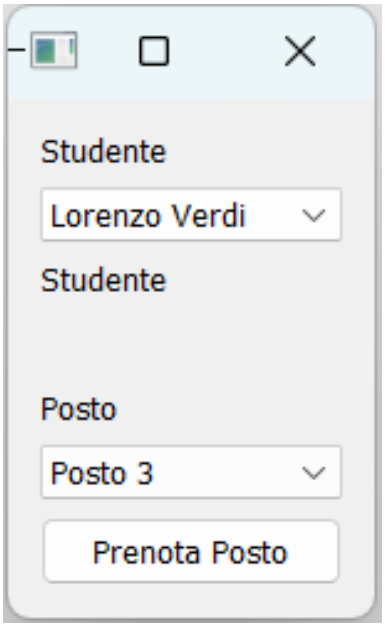

Figura 5.13: `VistaInserisciPrenotazioniPosto`

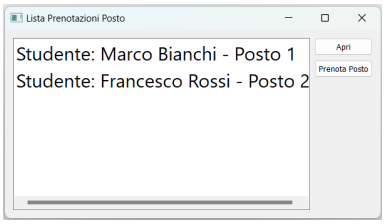

Figura 5.14: `VistaListaPrenotazioniPosto`

È possibile annullare una prenotazione specifica accedendo ai dettagli e cliccando sul pulsante "Disdici". Inoltre, il sistema elimina automaticamente le prenotazioni scadute. Come per gli altri moduli, i dati delle prenotazioni dei posti vengono salvati automaticamente.

CAPITOLO 6

Testing

Il processo di testing del software riveste un ruolo cruciale nella verifica e validazione dei sistemi, con l'obiettivo di assicurare che un programma sia in grado di svolgere i compiti per cui è stato progettato, individuando eventuali errori prima della sua messa in produzione. Il processo di testing si articola su due obiettivi principali:

- *dimostrare che il software soddisfi i requisiti stabiliti;*
- *individuare comportamenti errati o non conformi.*

Il primo obiettivo corrisponde al processo di test di convalida, che si concentra sul controllo del corretto funzionamento del sistema attraverso l'utilizzo di casi di test rappresentativi dell'uso previsto. Il secondo obiettivo, invece, si riferisce al test dei difetti, volto alla progettazione di casi di test specifici per scoprire eventuali malfunzionamenti e anomalie nel software. Nel presente capitolo verranno illustrati i diversi tipi di test applicati al software oggetto di questa tesi, con l'obiettivo di verificare e garantire il corretto funzionamento del programma.

6.1 Test studenti

Nella classe `TestListaStudenti` (Listato 1.1) sono stati implementati quattro metodi per verificare il corretto funzionamento delle principali funzionalità della classe `ListaStudenti`. Di seguito viene fornita una descrizione dettagliata del loro scopo:

