Nel pdf allegato assieme a questa documentazione, è presente il diagramma delle classi. Quest'ultimo ci permette di individuare l'ossatura di un sistema software, poiché mette in evidenza le classi, i loro attributi (cioè i loro dati), i loro metodi (cioè le azioni/operazioni) e le relazioni tra di esse.

Nel nostro caso, abbiamo effettuato una serie di scelte per garantire una progettazione affidabile e duratura, con buoni livelli di coesione ed accoppiamento tra le classi.

COESIONE: permette di misurare quanto fortemente le parti che sono incluse nello stesso "modulo" lavorano insieme per raggiungere un unico scopo.

1) Per quanto riguarda le entità del nostro progetto, vi è una coesione funzionale molto alta, poiché tutti gli attributi ed i metodi permettono di rappresentare e gestire un singolo concetto di dominio (come un libro, uno studente o un prestito), senza che intervenga la GUI.

Dato che non ci sono sequenze ferree di passi da eseguire in un ordine preciso all'interno della stessa classe, la coesione sequenziale è molto meno evidente; né si vede coesione temporale, dato che le funzioni dei vari moduli sono indipendenti tra di loro).

Invece per quanto riguarda la coesione logica, ci potrebbero essere degli "esempi", poiché le classi Libro e Studente operano in maniera quasi simile per quanto concerne la raccolta dei dati. Inoltre, non vi è sicuramente coesione coincidentale, poiché le entità ed i loro metodi non sono assolutamente "scollegate" tra di loro.

2) Per quanto riguarda le collezioni, Catalogo ed Elenco (entrambi rispettivamente insiemi di Libro e Studente) presentano un'alta coesione funzionale, questo perché tutte le operazioni ruotano attorno alla gestione della collezione specifica. Vi è sicuramente coesione comunicazionale, per via della stessa struttura dati (TreeSet) che condividono entrambe le collezioni.

E dato che non appaiono gruppi di metodi/azioni prettamente generiche, escludiamo coesione coincidenziale e logica, oltre che temporale (per lo stesso discorso fatto per le entità).

3) Per quanto riguarda i controller, Menu_BibliotecaController, GestioneLibriController, GestioneStudentiController, Interfaccia_nuovoPrestitoController, VisualizzaStudente_viewController, vi è alta coesione funzionale, in parte legata alla schermata che controllano, in parte legata i dati su cui operano).

Vi è coesione anche per comunicazione, questo perché i metodi operano sugli stessi componenti GUI e spesso anche sugli stessi oggetti di dominio.

In alcuni controller vi è inoltre coesione sequenziale poiché certi metodi devono essere richiamati in un determinato ordine cronologico (ad es. in Interfaccia_nuovoPrestitoController facciamo: inserisci dati -> valida dati -> crea prestito -> aggiorna -> mostra notifica).

Non vi è una marcata coesione temporale e sicuramente non c'è coesione coincidentale, poiché le entità ed i loro metodi non sono assolutamente "scollegate" tra di loro.

ACCOPPIAMENTO: permette di misurare quanto un “modulo” dipende da un altro.

1) Per quanto riguarda le entità del nostro progetto, tra di loro sussiste un buon livello di accoppiamento: quello suoi dati, il livello più basso possibile (senza ovviamente considerare la possibile perfezione dovuta all’assenza totale di accoppiamento, una grande utopia). Questo perché la classi si passano parametri semplici e riferimenti tipati a oggetti di dominio e sfruttano i loro metodi pubblici, garantendo quindi che non vi sia accoppiamento per contenuti.

Non segnaliamo esempi di accoppiamento per contenuti (il peggiore), questo perché nel nostro progetto nessuna classe manipola direttamente le variabili private di un’altra; né segnaliamo esempi di accoppiamento per aree comuni, questo perché non riscontriamo l’uso di variabili globali o accessi ad un parametro da più classi contemporaneamente.

2) Per quanto riguarda i controller e la GUI, i primi sono accoppiati tramite dati alle entità e alle collezioni, seguendo l’esatto accoppiamento previsto in una architettura MVC / Entity-Control-Boundary, garantendo di ridurre il rischio che eventuali modifiche GUI “rompano” le entità.

Vi è tuttavia una piccola parte di accoppiamento per controllo, poiché i controller decidono il flusso dei casi d’uso e quindi possono invocare in sequenza delle operazioni su più entità (ad es. verifiche su Studente, l’aggiornamento su Prestito, notifica per una modifica, etc.) -> determinando un controllo più “diretto” da parte della GUI.

Per quanto riguarda l’accoppiamento per timbro, esso potrebbe “apparire”, più che a livello di class diagram, a livello delle firme dei metodi (poichè nel passaggio delle strutture dati potrebbero capitare anche informazioni non necessarie).

PRINCIPI DI BUONA PROGETTAZIONE:

1) Le classi di dominio si occupano solo del modello dati e di operazioni legate al dominio biblioteca, senza mischiare GUI e file. Seguono quindi l'applicazione del “Single Responsability Principle”: in questo modo le eventuali modifiche toccano solo una classe/interfaccia del programma, e non un intero modulo di esso.

2) Riteniamo che è stato seguito anche l’ “Open-Closed Principle”, poiché I nostri moduli, le classi e le funzioni a loro associate risultano essere più propense all'estensione piuttosto che alla modifica.

3) Abbiamo sicuramente fatto affidamento all’ “Interface Segregation Principle”, poiché abbiamo preferito un’implementazione con più interfacce, più specifiche e compatte, rispetto ad una implementazione più rigorosa ma meno efficace, permettendo così di ridurre l'accoppiamento tra componenti e facilitando la sostituzione di parti del sistema senza impatti sulle altre.

