DOCUMENTO DI PROGETTAZIONE

Gruppo 16:
- Iannaccone Dana (matr.0612709501)
- Nigro Laura (matr.0612709525)
- Purcaro Andrea (matr.0612709382)
- Zamosteanu Ionela (matr.0612709590)

1. Finalità del documento

Dopo aver provveduto a ragionare a tutto tondo sui requisiti afferenti alla nostra richiesta progettuale, presentiamo, ora, un nuovo dossier, finalizzato alla fase di design.

In primis, motivata la scelta architetturale del sistema, procederemo ad una scomposizione in moduli dello stesso, arrivando a realizzare un diagramma dei package identificati.

Costruiremo, poi, i diagrammi delle classi coinvolte nella nostra applicazione e delle quali implementeremo uno scheletro. Illustreremo, ancora, dei diagrammi di sequenza per le interazioni maggiormente significative, oltre ad una serie di ulteriori immagini opportunamente commentate e motivate. Tenteremo di fare luce su tutte le scelte effettuate, puntando a rientrare nei vincoli previsti. Infine, rilasceremo anche la documentazione delle interfacce pubbliche, servendoci di Doxygen.

In particolare, al fine di visionare al meglio proprio questo aspetto, è consigliato recarsi, mediante https://github.com/purcaroandrea/g16_projectIS.git, presso quanto pubblicato sul nostro GitHub, dove risulteranno consultabili le classi annotate e il Doxyfile associato.

2. Architettura del sistema

2.1. Scelta architetturale

Il nostro sistema si articolerà su una tripartizione logico-funzionale che provveda a curare separatamente gli aspetti legati alla gestione dei dati, alla presentazione visiva e al loro coordinamento. Questa scelta è in linea con quanto dichiarato nel nostro documento SRS, in particolare in relazione agli obiettivi perseguiti di scalabilità e manutenibilità, oltre che al già dichiarato intento di adottare un'architettura MVC (Model-View-Controller; si vedano FC-2, FC-3 e FC-4).

Nella fattispecie, valutiamo sinteticamente quanto offertoci da ciascun livello:

- il Model, in quanto gestore dei dati fondamentali in gioco e della rispettiva memorizzazione, si fa carico delle classi di dominio dell'applicazione (quindi Libro, Studente e Prestito); inoltre, ottempera alla necessità di caricare e salvare su file l'archivio della biblioteca; gestisce, pertanto, classi di servizio, come ArchivioLibri, ArchivioStudenti e ArchivioPrestiti. Vengono soddisfatti, dunque, i requisiti relativi alle esigenze di dati e informazioni (DF-1, DF-2, DF-3) e alla persistenza del sistema (IF-15, FC-6). Si occupa, quindi, dello stato applicativo e delle regole di business. Questo livello è

indipendente dall'interfaccia grafica e può essere riutilizzato o esteso senza impattare sugli altri strati.

- la View, che si compone delle varie schermate (in JavaFX) riferite alla gestione di libri, studenti e prestiti, si rapporta alle azioni interattive propugnate dal bibliotecario, consentendo di visualizzare a schermo le conseguenze delle operazioni da questi effettuate. Vengono espletati, dunque, i requisiti confacentisi all'interfaccia utente (UI-1, …, UI-9), con la risultanza di semplicità e intuitività della medesima.
- il Controller (qui abbiamo ControllerLibri, ControllerStudenti e ControllerPrestiti), invece, come anticipato, media tra View e Model: esso riceve degli eventi dalla prima, li traduce in chiamate al secondo e applica regole di dominio, come controlli di validazione e aggiornamenti consequenziali; la View sarà poi aggiornata in base all'esito delle operazioni. Si ha una stretta relazione tra questo livello e i business flow (BF-1, BF-2, BF-3), che abbiamo definito nel documento di specifica dei requisiti.

Questa struttura consente, in definitiva, di apportare modifiche all'interfaccia utente, senza temere stravolgimenti imprevisti sui dati inseriti e sulla logica di business; ammette, poi, l'estensione del dominio applicativo, con la possibilità di intervenire sul modello e sui dati da esso configurati; centralizza le funzionalità di caricamento e salvataggio dell'archivio, garantendo persistenza e assenza di rischio di inconsistenze; assicura maggiore testabilità per le classi di Model e Controller, oltre che possibili sostituzioni della View, come per un eventuale passaggio futuro ad un'interfaccia web.

2.2. Descrizione dei moduli principali

I moduli, altresì definiti come package, costituiscono la struttura portante dell'architettura del sistema e rappresentano il primo livello di organizzazione logica dell'applicazione. La loro suddivisione non è casuale, ma risponde alla necessità di garantire la separazione delle responsabilità prevista dal pattern architetturale MVC, adottato come riferimento per l'intero progetto. Coerentemente con questo modello, ogni package è progettato per svolgere un insieme ben definito di compiti e per collaborare con gli altri attraverso interfacce chiare e controllate, evitando sovrapposizioni di ruoli o dipendenze indesiderate.

L'obiettivo di una scomposizione modulare efficace è quello di ottenere alta coesione e basso accoppiamento, ossia che, rispettivamente, all'interno di uno stesso modulo ci sia un forte legame tra ciò che vi è incluso, ma si abbia una minima interdipendenza tra i diversi package.

Forti di un approccio pienamente orientato agli oggetti, possiamo dunque analizzare nel dettaglio la struttura dei package e le responsabilità che ciascuno di essi assume all'interno dell'architettura complessiva.

2.2.1. Package biblioteca

Rappresenta l'entry-point dell'intera applicazione, ergendosi a modulo atipico; infatti, a differenza degli altri package, che si focalizzano su specifiche aree funzionali, esso si pone come livello di coordinamento generale dell'intero sistema, governando il flusso comprensivo

dell'applicazione e garantendo che vi si possa navigare tra le diverse sezioni in maniera coerente e lineare.
Al suo interno sono presenti due componenti fondamentali:

- **Main**, che avvia l'applicazione, inizializza l'interfaccia grafica e prepara le strutture principali necessarie al funzionamento del sistema;
- **BibliotecaInterfaccia1Controller**, ovvero il controller principale dell'interfaccia, responsabile della gestione delle schermate, della ricezione degli input dell'utente e dell'instradamento delle richieste verso i controller specifici dei sottosistemi (libri, studenti, prestiti).

Questo package non contiene logica di dominio, né operazioni legate alla gestione dei dati: il suo ruolo è puramente di orchestrazione. È qui che viene coordinata la comunicazione tra l'interfaccia grafica e i moduli applicativi, assicurando che ogni richiesta dell'utente venga indirizzata al componente corretto e che la transizione tra le diverse funzionalità avvenga in modo fluido.

2.2.2. Package view

Il package view costituisce il livello di presentazione dell'applicazione e raccoglie tutte le interfacce grafiche rivolte all'utente. È suddiviso in tre sottopackage (view.libri, view.studenti e view.prestiti), ciascuno dedicato a una specifica area funzionale del sistema.
Le classi contenute in questo modulo si occupano esclusivamente di mostrare i dati e raccogliere gli input dell'utente, delegando ogni operazione applicativa ai rispettivi controller. In accordo con il pattern MVC, le View non contengono logica di dominio, né accedono direttamente ai modelli: fungono da ponte tra l'utente e i controller, garantendo un'interazione chiara e un basso accoppiamento con il resto del sistema.

2.2.3. Package gestione.libri

Il package gestione.libri raccoglie le componenti dedicate alla gestione del catalogo bibliografico. Include le classi di dominio (Libro) e l'archivio corrispondente (ArchivioLibri), responsabile della memorizzazione, ricerca e organizzazione dei volumi. Il controller (LibriController) coordina le operazioni richieste dall'interfaccia, applicando le regole di business, come il controllo dell'unicità dell'ISBN. Il package presenta un'elevata coesione interna e interagisce con gli altri moduli solo quando necessario, mantenendo un basso accoppiamento.

2.2.4. Package gestione.studenti

Il package gestione.studenti gestisce l'anagrafica degli studenti registrati nel sistema. Comprende la classe di dominio (Studente) e l'archivio dedicato (ArchivioStudenti), che offre funzionalità di inserimento, modifica, rimozione e ricerca. Il controller (StudentiController) si occupa della validazione dei dati e dell'univocità della matricola, fungendo da intermediario tra View e Model. Il package è altamente coeso e mantiene dipendenze minime verso gli

altri moduli, limitate ai casi in cui è necessario verificare i prestiti associati a uno studente.