- `test-aggiungi-studente` : questo metodo verifica che uno studente venga aggiunto correttamente alla lista tramite l'utilizzo della funzione `aggiungi-studente`;
- `test-rimuovi-studente` : valida che la funzione `rimuovi-studente-by-id` rimuova in modo appropriato uno specifico studente dalla lista, utilizzando il relativo ID come criterio di selezione;
- `test-get-studente-by-index` : controlla che la funzione `get-studente-by-index` restituisca lo studente corretto in base alla posizione indicata nella lista;
- `test-save-and-load` : assicura che il metodo `save-data` salvi accuratamente i dati su disco e che il caricamento automatico della lista permetta di ripristinare gli studenti senza alcuna perdita di informazioni.

```
import unittest
from listastudenti.model.ListaStudenti import ListaStudenti
from studente.model.Studente import Studente

class TestListaStudenti(unittest.TestCase):

   def setUp(self):
      self.lista_studenti = ListaStudenti()
      self.studente1 = Studente("id1", "Marco", "Bianchi", "RSSMRA70A01F205Z", "Via Roma 1", "
           marcobianchi@gmail.com", "1234567890", "20")
      self.studente2 = Studente("id2", "Francesco", "Rossi", "BNCALC80B01F205X", "Via Milano 2", "
           francescorossi@gmail.com", "0987654321", "20")
      self.lista_studenti.aggiungi_studente(self.studente1)
      self.lista_studenti.aggiungi_studente(self.studente2)

   def test_aggiungi_studente(self):
      studente3 = Studente("id3", "Lorenzo", "Verdi", "VRDANN85C01F205Y", "Via Napoli 3", "
           lorenzoverdi@gmail.com", "1122334455", "20")
      self.lista_studenti.aggiungi_studente(studente3)
      self.assertIn(studente3, self.lista_studenti.get_lista_studenti())

   def test_rimuovi_studente(self):
      self.lista_studenti.rimuovi_studente_by_id("id1")
      self.assertNotIn(self.studente1, self.lista_studenti.get_lista_studenti())

   def test_get_studente_by_index(self):
      studente = self.lista_studenti.get_studente_by_index(0)
      self.assertEqual(studente, self.studente1)

   def test_save_and_load(self):
      self.lista_studenti.save_data()
      nuova_lista = ListaStudenti()
      self.assertEqual(len(nuova_lista.get_lista_studenti()), len(self.lista_studenti.get_lista_studenti()
           ))
```

Listato 6.1: Classe `TestListaStudenti`

Per eseguire i test, è necessario accedere al terminale dell'ambiente di sviluppo ed utilizzare il comando: `python -m unittest nome-file.py`. Per eseguire i test descritti di seguito, è sufficiente sostituire `nome-file` con il nome specifico del file contenente i test desiderati. Nel caso in cui si vogliano eseguire i test relativi alla classe `TestListaStudenti`, il comando sarà: `python -m unittest TestListaStudenti.py`

6.2 Test dipendenti

La classe `TestControlloreDipendente`, mostrata nel Listato 1.2, verifica che la classe `ControlloreDipendente` acceda correttamente agli attributi del modello `Dipendente`.

```
   class TestControlloreDipendente(unittest.TestCase):
   def test_controllore_accesso(self):
      dipendente = Dipendente(
         id=1,
         nome="Laura",
         cognome="Chiarini",
         datanascita="01/01/2000",
         luogonascita="Ancona",
         cf="CHRLRA00A41A271F",
         telefono="3345678989",
         email="laurachiarini@gmail.com",
         licenza="Si"
      )
      controllore = ControlloreDipendente(dipendente)

      self.assertEqual(controllore.get_id_dipendente(), 2)
      self.assertEqual(controllore.get_nome_dipendente(), "Laura")
      self.assertEqual(controllore.get_cognome_dipendente(), "Chiarini")
      self.assertEqual(controllore.get_cf_dipendente(), "CHRLRA00A41A271F")
      self.assertEqual(controllore.get_datanascita_dipendente(), "01/01/2000")
      self.assertEqual(controllore.get_luogonascita_dipendente(), "Ancona")
      self.assertEqual(controllore.get_email_dipendente(), "laurachiarini@gmail.com")
      self.assertEqual(controllore.get_telefono_dipendente(), "3345678989")
      self.assertEqual(controllore.get_licenza_dipendente(), "Si")
```