2.2.5. Package gestione.prestiti

Il package gestione.prestiti rappresenta il nucleo della logica applicativa relativa ai prestiti. Contiene la classe Prestito, che modella una singola operazione di prestito, e ArchivioPrestiti, che gestisce la collezione dei prestiti attivi e storici, applicando i vincoli di dominio (come il limite di tre prestiti attivi per studente). Il controller (PrestitiController) coordina le operazioni di prestito e restituzione, interagendo con i moduli dei libri e degli studenti. Pur essendo il package più interconnesso, mantiene un accoppiamento controllato e una coesione molto elevata.

2.2.6. Package persistence

Questo modulo è, chiaramente, dedicato alla gestione della persistenza dei dati dell'applicazione. Include componenti come ArchivioRepository, responsabile del salvataggio e del caricamento delle collezioni, e StatoBiblioteca, che rappresenta lo stato complessivo del sistema. Questo modulo è indipendente dalla logica applicativa e interagisce principalmente con i controller, garantendo un basso accoppiamento e permettendo di sostituire o estendere le modalità di persistenza senza incidere sugli altri package.

2.3. Responsabilità e dipendenza dei package

Modulo / package	Responsabilità principali	Dipendenze in uscita (usa...)	Dipendenze in ingresso (usato da...)
biblioteca	Entry-point dell'applicazione; avvio del sistema; creazione e configurazione dei componenti principali; collegamento tra controller, view e layer di gestione/persistence.	gestione.libri.controller, gestione.studenti.controller, gestione.prestiti.controller	Nessuno (entry-point)
gestione.libri.model	Classi dominio libri: Libro, ArchivioLibri; rappresentazione dei dati e regole base sull'entità libro.	Nessuna (dipende solo da Java standard)	gestione.libri.controller, persistence (tramite ArchivioRepository / StatoBiblioteca) gestione.prestiti

gestione.libri.controller	Coordinamento delle operazioni sui libri; mediazione tra view dei libri e modello (Libro, ArchivioLibri).	gestione.libri.model, persistence	biblioteca, view.libri,
gestione.studenti.model	Classi dominio studenti: Studente, ArchivioStudenti; gestione struttura dati e vincoli base (es. unicità matricola).	Nessuna (dipende solo da Java standard)	gestione.studenti.controller, persistence (tramite ArchivioRepository / StatoBiblioteca), gestione.prestiti
gestione.studenti.controller	Coordinamento delle operazioni sugli studenti; mediazione tra view studenti e modello (Studente, ArchivioStudenti).	gestione.studenti.model, persistence	biblioteca, view.studenti
gestione.prestiti.model	Classi dominio prestiti: Prestito, ArchivioPrestiti; rappresentazione dei prestiti attivi e storici e regole di base sull'entità prestito.	Nessuna (dipende solo da Java standard)	gestione.prestiti.controller, persistence (tramite ArchivioRepository / StatoBiblioteca), gestione.prestiti
gestione.prestiti.controller	Gestione del ciclo di vita dei prestiti; applicazione dei vincoli (max prestiti, disponibilità copie); coordinamento tra prestiti, libri e studenti.	gestione.prestiti.model, gestione.libri.model, gestione.studenti.model, persistence	biblioteca, view.prestiti
view.libri	Interfacce grafiche per la gestione dei libri; raccolta input utente e inoltro delle richieste al controller dei libri.	gestione.libri.controller	Nessuno

view.studenti	Interfacce grafiche per la gestione degli studenti; raccolta input e inoltro al controller studenti.	gestione.studenti.controller	Nessuno
view.prestiti	Interfacce grafiche per la gestione dei prestiti; dialogo con il controller prestiti e lettura dati di libri e studenti.	gestione.prestiti.controller	Nessuno
persistence	Gestione salvataggio/caricamento degli archivi tramite ArchivioRepository e stato complessivo della biblioteca tramite StatoBiblioteca.	gestione.libri.model, gestione.studenti.model, gestione.prestiti.model	gestione.libri.controller, gestione.studenti.controller, gestione.prestiti.controller

2.4. Diagramma dei package

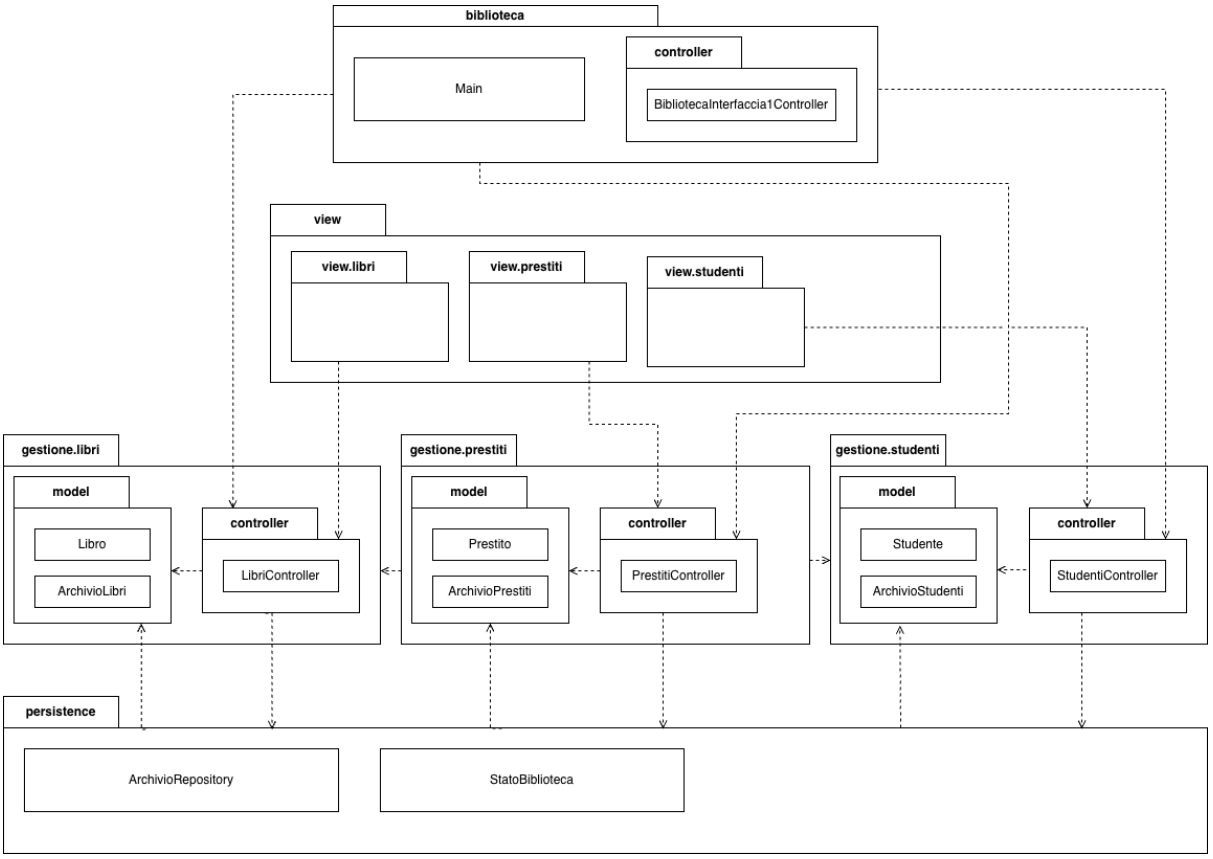
Archiviamo questa sezione con un'illustrazione che, in modo sintetico, restituisca una visione d'insieme circa le connessioni tra i moduli principali del sistema. Il diagramma dei package evidenzia, difatti, le dipendenze architetturali attraverso frecce direzionali che indicano il verso dell'utilizzo: ogni freccia parte dal package che dipende e punta verso quello utilizzato.

Nella parte inferiore dell'immagine che segue, è collocato il package **persistence**, che viene utilizzato da tutti i controller applicativi e, al contempo, si serve dei model associati (ciò si ha per i package di gestione). La sua posizione riflette il ruolo trasversale e passivo nella gestione dello stato del sistema. Sul versante opposto, in seconda linea, troviamo il package view, suddiviso in tre sottosezioni (view.libri, view.studenti, view.prestiti), ciascuna delle quali dipende esclusivamente dal modulo di gestione corrispondente e, nello specifico, dal relativo controller.

Al centro della struttura si colloca il package gestione.prestiti, che assume un ruolo di snodo: esso dipende sia da gestione.libri che da gestione.studenti, in quantos ogni operazione di prestito coinvolge inevitabilmente un libro e uno studente. I tre moduli di gestione sono a loro volta utilizzati dal package biblioteca, che funge da orchestratore generale e coordina l'interazione tra interfaccia e logica applicativa.

Nel complesso, il diagramma conferma la presenza di un flusso architetturale ben definito, privo di dipendenze circolari e coerente con i principi di

separazione dei ruoli. Non si evidenziano interazioni dirette tra persistence e view, a conferma della corretta applicazione del pattern MVC e della modularità del sistema.



3. Modello statico

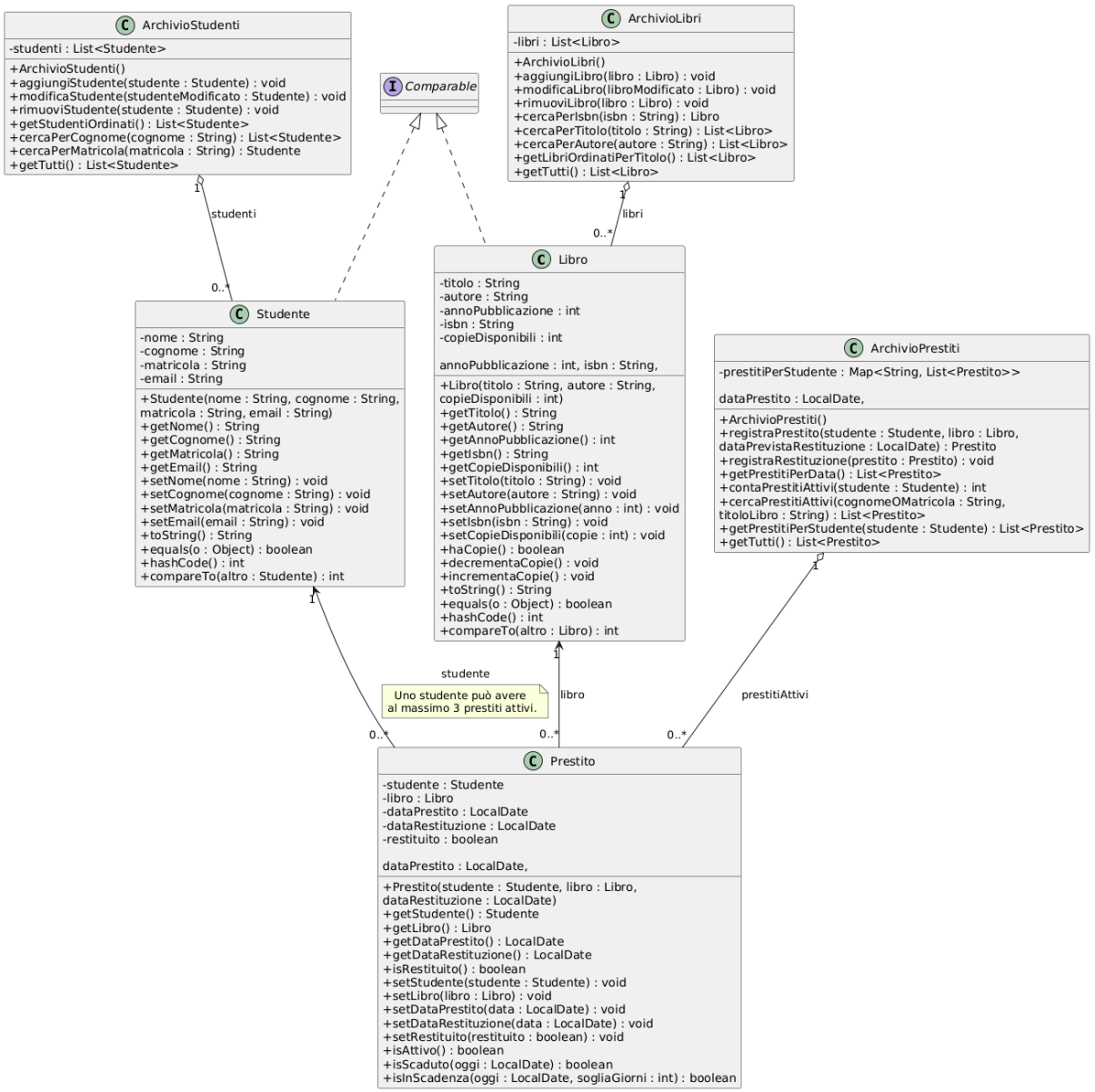
3.1. Panoramica e diagramma delle classi

Il modello statico del sistema rappresenta la struttura concettuale e organizzativa dell'applicazione di gestione della biblioteca universitaria, descrivendo in modo formale le entità principali, i loro attributi, il comportamento pubblico esposto e le relazioni che intercorrono tra di esse.

Il diagramma delle classi che segue, a tal proposito, fornisce una visione d'insieme dell'intero dominio applicativo, evidenziando come i concetti fondamentali (Studente, Libro e Prestito) siano modellati e gestiti attraverso gli archivi dedicati (ArchivioStudenti, ArchivioLibri, ArchivioPrestiti).

L'obiettivo del modello statico è garantire un'illustrazione chiara, coerente e facilmente estendibile dell'intero sistema, assicurando un elevato livello di coesione interna alle classi e un basso accoppiamento tra i diversi componenti.

Il diagramma evidenzia, inoltre, alcuni vincoli di business fondamentali presenti nei requisiti, come il limite massimo di tre prestiti attivi per studente e il controllo delle copie disponibili per ciascun libro.



3.2. Descrizione delle principali classi

Le classi descritte di seguito costituiscono il nucleo del modello di dominio del sistema. Esse rappresentano gli elementi fondamentali per la gestione dei libri, degli studenti e dei prestiti, e definiscono le strutture dati e le operazioni che supportano i casi d'uso descritti nel documento SRS.

3.2.1. Classe Libro

Responsabilità

Rappresenta un volume presente nel catalogo della biblioteca e racchiude tutte le informazioni necessarie alla sua identificazione e disponibilità.

È l'unità informativa su cui si basano le operazioni di inserimento, modifica, ricerca e prestito dei libri.

Attributi principali

- titolo : String
- autore : String
- annoPubblicazione : int
- isbn : String

- copieDisponibili : int

Metodi pubblici principali

- Costruttore con tutti i campi obbligatori.
- Metodi getter/setter per ogni attributo.
- boolean haCopie() – indica se sono ancora presenti, in biblioteca, copie prestabili.
- void decrementaCopie() / void incrementaCopie() – aggiornano il numero di copie disponibili, a seguito di prestiti e restituzioni.
- Metodi standard toString(), equals(Object) e hashCode().
- Implementazione dell'interfaccia Comparable<Libro> tramite compareTo(Libro altro), che definisce l'ordinamento naturale (per titolo) utilizzato da ArchivioLibri per produrre la lista ordinata, come descritto nel documento SRS.

Relazioni

- La classe Libro è coinvolta in un'associazione unidirezionale molti-a-uno proveniente da Prestito: ogni prestito fa riferimento a un singolo libro, mentre un libro può comparire in più prestiti nel tempo, compatibilmente con il numero di copie disponibili.
- Gli oggetti della classe Libro sono aggregati in ArchivioLibri, che mantiene la collezione di tutti i volumi registrati nel sistema. L'archivio non è responsabile della creazione né del ciclo di vita dei libri, ma ne gestisce esclusivamente la memorizzazione e la ricerca.

3.2.2. Classe Studente

Responsabilità

Modella uno studente registrato nel sistema, cioè un potenziale beneficiario dei servizi della biblioteca. Raccoglie i dati anagrafici e identificativi utilizzati nelle operazioni di ricerca e nella tracciabilità dei prestiti.

Attributi principali

- nome : String
- cognome : String
- matricola : String
- email : String

Metodi pubblici principali

- Costruttore che inizializza tutti gli attributi obbligatori.
- Metodi *getter/setter* per ogni attributo.
- Metodo standard toString().
- Metodi equals(Object) e hashCode() basati sulla matricola.
- Implementazione di Comparable<Studente> (mediante compareTo(Studente altro)), con ordinamento per cognome e nome, utile per la visualizzazione della lista studenti ordinata.