Listato 6.2: Classe `TestControlloreDipendente`

La classe `TestVistaDipendente` (Listato 1.3) verifica che `VistaDipendente` mostri correttamente i dati del dipendente e chiami le callback appropriate quando il pulsante di eliminazione viene premuto. Per fare ciò è stato utilizzato il modulo `MagicMock`, al fine di simulare il comportamento delle funzioni di callback.

```
    from unittest.mock import MagicMock

class TestVistaDipendente(unittest.TestCase):
    def test_interfaccia_elimina_dipendente(self):
        dipendente = Dipendente(
            id=3,
            nome="Matteo",
            cognome="de Angelis",
            datanascita="01/01/2000",
            luogonascita="Ancona",
            cf="DNLMTT3555CGE2",
            telefono="3334567898",
            email="matteodeangelis@gmail.com",
            licenza="Si"
        )

        elimina_dipendente_mock = MagicMock()
        elimina_callback_mock = MagicMock()

        vista = VistaDipendente(dipendente, elimina_dipendente_mock, elimina_callback_mock)

        self.assertEqual(vista.windowTitle(), "Matteo")

        for i in range(vista.layout().count()):
            widget = vista.layout().itemAt(i).widget()
            if isinstance(widget, QPushButton) and widget.text() == "Elimina":
                widget.click()

        elimina_dipendente_mock.assert_called_once_with(3)
        elimina_callback_mock.assert_called_once()
```

Listato 6.3: Classe `TestVistaDipendente`

6.3 Test biblioteca

Nella classe `TestListaLibri` (Listato 1.4) sono stati implementati tre test. Il metodo `setUp` prepara i dati iniziali per effettuare le verifiche; successivamente vengono effettuati i veri e propri test; questi ultimi sono descritti di seguito:

- `test-caricamento-dati-iniziali`, verifica che `ListaLibri` carichi i dati iniziali dal file `JSON` quando il file `pickle` non è presente;
- `test-aggiunta-rimozione-libro`, controlla che i metodi `aggiungi-libro` e `rimuovi-libro-by-id` funzionino correttamente;
- `test-salvataggio-dati`, verifica che il metodo `save-data` salvi correttamente i dati in formato `pickle`.

```
    import unittest
import os
from unittest.mock import patch, mock_open
import json
import pickle
from libro.model.Libro import Libro
from listalibri.model.ListaLibri import ListaLibri

class TestListaLibri(unittest.TestCase):
    def setUp(self):

        self.mock_initial_data = [
            {"id": "libro1", "titolo": "La coscienza di Zeno", "categoria": "Romanzo"},
            {"id": "libro2", "titolo": "Il fu Mattia Pascal", "categoria": "Romanzo"}
        ]
        self.lista_libri_file = 'listalibri/data/lista_libri_iniziali.json'
        self.lista_pickle_file = 'listalibri/data/lista_libri_salvata.pickle'

    @patch("builtins.open", new_callable=mock_open, read_data=json.dumps(mock_initial_data))
    @patch("os.path.isfile", return_value=False)
    def test_caricamento_dati_iniziali(self, mock_isfile, mock_open):
        lista_libri = ListaLibri()
        self.assertEqual(len(lista_libri.get_lista_libri()), 2)
        self.assertEqual(lista_libri.get_lista_libri()[0].titolo, "Libro 1")
        self.assertEqual(lista_libri.get_lista_libri()[1].categoria, "Saggistica")

    def test_aggiunta_rimozione_libro(self):

        libro = Libro("libro3", "Rosso Malpelo", "Romanzo")
        lista_libri = ListaLibri()
```

```
        lista_libri.aggiungi_libro(libro)
        self.assertEqual(len(lista_libri.get_lista_libri()), 1)

        lista_libri.rimuovi_libro_by_id("libro3")
        self.assertEqual(len(lista_libri.get_lista_libri()), 0)

    @patch("builtins.open", new_callable=mock_open)
    def test_salvataggio_dati(self, mock_open):
        lista_libri = ListaLibri()
        libro = Libro("libro4", "I Promessi Sposi", "Romanzo")
        lista_libri.aggiungi_libro(libro)
        lista_libri.save_data()

        mock_open.assert_called_with(self.lista_pickle_file, 'wb')
        handle = mock_open()
        handle.write.assert_called()
```

Listato 6.4: Classe `TestListaLibri`

6.4 Test prenotazione libri

I test implementati nella classe `TestListaPrenotazioni`, mostrata nel Listato 1.5, sono tre. Il loro scopo è, rispettivamente, quello di verificare l'aggiunta di una prenotazione, la rimozione di quest'ultima e assicurare che l'utente possa accedere alla lista delle prenotazioni correttamente. I metodi utilizzati sono descritti di seguito:

- `test-aggiungi-prenotazione`; ha lo scopo di verificare che una nuova prenotazione venga aggiunta correttamente alla lista e salvata nel file `pickle`, prendendo come input uno studente e un libro disponibile;
- `test-rimuovi-prenotazione-by-id`; verifica che una prenotazione venga rimossa correttamente e che lo stato del libro preso in prestito venga aggiornato;
- `test-get-lista-delle-prenotazioni`; si assicura che la lista delle prenotazioni venga recuperata correttamente dal controller `ControlloreListaPrenotazioni`.

```
    import unittest
import os
import pickle
from unittest.mock import MagicMock
from listaprenotazioni.model.ListaPrenotazioni import ListaPrenotazioni
from listaprenotazioni.controller.ControlloreListaPrenotazioni import ControlloreListaPrenotazioni
from prenotazione.model.Prenotazione import Prenotazione
from studente.model.Studente import Studente
from libro.model.Libro import Libro

class TestListaPrenotazioni(unittest.TestCase):
    def setUp(self):
        self.file_prenotazioni = 'listaprenotazioni/data/lista_prenotazioni_salvata.pickle'
        self.file_libri = 'listalibri/data/lista_libri_salvata.pickle'

        self.libro1 = Libro("1", "Libro A", True)
        self.libro2 = Libro("2", "Libro B", True)
        self.lista_libri = [self.libro1, self.libro2]
        os.makedirs(os.path.dirname(self.file_libri), exist_ok=True)
        with open(self.file_libri, 'wb') as f:
            pickle.dump(self.lista_libri, f)

        self.studente1 = Studente("Marco", "Bianchi")
        self.prenotazione1 = Prenotazione("marcobianchi", self.studente1, self.libro1, "25/11/2024")
        self.lista_prenotazioni = [self.prenotazione1]
        os.makedirs(os.path.dirname(self.file_prenotazioni), exist_ok=True)
        with open(self.file_prenotazioni, 'wb') as f:
            pickle.dump(self.lista_prenotazioni, f)

    def clear(self):

        if os.path.isfile(self.file_prenotazioni):
            os.remove(self.file_prenotazioni)
        if os.path.isfile(self.file_libri):
            os.remove(self.file_libri)

    def test_aggiungi_prenotazione(self):

        lista_prenotazioni = ListaPrenotazioni()
        studente2 = Studente("Francesco", "Rossi")
        prenotazione2 = Prenotazione("francescorossi", studente2, self.libro2, "26/11/2024")
```

```
        lista_prenotazioni.aggiungi_prenotazione(prenotazione2)

        self.assertEqual(len(lista_prenotazioni.get_lista_prenotazioni()), 2)
        self.assertEqual(lista_prenotazioni.get_lista_prenotazioni()[-1], prenotazione2)

    def test_rimuovi_prenotazione_by_id(self):

        lista_prenotazioni = ListaPrenotazioni()
        lista_prenotazioni.rimuovi_prenotazione_by_id("marcobianchi")

        self.assertEqual(len(lista_prenotazioni.get_lista_prenotazioni()), 0)
        with open(self.file_libri, 'rb') as f:
            libri = pickle.load(f)
        self.assertTrue(libri[0].disponibile)

    def test_get_lista_delle_prenotazioni(self):

        controller = ControlloreListaPrenotazioni()
        lista_prenotazioni = controller.get_lista_delle_prenotazioni()

        self.assertEqual(len(lista_prenotazioni), 1)
        self.assertEqual(lista_prenotazioni[0].id, "marcobianchi")

if __name__ == '__main__':
    unittest.main()
```