Relazioni

- La classe Studente partecipa ad un'associazione unidirezionale molti-a-uno proveniente da Prestito: ogni prestito è riferito a un singolo studente, mentre uno studente può risultare associato a più prestiti attivi nel tempo, nel rispetto del vincolo di dominio che limita a tre il numero massimo di prestiti contemporaneamente attivi.
- Gli oggetti della classe Studente sono aggregati in ArchivioStudenti, che mantiene la collezione completa degli studenti registrati nel

sistema. L'archivio non gestisce la creazione né il ciclo di vita degli studenti, ma si occupa esclusivamente della loro memorizzazione, ricerca e organizzazione.

3.2.3. Classe Prestito

Responsabilità

Rappresenta una singola operazione di prestito di un libro in favore di uno studente, attiva o conclusa. Contiene le informazioni necessarie alla determinazione delle scadenze, al controllo dei ritardi e all'aggiornamento della disponibilità dei volumi.

Attributi principali

- studente : Studente – destinatario del prestito.
- libro : Libro – volume concesso.
- dataPrestito : LocalDate – data di inizio del prestito.
- dataRestituzione : LocalDate – data prevista per la restituzione.
- restituito : boolean – indica se il prestito è stato chiuso.

Metodi pubblici principali

- Costruttore che inizializza tutti gli attributi.
- Metodi getter/setter per tutti gli attributi.
- boolean isAttivo() – restituisce vero se il prestito non è ancora restituito.
- boolean isScaduto(LocalDate oggi) – verifica se la data prevista di restituzione è già trascorsa.
- boolean isInScadenza(LocalDate oggi, int sogliaGiorni) – consente di individuare prestiti prossimi alla scadenza, come richiesto per l'evidenziazione dei prestiti in ritardo o imminenti.

Relazioni

- La classe Prestito è il punto di origine di due associazioni unidirezionali molti-a-uno: ogni oggetto Prestito fa riferimento a un singolo oggetto della classe *Studente* e a un singolo oggetto della classe *Libro*.
- Gli oggetti della classe Prestito sono aggregati all'interno di ArchivioPrestiti, che gestisce la collezione completa dei prestiti registrati nel sistema. L'archivio non controlla la creazione né il ciclo di vita dei prestiti, ma ne cura esclusivamente la memorizzazione, l'organizzazione e le operazioni di ricerca e aggiornamento.

3.2.4. Classe ArchivioLibri

Responsabilità

Gestisce l'insieme dei libri presenti nel sistema: inserimento, modifica, cancellazione, ricerca e ordinamento ne sono le operazioni relative. È l'unico punto del modello autorizzato a manipolare direttamente la lista di Libro, garantendo l'integrità dei dati e l'unicità dell'ISBN.

Attributi principali

- libri : List<Libro> – collezione dei libri registrati.

Metodi pubblici principali

- ArchivioLibri() – inizializza la lista interna.
- void aggiungiLibro(Libro libro) – inserisce un nuovo libro dopo i controlli sull'ISBN.

- void modificaLibro(Libro libroModificato) – aggiorna i dati di un libro esistente.
- void rimuoviLibro(Libro libro) – rimuove un libro dal catalogo.
- Libro cercaPerIsbn(String isbn) – ricerca un libro tramite il codice identificativo.
- List<Libro> cercaPerTitolo(String Titolo) – ricerca un libro in base ad un titolo passato come parametro.
- List<Libro> cercaPerAutore(StringAutore) – ricerca un libro in base ad un autore passato come parametro.
- List<Libro> getLibriOrdinatiPerTitolo() – restituisce la lista ordinata, sfruttando Comparable<Libro>.
- List<Libro> getTutti() – restituisce tutti i libri presenti.

Relazioni

- La classe ArchivioLibri aggrega oggetti della classe Libro, mantenendo la collezione completa dei volumi registrati nel sistema. L'archivio non è responsabile della creazione né del ciclo di vita dei libri, ma ne gestisce esclusivamente la memorizzazione, l'organizzazione e le operazioni di ricerca, fungendo da contenitore logico della loro presenza nel sistema.

3.2.5. Classe ArchivioStudenti

Responsabilità

Si occupa della gestione dell'elenco degli studenti, traducentesi in inserimento, modifica, rimozione e ricerca per cognome o matricola, oltre alla possibilità di visualizzare la lista ordinata.

Attributi principali

- studenti : List<Studente> – collezione degli studenti registrati.

Metodi pubblici principali

- ArchivioStudenti() – inizializza la lista interna.
- void aggiungiStudente(Studente studente) – inserisce un nuovo studente, controllando l'univocità della matricola.
- void modificaStudente(Studente studenteModificato) – aggiorna i dati di uno studente.
- void rimuoviStudente(Studente studente) – elimina uno studente dal sistema.
- List<Studente> getStudentiOrdinati – restituisce la lista ordinata sfruttando Comparable<Studente>.
- List<Studente> cercaPerCognome(String cognome) – ricerca per cognome.
- Studente cercaPerMatricola(String matricola) – ricerca puntuale per matricola.

Relazioni

- La classe ArchivioStudenti aggrega oggetti della classe Studente, mantenendo la collezione completa degli studenti registrati nel sistema. L'archivio non è responsabile della creazione né del ciclo di vita degli studenti, ma ne gestisce esclusivamente la memorizzazione, l'organizzazione e le operazioni di ricerca, fungendo da contenitore logico della loro presenza nel sistema.

3.2.6. Classe ArchivioPrestiti

Responsabilità

È il componente centrale per la gestione dei prestiti. Mantiene l'elenco dei prestiti attivi e passati, applica i vincoli di business, in particolare il limite di tre prestiti attivi per studente e il controllo delle copie disponibili, e fornisce viste ordinate e filtri di ricerca utilizzati dall'interfaccia.

Attributi principali

- prestitiPerStudente : Map<String, List<Prestito>> – indicizza i prestiti per matricola dello studente, facilitando il conteggio e il recupero rapido dei prestiti attivi.

Metodi pubblici principali

- ArchivioPrestiti() – inizializza le strutture interne.
- Prestito registraPrestito(Studente studente, Libro libro, LocalDate dataRestituzione)
    - crea un nuovo prestito,
    - verifica che lo studente non abbia superato il limite di tre prestiti,
    - controlla la disponibilità del libro e, in caso positivo, decrementa le copie disponibili.
- void registraRestituzione(Prestito prestito) – segna il prestito come restituito, aggiorna la disponibilità del libro e rimuove il prestito dall'elenco degli attivi.
- List<Prestito> getPrestitiPerData() – restituisce i prestiti ancora attivi ordinati per data prevista di restituzione.
- int contaPrestitiAttivi(Studente studente) – restituisce il numero di prestiti attivi per uno studente passato come parametro.
- List<Prestito> cercaPrestitiAttivi(String cognome, String matricola, String titoloLibro) – ricerca combinata per studente e libro.
- List<Prestito> getPrestitiPerStudente(Studente studente) – restituisce tutti i prestiti di uno studente, utile anche per controllare il numero di prestiti attivi.
- List<Prestito> getTutti() – restituisce tutti i prestiti attivi.

Relazioni

- La classe ArchivioPrestiti aggrega oggetti della classe Prestito, mantenendo la collezione completa dei prestiti registrati nel sistema. L'archivio non è responsabile della creazione né del ciclo di vita dei prestiti, ma ne gestisce esclusivamente la memorizzazione, l'organizzazione e le operazioni di ricerca e aggiornamento, fungendo da contenitore logico delle informazioni relative ai prestiti attivi e storici.

3.3. Scelte progettuali: coesione, accoppiamento e principi adottati

Il modello statico adottato per l'applicazione riflette una serie di scelte progettuali orientate alla chiarezza strutturale, alla manutenibilità del codice e alla corretta separazione delle responsabilità. Le classi principali (Libro, Studente e Prestito) rappresentano entità del dominio con caratteristiche e comportamenti ben definiti, mentre gli archivi dedicati (ArchivioLibri, ArchivioStudenti e ArchivioPrestiti) svolgono il ruolo di contenitori logici e gestori delle rispettive collezioni. Questa distinzione

netta tra dati e gestione dei dati costituisce uno degli elementi cardine dell'intero modello.

Dal punto di vista della **coesione**, ogni classe presenta un insieme di responsabilità omogenee e chiaramente circoscritte. La classe Libro si occupa esclusivamente della rappresentazione dei dati bibliografici e della gestione delle copie disponibili; Studente modella i dati anagrafici e identificativi senza inglobare logiche relative ai prestiti; Prestito incapsula la relazione tra studente e libro, insieme alle informazioni temporali e allo stato dell'operazione. Gli archivi, a loro volta, mantengono un'elevata coesione interna, poiché si limitano alla memorizzazione, ricerca e organizzazione delle rispettive entità, senza assumere responsabilità estranee al loro ruolo.

Per quanto riguarda l'**accoppiamento**, il sistema è progettato per mantenere dipendenze ridotte e ben controllate. Le associazioni tra le classi sono unidirezionali: un oggetto Prestito conosce lo studente e il libro coinvolti, ma né Studente né Libro mantengono riferimenti ai prestiti. Questa scelta evita dipendenze circolari e riduce il rischio di inconsistenze, oltre a semplificare la gestione del ciclo di vita degli oggetti. Analogamente, gli archivi non dipendono gli uni dagli altri e non condividono strutture dati interne: ciascuno opera in modo autonomo sulla propria collezione, contribuendo a mantenere un basso livello di accoppiamento tra i componenti del sistema.

Le scelte progettuali adottate rispettano, inoltre, diversi **principi di buona progettazione orientata agli oggetti**. Il **Single Responsibility Principle (SRP)** è applicato in modo rigoroso: ogni classe svolge un unico compito ben definito, evitando sovrapposizioni di responsabilità. Il **principio dell'incapsulamento** è rispettato grazie alla protezione degli attributi e all'accesso controllato tramite metodi getter e setter. L'uso dell'interfaccia Comparable<T> per le classi Libro e Studente consente di definire un ordinamento naturale senza introdurre dipendenze aggiuntive, favorendo l'estensibilità del sistema in conformità con l'**Open/Closed Principle (OCP)**. Infine, la scelta di centralizzare la gestione dei prestiti in ArchivioPrestiti evita duplicazioni di informazioni e garantisce una visione coerente dello stato del sistema, in linea con il principio di **Information Hiding**.
Nel complesso, il modello statico risulta quindi **coerente, modulare e facilmente estendibile**, capace di supportare in modo efficace i requisiti funzionali del sistema e di costituire una base solida per le fasi successive di implementazione e manutenzione.

4. Modello dinamico

4.1. Interazioni tra classi

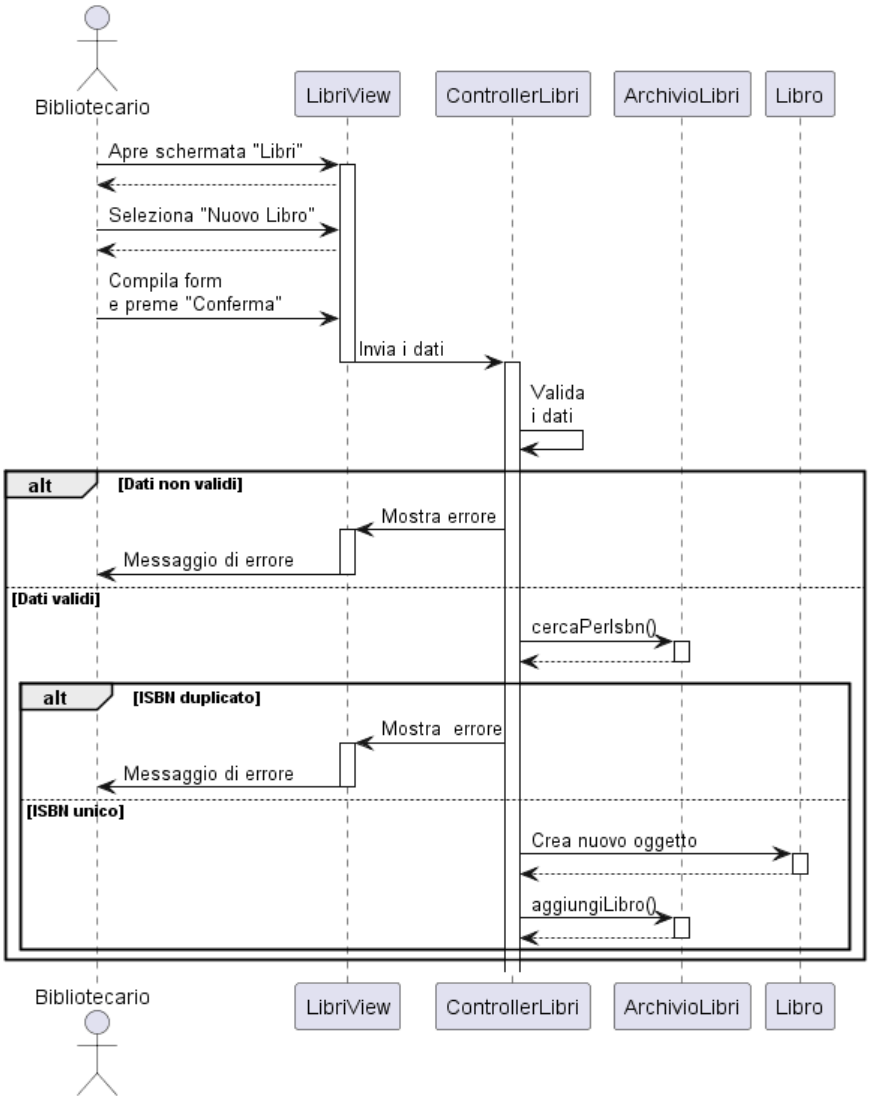
In questa sezione vengono documentate le interazioni tra le classi per i casi d'uso ritenuti più significativi, in modo da mostrare come il modello statico venga effettivamente utilizzato a runtime.
Sono stati modellati, dunque, i seguenti casi d'uso:

- **UC-1: Inserimento di un libro**
- **UC-10: Ricercare uno studente**
- **UC-11: Registrare un prestito**
- **UC-14: Registrare la restituzione**

Per ciascuno di essi viene presentato un **diagramma di sequenza UML** (in PlantUML), congiuntamente ad una breve descrizione delle responsabilità delle classi coinvolte.

UC-1: Inserimento di un libro

Diagramma di sequenza:



Obiettivo: Inserire nel sistema un nuovo libro, specificandone titolo, autore, anno di pubblicazione, ISBN e numero di copie disponibili; aggiornare la lista dei libri.

Classi coinvolte:

- **LibriView**: rappresenta le interfacce con cui il bibliotecario interagisce.
- **ControllerLibri**: coordina l'operazione di registrazione del prestito.
- **ArchivioLibri**: gestisce la collezione di oggetti Libro.
- **Libro**: classe di dominio che rappresenta un singolo oggetto Libro.

Commento al diagramma:

1. Il bibliotecario apre la schermata "Libri" e seleziona "Nuovo Libro".
2. Il bibliotecario compila il form della LibriView e preme il pulsante di conferma.
3. La view invia i dati al ControllerLibri, che si occupa di:

3a. Validare i dati (campi vuoti, anno non numerico o superiore al corrente, ISBN non nel formato corretto).

3a.1 Se le informazioni inserite non sono valide, il controller richiede alla view di mostrare un messaggio di errore.

3b. Verificare che il libro non sia già presente, invocando cercaPerIsbn() su ArchivioLibri.

3b.1 Se l'ISBN è duplicato, il controller richiede alla view di mostrare un messaggio di errore.