Listato 6.5: Classe `TestListaPrenotazioni`

6.5 Test prenotazione posti

Nella classe `TestVistaListaPrenotazioniPosto` (Listato 1.6) i test implementati hanno lo scopo di controllare la disponibilità dei posti nella biblioteca e assicurare il corretto salvataggio delle prenotazioni in un file `pickle`. Inizialmente sono stati definiti il metodo `setUp`, che configura i dati iniziali, i file `mock` per i test, e il metodo `clear`, che rimuove i file creati per simulare l'esecuzione dell'applicazione. I test utilizzati in questa classe sono descritti di seguito:

- `test-get-posti-disponibili`: recupera la lista dei posti disponibili e verifica che quest'ultima contenga solo i posti liberi;
- `test-salvataggio-dati-prenotazioni`: assicura che le modifiche alle prenotazioni vengano salvate correttamente;
- `test-caricamento-dati-prenotazioni`: garantisce che il sistema carichi correttamente i dati delle prenotazioni salvati.

```
class TestVistaListaPrenotazioniPosto(unittest.TestCase):
    def setUp(self):
        self.file_studenti = 'listastudenti/data/lista_studenti_salvata.pickle'
        self.file_prenotazioni = 'listaprenotazioniposto/data/lista_prenotazioni_posto_salvata.pickle'
        self.file_posti = 'listaposti/data/lista_posti_salvata.pickle'

        self.studente1 = Studente("1", "Marco", "Bianchi")
        self.studente2 = Studente("2", "Francesco", "Rossi")
        self.posto1 = Posto("1")
        self.posto2 = Posto("2")
        self.prenotazione1 = PrenotazionePosto("1", self.studente1, self.posto1)

        os.makedirs(os.path.dirname(self.file_studenti), exist_ok=True)
        with open(self.file_studenti, 'wb') as f:
            pickle.dump([self.studente1, self.studente2], f)

        os.makedirs(os.path.dirname(self.file_posti), exist_ok=True)
        with open(self.file_posti, 'wb') as f:
            pickle.dump([self.posto1, self.posto2], f)

        os.makedirs(os.path.dirname(self.file_prenotazioni), exist_ok=True)
        with open(self.file_prenotazioni, 'wb') as f:
            pickle.dump([self.prenotazione1], f)

        self.controller = ControlloreListaPrenotazioniPosto()

    def clear(self):
        if os.path.isfile(self.file_studenti):
            os.remove(self.file_studenti)
        if os.path.isfile(self.file_posti):
```

```
            os.remove(self.file_posti)
        if os.path.isfile(self.file_prenotazioni):
            os.remove(self.file_prenotazioni)

    def test_get_posti_disponibili(self):
        posti_disponibili = self.controller.get_posti_disponibili()
        self.assertEqual(len(posti_disponibili), 1)
        self.assertEqual(posti_disponibili[0].id, "2")

    def test_salvataggio_dati_prenotazioni(self):


        studente = self.controller.get_studenti()[1]
        posto = self.controller.get_posti_disponibili()[0]
        nuova_prenotazione = PrenotazionePosto(studente.id, studente, posto)
        self.controller.aggiungi_prenotazione_posto(nuova_prenotazione)
        self.controller.save_data()

        with open(self.file_prenotazioni, 'rb') as f:
            prenotazioni_salvate = pickle.load(f)
        self.assertEqual(len(prenotazioni_salvate), 2)
        self.assertEqual(prenotazioni_salvate[-1].posto.id, "2")

    def test_caricamento_dati_prenotazioni(self):

      self.controller.carica_dati()


        prenotazioni = self.controller.get_lista_delle_prenotazioni_posto()
        self.assertEqual(len(prenotazioni), 1)
        self.assertEqual(prenotazioni[0].studente.nome, "Marco")
        self.assertEqual(prenotazioni[0].posto.id, "1")

if __name__ == '__main__':
    unittest.main()
```

Listato 6.6: Classe `TestVistaListaPrenotazioniPosto`

CAPITOLO 7

Discussione

In questo ultimo capitolo, si analizzano e interpretano i risultati ottenuti durante il processo di progettazione e sviluppo dell'applicazione software per la gestione di una biblioteca. L'obiettivo è riflettere sul percorso intrapreso, valutando l'efficacia delle scelte tecniche, metodologiche e progettuali effettuate. Inoltre, vengono presentate delle prospettive di miglioramento e sviluppo futuro del sistema.