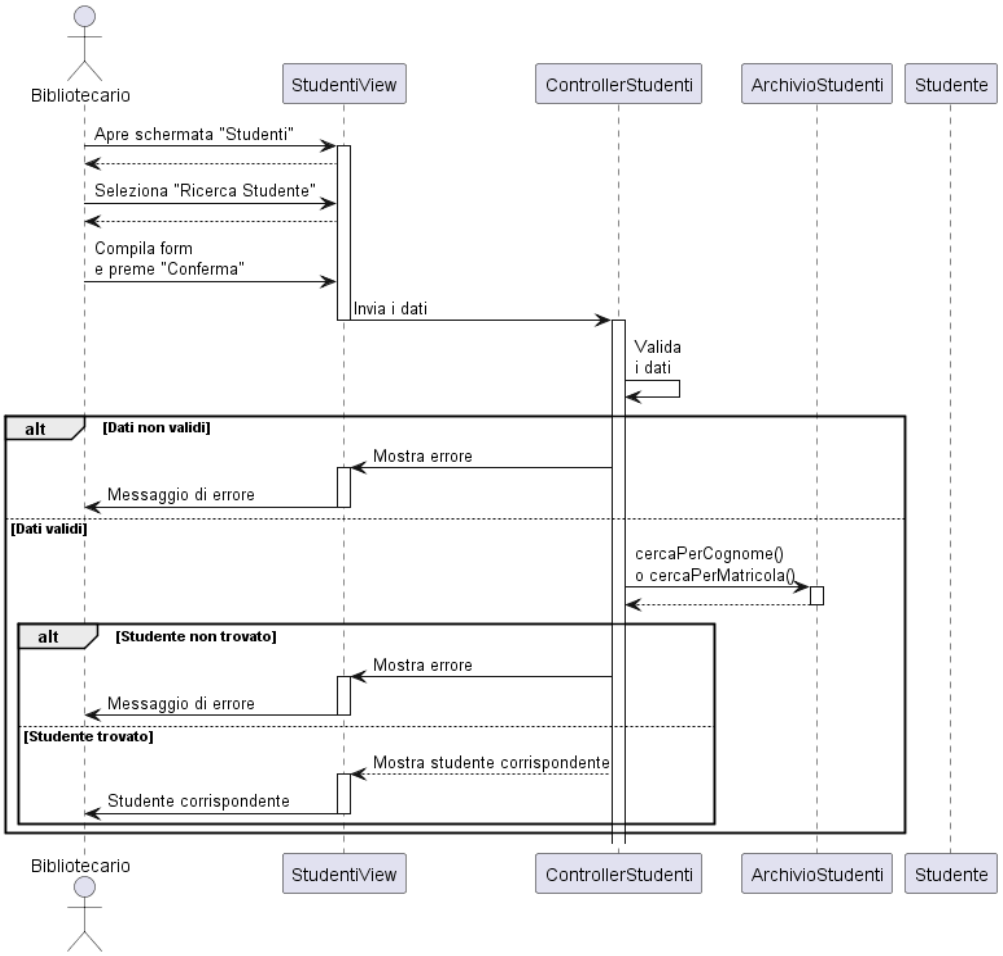
4. Se le verifiche hanno esito positivo, il controller:

4a. Crea un nuovo oggetto Libro.

4b. Aggiorna la lista dei libri, invocando aggiungiLibro() su ArchivioLibri.

UC-10: Ricercare uno studente

Diagramma di sequenza:



Obiettivo: Ricercare uno studente registrato nel sistema, specificandone cognome o matricola.

Classi coinvolte:

- **StudentiView**: rappresenta le interfacce con cui il bibliotecario interagisce.

- **ControllerStudenti**: riceve l'evento di conferma, valida i dati, dialoga con l'archivio.
- **ArchivioStudenti**: gestisce la collezione di oggetti Studente.
- **Studente**: classe di dominio che rappresenta un singolo oggetto Studente.

Commento al diagramma:

1. Il bibliotecario apre la schermata "Studenti" e seleziona "Ricerca Studente".
2. Il bibliotecario compila il form della StudentiView e preme il pulsante di conferma.
3. La view invia i dati al ControllerStudenti, che si occupa di:
    - **3a.** Validare i dati (campi vuoti).
        - **3a.1** Se le informazioni inserite non sono valide, il controller richiede alla view di mostrare un messaggio di errore.
    - **3b.** Verificare che lo studente sia già registrato nel sistema, invocando cercaPerCognome() o cercaPerMatricola() su ArchivioStudenti.
        - **3b.1** Se lo studente non esiste, il controller richiede alla view di mostrare un messaggio di errore.
4. Se le verifiche hanno esito positivo, il controller richiede alla view di mostrare lo studente corrispondente.

UC-11: Registrare un prestito

Obiettivo: Registrare un prestito, specificandone studente interessato, libro e data del prestito, e rispettando i seguenti vincoli:

- Lo studente non ha superato il limite massimo di prestiti attivi (3).
- È disponibile almeno una copia del libro.

Aggiornare le copie disponibili del libro e la lista prestiti attivi dello studente.

Classi coinvolte:

- **PrestitiView**: rappresenta le interfacce con cui il bibliotecario interagisce.
- **PrestitiController**: coordina l'operazione di registrazione del prestito.
- **ArchivioPrestiti**: contiene la collezione di oggetti Prestito.
- **Prestito**, **Libro**, **Studente**: classi di dominio che rappresentano le entità coinvolte nel prestito.

Commento al diagramma:

1. Il bibliotecario apre la schermata "Prestiti" e seleziona "Nuovo Prestito".
2. Il bibliotecario compila il form della PrestitiView e preme il pulsante di conferma.
3. La view invia i dati al ControllerPrestiti, che si occupa di:
    - **3a.** Validare i dati (campi vuoti, data in formato non valido o superiore alla corrente).
        - **3a.1** Se le informazioni inserite non sono valide, il controller richiede alla view di mostrare un messaggio di errore.

3b. Verificare che lo studente non abbia superato il limite massimo di prestiti attivi, invocando contaPrestitiAttivi() su ArchivioPrestiti.

3b.1 Se lo studente ha già 3 prestiti attivi, il controller richiede alla view di mostrare un messaggio di errore.

3c. Se il limite non è superato, il controller verifica che il libro abbia copie disponibili, invocando haCopie() su Libro.

3c.1 Se il libro non è disponibile, il controller richiede alla view di mostrare un messaggio di errore.

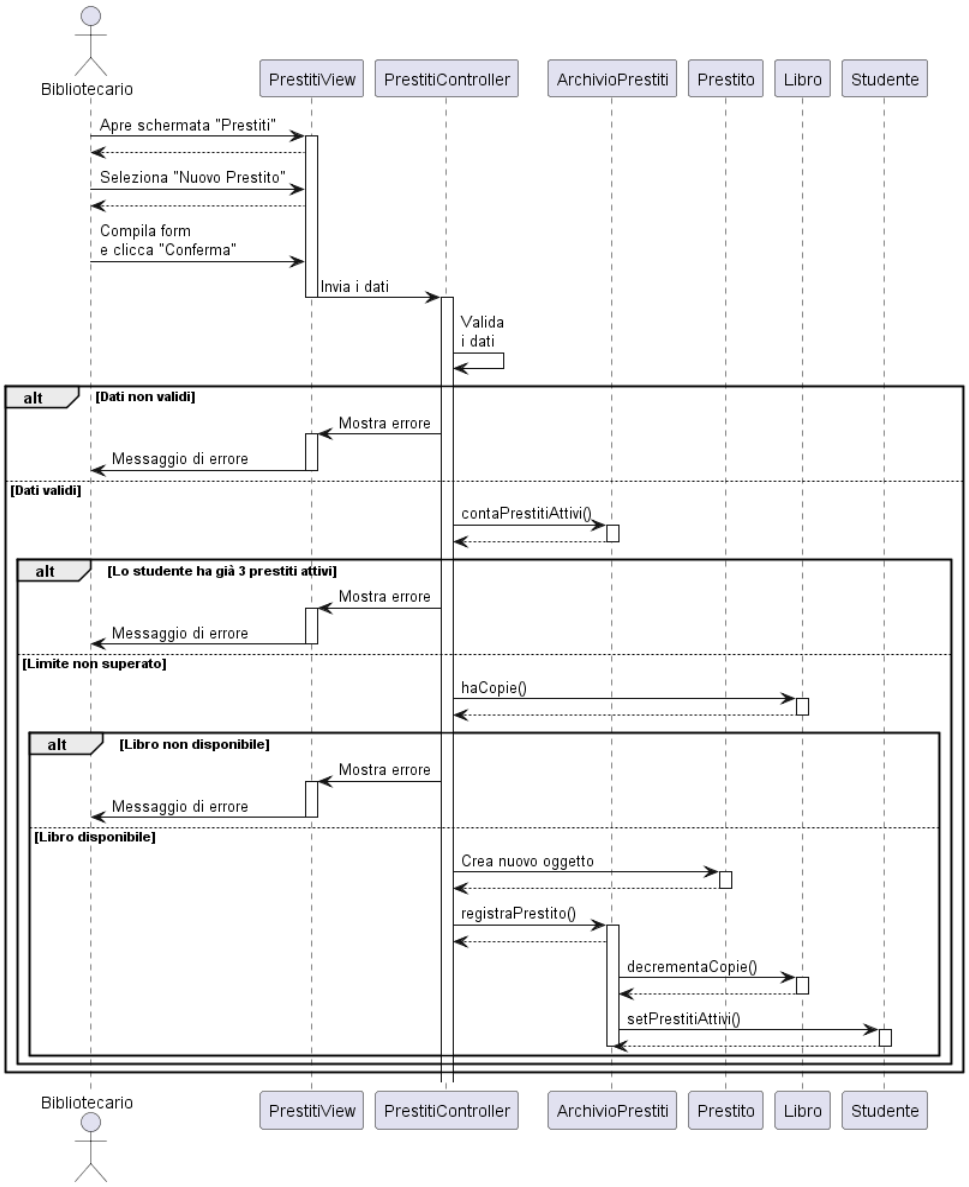
4. Se tutti i vincoli sono rispettati, il controller crea un nuovo oggetto Prestito e invoca registraPrestito() su ArchivioPrestiti.

5. L'ArchivioPrestiti:

5a. Aggiorna il numero di copie disponibili del libro, invocando **decrementaCopie() su Libro.**

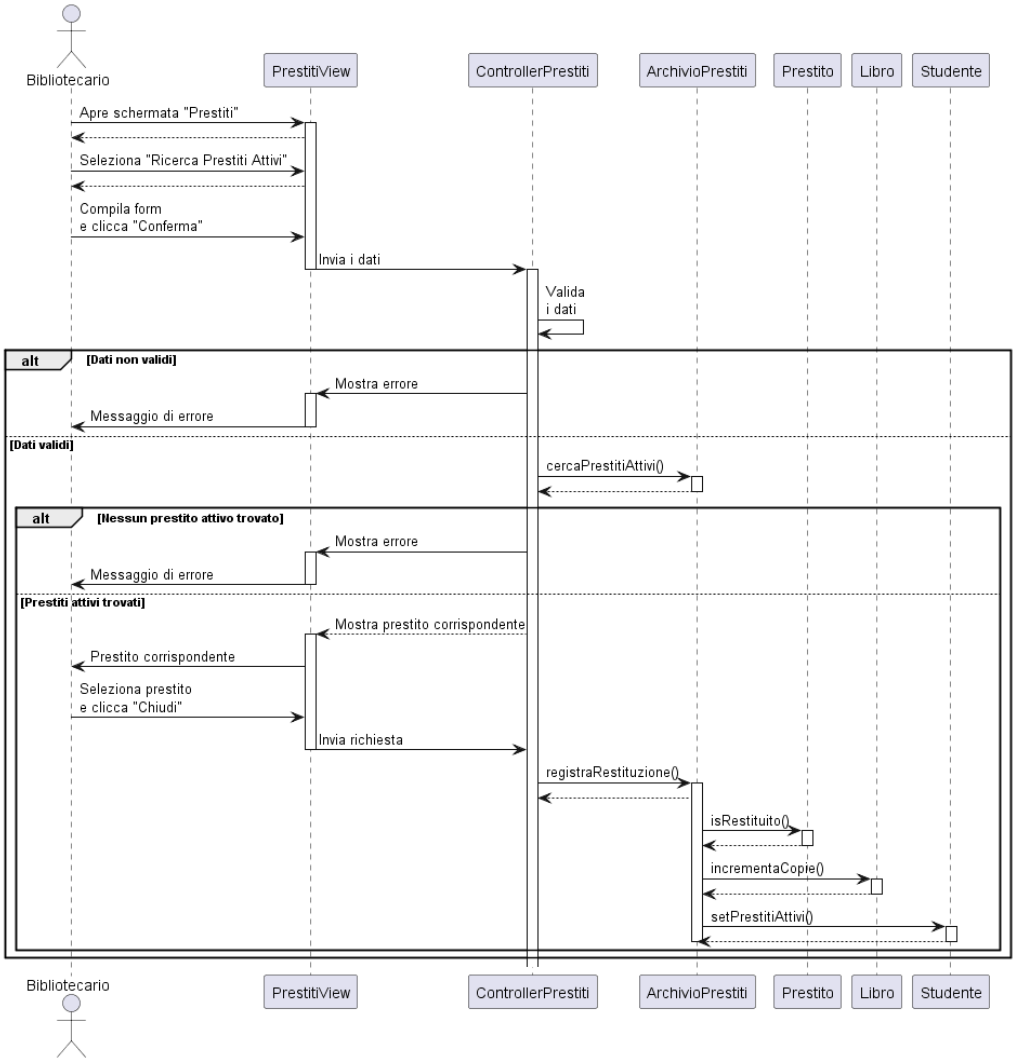
5b. Aggiorna la lista prestiti attivi dello studente, invocando **setPrestitiAttivi() su Studente.**

Diagramma di sequenza:



UC-14: Registrare la restituzione

Diagramma di sequenza:



Obiettivo: Registrare la restituzione di un libro, aggiornare il numero di copie disponibili del libro e la lista dei prestiti attivi dello studente.

Classi coinvolte:

- **PrestitiView**: rappresenta le interfacce con cui il bibliotecario interagisce.
- **ControllerPrestiti**: riceve la richiesta di chiusura e coordina le operazioni.
- **ArchivioPrestiti**: gestisce la collezione di oggetti Prestito.
- **Prestito**, **Libro**, **Studente**: classi di dominio coinvolte nella restituzione.

Commento al diagramma:

1. Il bibliotecario apre la schermata “Prestiti” e seleziona “Ricerca Prestiti Attivi”.
2. Il bibliotecario inserisce il criterio di ricerca del prestito (cognome o matricola dello studente per cui è attivo il prestito e titolo del libro) nel form della PrestitiView e preme il pulsante di conferma.
3. La view invia i dati al ControllerPrestiti che si occupa di:
    3a. Validare i dati (campi vuoti).

3a.1 Se le informazioni inserite non sono valide, il controller richiede alla view di mostrare un messaggio di errore.

3b. Verificare che il prestito risulti nel sistema, invocando cercaPrestitiAttivi() su ArchivioPrestiti.

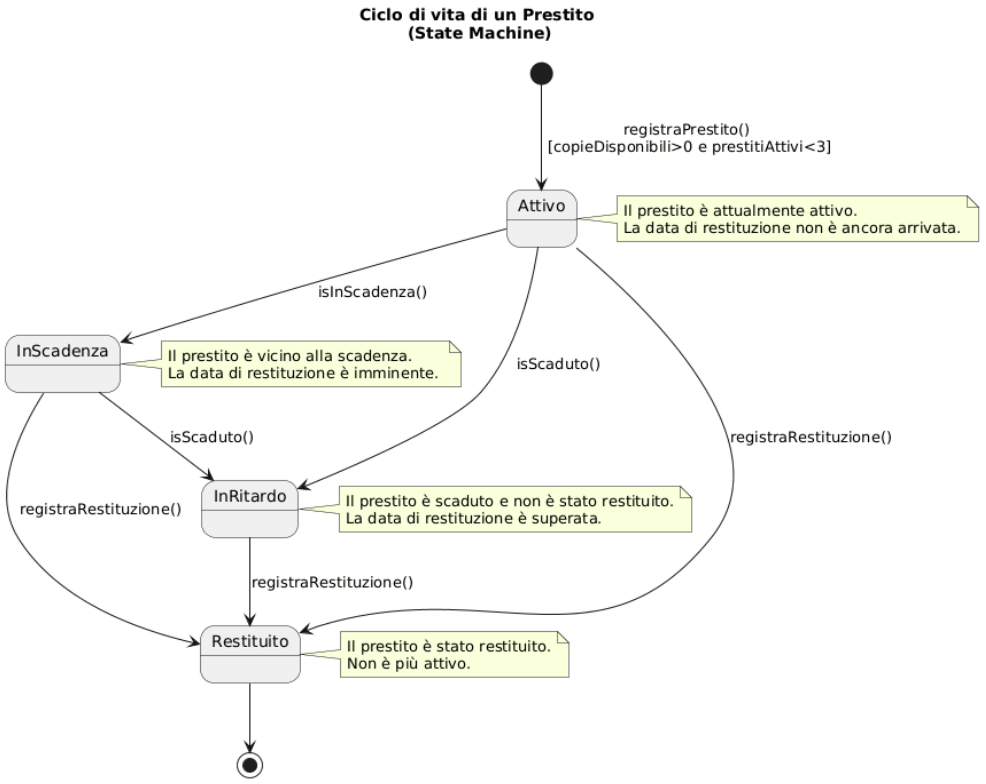
3b.1 In caso contrario, il controller richiede alla view di mostrare un messaggio di errore.