7.1 Analisi dei risultati ottenuti

Il progetto si proponeva di realizzare un'applicazione software in grado di gestire in modo efficace ed intuitivo le principali operazioni di una biblioteca, tra cui l'organizzazione del catalogo, la registrazione delle prenotazioni e la gestione degli utenti. L'obiettivo è stato raggiunto attraverso l'applicazione di un approccio metodologico rigoroso e l'adozione di strumenti e tecniche consolidati nell'ambito dell'ingegneria del software. I risultati ottenuti possono essere suddivisi in due ambiti principali:

- *Funzionalità del sistema*: l'applicazione offre tutte le funzionalità pensate nella fase progettuale, supportando le operazioni standard della biblioteca con un'interfaccia utente intuitiva. L'implementazione ha soddisfatto i requisiti funzionali identificati in fase di analisi, come dimostrato dai test eseguiti durante la fase di validazione.
- *Qualità del software*: grazie all'adozione del pattern architetturale MVC, il sistema risulta modulare e manutenibile. Le interazioni tra le componenti sono state progettate per garantire flessibilità e scalabilità, permettendo futuri aggiornamenti senza dover modificare integralmente lo scheletro dell'applicazione.

7.2 Interpretazione dei risultati

L'analisi complessiva del progetto suggerisce che il software sviluppato rappresenta una soluzione valida ed efficace per la gestione di una biblioteca. Tra i principali punti di forza si annoverano:

- *Facilità d'uso*: l'interfaccia utente è stata progettata con un focus sull'usabilità, consentendo un apprendimento rapido anche a utenti con limitata esperienza tecnologica.

- *Adattabilità* : il sistema è stato progettato per essere adattabile a biblioteche di diversa dimensione, dalla piccola biblioteca comunale a sistemi più complessi, grazie alla modularità delle sue componenti.
- *Affidabilità* : i test effettuati non hanno evidenziato malfunzionamenti critici, confermando la robustezza del sistema.

Tuttavia, durante lo sviluppo del progetto, sono emerse alcune aree che potrebbero beneficiare di ulteriori miglioramenti, rappresentando spunti significativi per future ottimizzazioni. Il sistema, pur soddisfacendo pienamente i requisiti iniziali, non offre al momento integrazioni dirette con servizi esterni, come piattaforme di catalogazione online o librerie di e-book; l'introduzione di tali funzionalità amplierebbe significativamente l'esperienza degli utenti, favorendo una maggiore interoperabilità. Infine, nonostante il software sia stato progettato per essere scalabile, un utilizzo su larga scala potrebbe richiedere ulteriori interventi di ottimizzazione, in particolare per garantire un'efficace gestione delle risorse in contesti distribuiti o caratterizzati da un elevato numero di operazioni simultanee.

7.3 Contributo al contesto applicativo

Il software sviluppato offre un contributo significativo al contesto di riferimento, poiché risponde alle esigenze specifiche di una biblioteca moderna, fornendo strumenti per migliorare la gestione operativa e l'interazione con gli utenti. L'adozione del sistema potrebbe ridurre gli errori manuali, ottimizzare i tempi di gestione e incrementare la soddisfazione degli utenti finali.

Dal punto di vista metodologico, il progetto dimostra come l'applicazione di principi ingegneristici e pattern consolidati possa portare a soluzioni robuste e adattabili. La documentazione prodotta durante le varie fasi dello sviluppo costituisce un ulteriore valore aggiunto, offrendo una base solida per eventuali modifiche o estensioni future.