4. Se la verifica ha esito positivo, il controller richiede alla view di mostrare il prestito corrispondente.
5. Il bibliotecario seleziona il prestito dalla PrestitiView e preme il pulsante “Chiudi”.
6. La view invia la richiesta al controller, che registra la restituzione, invocando registraRestituzione() su ArchivioPrestiti.
7. L’ArchivioPrestiti:
   - **7a.** Segna il prestito come restituito tramite isRestituito() su Prestito.
   - **7b.** Aggiorna il numero di copie disponibili del libro tramite incrementaCopie() su Libro.
   - **7c.** Aggiorna la lista prestiti attivi dello studente, invocando setPrestitiAttivi su Studente.

4.2. Macchina a stati del Prestito

Come ulteriore diagramma UML, ci riserviamo di illustrare un diagramma di macchina a stati, nella specificità dello studio dello stato del generico oggetto Prestito. A seguito di peculiari eventi, denominati Trigger, e verificate delle condizioni note come Guardie, assisteremo a delle transizioni di stato.
Gli stati principali che esso può assumere sono:

- **Attivo** – il prestito è stato registrato e non è ancora terminato, ossia ancora non è avvenuta una restituzione.
- **Restituito** – il libro è stato riconsegnato.
- **inScadenza** – il prestito deve essere terminato entro una certa soglia di giorni, altrimenti scatterà l’evidenziazione di un ritardo.
- **inRitardo** – il prestito non è stato terminato entro la data prevista della restituzione del libro, quindi se ne ha una segnalazione.
  Questi ultimi due stati sono da intendersi come proprietà derivate, poiché dipendenti unicamente dalla data di uno specifico giorno corrente e dalla data prevista per la riconsegna del libro. I metodi isScaduto() e isInScadenza(), a tale scopo, sono fortemente esemplificativi.



5. Design dell'interfaccia utente

Ultimiamo il nostro documento di progettazione con la descrizione e l'illustrazione dei mock-up delle schermate principali della nostra interfaccia utente.

Ogni immagine sarà preceduta da una rappresentazione testuale e procederemo iterativamente, circa un ordine logico da noi designato.

Esordiamo con la prima schermata e vediamone i dettagli:

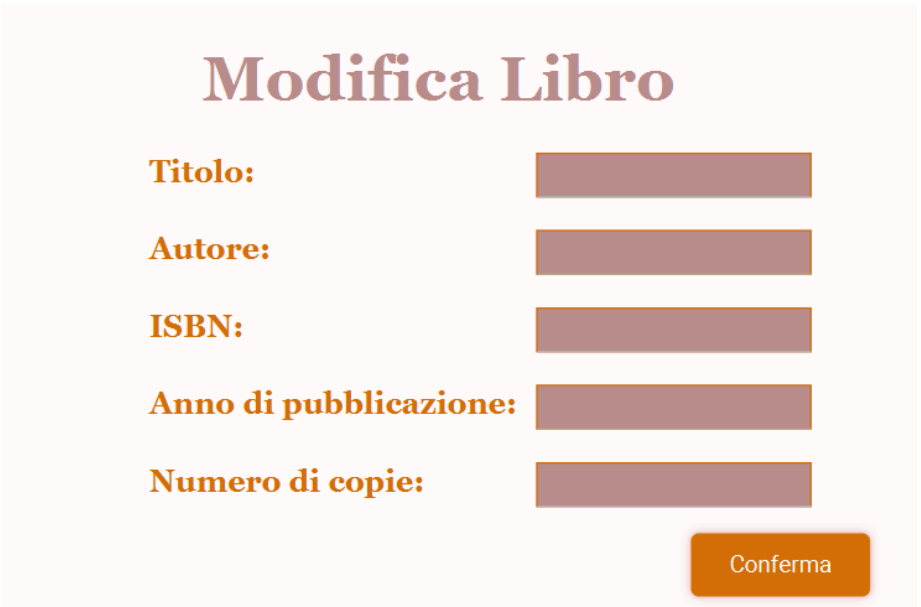
5.1. **Titolo:** "Biblioteca Universitaria", identifica lo scopo dell'applicazione.

Sfondo: Sulla destra è presente una pila di libri, un elemento grafico immediatamente riconoscibile che rafforza il tema della biblioteca e aggiunge un tocco visivo.

Area Centrale: Il cuore dell'interazione è costituito dai tre pulsanti principali, disposti in maniera verticale.

5.2. **Titolo:** “Gestione Libri”, serve ad identificare la sezione.
Pulsanti: Tre pulsanti orizzontali che rappresentano le funzioni di gestione dei libri.

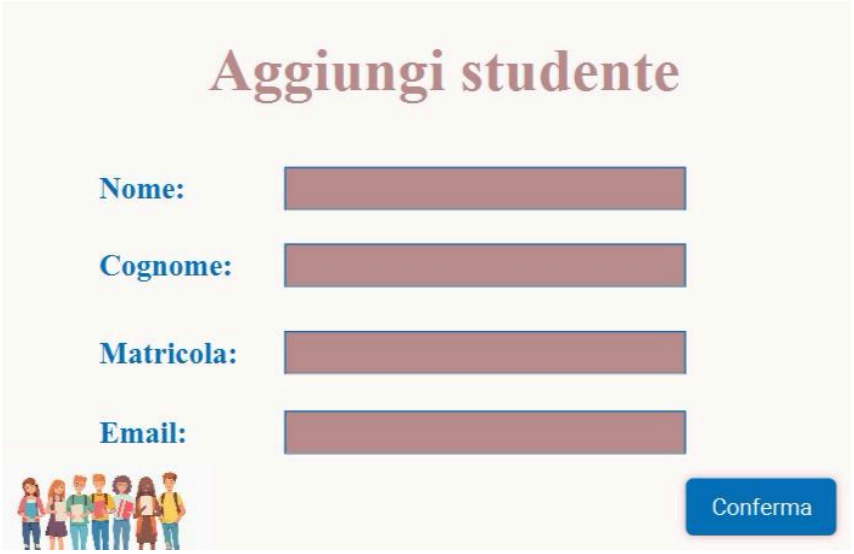
5.3. **Titolo:** “Modifica Libro”, serve ad identificare la sezione.
Campi di input: Sono presenti i 5 campi per l’identificazione e la descrizione di un libro, affiancati da aree di testo dove inserire i dati.
Pulsante d’azione: “Conferma”, serve a salvare le modifiche.



5.4. **Titolo:** “Ricerca Libro”, serve ad identificare la sezione.
Pulsanti: Due pulsanti orizzontali che rappresentano le funzioni successive alla ricerca, ovvero una modifica o una rimozione del libro.

5.5. **Titolo**: “Ricerca Prestito”, serve ad identificare la sezione.
Campo di input: È presente un campo di input per l’inserimento di due parametri: l’identificativo dello studente (cognome o matricola) e il titolo del libro.
Sfondo: L’icona della lente d’ingrandimento, posta accanto alla barra di ricerca, suggerisce chiaramente la funzione legata all’input testuale.
Pulsante d’azione: “Conferma”, serve a restituire dei risultati.

5.6. **Titolo**: “Aggiungi studente”, serve ad identificare la sezione.
Campi di input: Sono presenti i 4 campi per l’identificazione di un nuovo studente, affiancati da aree di testo dove inserire i dati.
Pulsante d’azione: “Conferma”, serve a salvare i dati dello studente.
Sfondo: In basso a sinistra, è presente un’immagine decorativa raffigurante degli studenti, che aggiunge un tocco visivo e rafforza il tema dell’interfaccia.



5.7. **Titolo**: “Registro studenti”, serve ad identificare la sezione.
Tabella: Contiene il registro delle informazioni degli studenti, ordinato per cognome e nome.
Sfondo: Attualmente la tabella mostra il messaggio "Nessun contenuto nella tabella", indicando che non ci sono dati registrati.

5.8. **Titolo**: “Ricerca Prestito”, serve ad identificare la sezione.
Pulsante d’azione: “Chiudi Prestito”, serve a registrare la restituzione di un libro preso in prestito.