7.4 Possibili miglioramenti

Le potenziali direzioni per lo sviluppo futuro del sistema includono:

- *Automazione avanzata*: integrazione di funzionalità basate su Intelligenza Artificiale, come la classificazione automatica dei libri o l'integrazione di assistenti virtuali che aiutino l'utente.
- *Supporto mobile*: sviluppo di una versione mobile dell'applicazione per consentire agli utenti di accedere ai servizi della biblioteca da dispositivi portatili.
- *Integrazione con piattaforme esterne*: implementazione di API per l'interoperabilità con altri sistemi bibliotecari o piattaforme di gestione di documenti.

7.5 Conclusioni finali

Il lavoro svolto in questa tesi rappresenta un esempio concreto di come le metodologie ingegneristiche possano essere applicate efficacemente per risolvere problemi reali. L'applicazione per la gestione della biblioteca ha dimostrato di essere uno strumento utile e flessibile, rispondendo pienamente agli obiettivi prefissati. Le considerazioni emerse durante la discussione offrono spunti significativi per il miglioramento e lo sviluppo futuro del sistema, garantendo un contributo duraturo al contesto applicativo e una base solida per ulteriori innovazioni.

Bibliografia

BARUFFINI, F. e OTHERS (2009), «Impiego di un Framework MVC nella realizzazione di un applicativo gestionale», .

BOSCAINI, M. (2017), *Imparare a programmare con Python*, 1, Apogeo Editore.

BUTTU, M. (2014), *Programmare con Python: Guida completa*, LSWR.

DAMIANI, E., MADRAVIO, M. e BÖHM, A. (2007), *UML pratico con elementi di ingegneria del software*, Pearson Italia Spa.

DE MIGUEL, L., LÓPEZ MAESTRESALAS, A. e LÓPEZ MOLINA, C. (2020), «A gentle introduction to Python», .

GHEZZI, C., JAZAYERI, M. e MANDRIOLI, D. (2004), *Ingegneria del software: fondamenti e principi*, Pearson Italia Spa.

GROVE, R. F. e OZKAN, E. (2011), «The MVC-web design pattern», in «International Conference on Web Information Systems and Technologies», vol. 2, p. 127–130, SCITEPRESS.

HORSTMANN, C. e NECAISE, R. D. (2014), *Concetti di informatica e fondamenti di Python*, Maggioli Editore.

LUCCI, S., MUSA, S. M. e KOPEC, D. (2022), «Artificial intelligence in the 21st century», .

MARCHI, L., MANCINI, D. e OTHERS (2009), «Gestione informatica dei dati aziendali», .

MONTANGERO, C. (2009), «Ingegneria del software A», .

RUSSO, S. (2014), «Introduzione all'Ingegneria del Software», .

SWEIGART, A. (2021), *Python oltre le basi: Programmare con stile*, HOEPLI EDITORE.

TAGLIATI, L. V. (2003), *UML e ingegneria del software: dalla teoria alla pratica*, Tecniche nuove.

THOMPSON, L. (2012), *PHP 6: Guida per lo sviluppatore*, HOEPLI EDITORE.

Sitografia

- Github: Where the world builds software – `www.github.com`
- Learn Univpm – `www.learn.univpm.it`
- Python documentation – `www.docs.python.org`
- Python GUIs – `www.pythonguis.com`
- QT – `www.qt.io`
- Stack Overflow –`www.stackoverflow.com`
- Wikipedia – `www.wikipedia.org`

Ringraziamenti

Il primo ringraziamento va ai miei genitori: a mia madre, per aver creduto in me dal momento esatto in cui sono nato, e a mio padre, per essere la sorgente da cui attingere forza quando ne ho bisogno.

Ringrazio Claudio e Francesco, per essere stati la mia bussola e per avermi fatto sentire sempre parte di una squadra, anche al di fuori di un palazzetto.

Ci tengo anche a ringraziare il Professor Ursino, che pazientemente mi ha guidato per la stesura di questa tesi con la massima disponibilità.

Infine un ringraziamento va ai miei amici, per aver reso questi anni più leggeri, per ogni parola di conforto e per ogni risata che mi avete regalato.