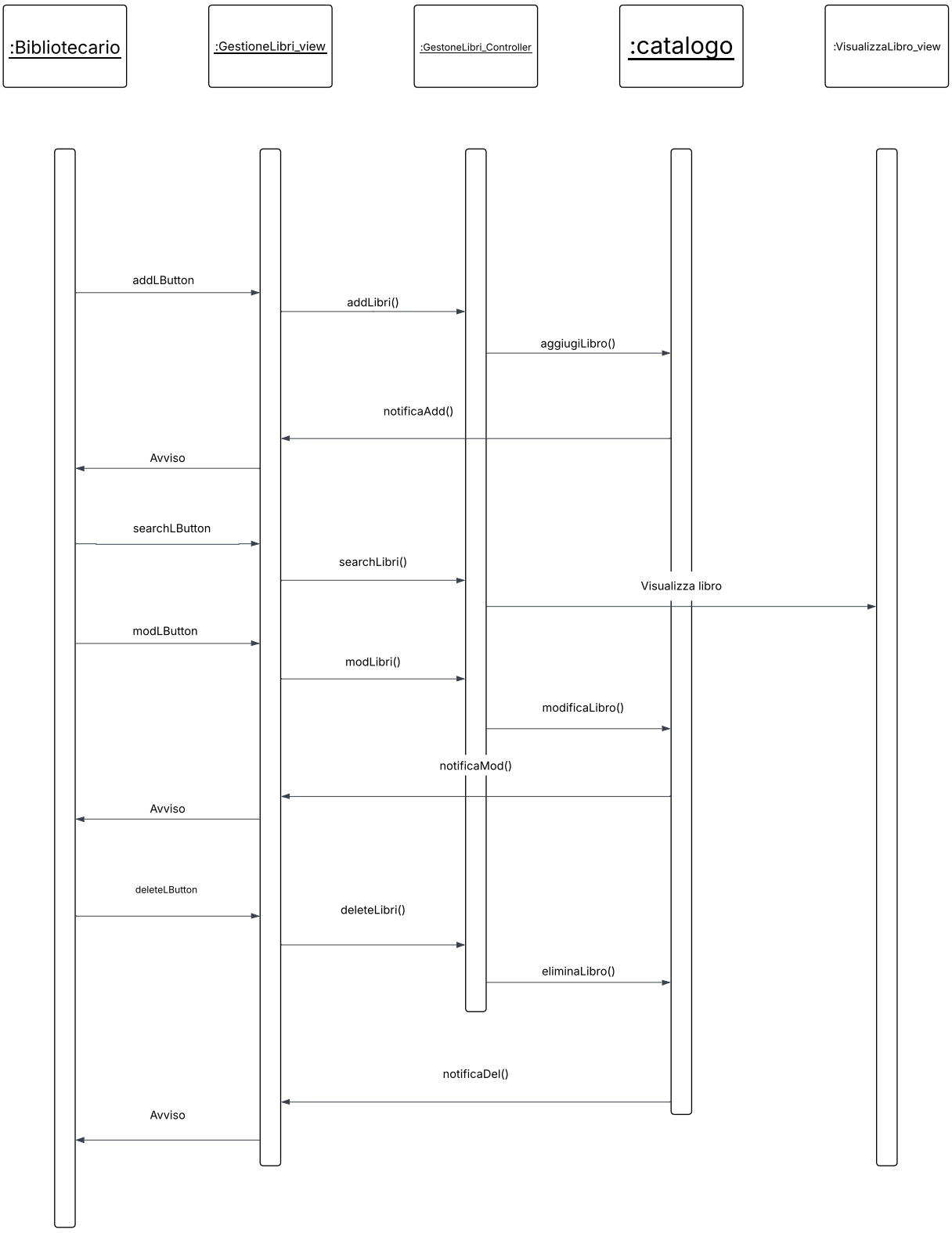
4) Ovviamente abbiamo seguito anche i principi di correttezza e robustezza: quindi il nostro sistema rispetto formalmente tutte le specifiche proposte dal progetto ed è in grado di gestire le eccezioni e di funzionare correttamente anche in presenza di input anomali.

5) Inoltre, tra di noi abbiamo seguito la regola del boy scout: ogni qualvolta uno di noi modificava il codice, prestava la massima attenzione per non comprometterne la qualità (abbiamo evitato quindi che eventuali errori si fossero protratti nel tempo, nel corso dei commit e delle push).

DIAGRAMMI DI SEQUENZA: questo tipo di diagramma metti in risalto il comportamento (di un sottoinsieme, di una sottoclasse) di un sistema software, in un singolo scenario. Cioè mostra quali oggetti sono coinvolti e quali messaggi vengono scambiati tra questi oggetti.

Qui il tempo segue l'ordine cronologico e scorre dall'alto verso il basso mentre gli oggetti sono elencati da sinistra verso destra e di messaggi sono rappresentati tramite frecce che indicano la direzione in cui viaggia il messaggio stesso.

1. Diagramma di sequenza per Libro:



Il seguente diagramma di sequenza descrive il flusso di interazioni per la gestione dei libri nel nostro sistema software di una biblioteca. Coinvolgendo ovviamente l'interfaccia GestioneLibri_view , il controller GestoneLibri_Controller ed il Catalogo.

Presenta come "attore" il Bibliotecario ed effettua una "visita" secondaria all'interfaccia VisualizzaLibro_view .

Flusso principale:

Inizia con l'attivazione dei pulsanti nell'interfaccia GestioneLibri_view, poi:

Operazioni Dettagliate:

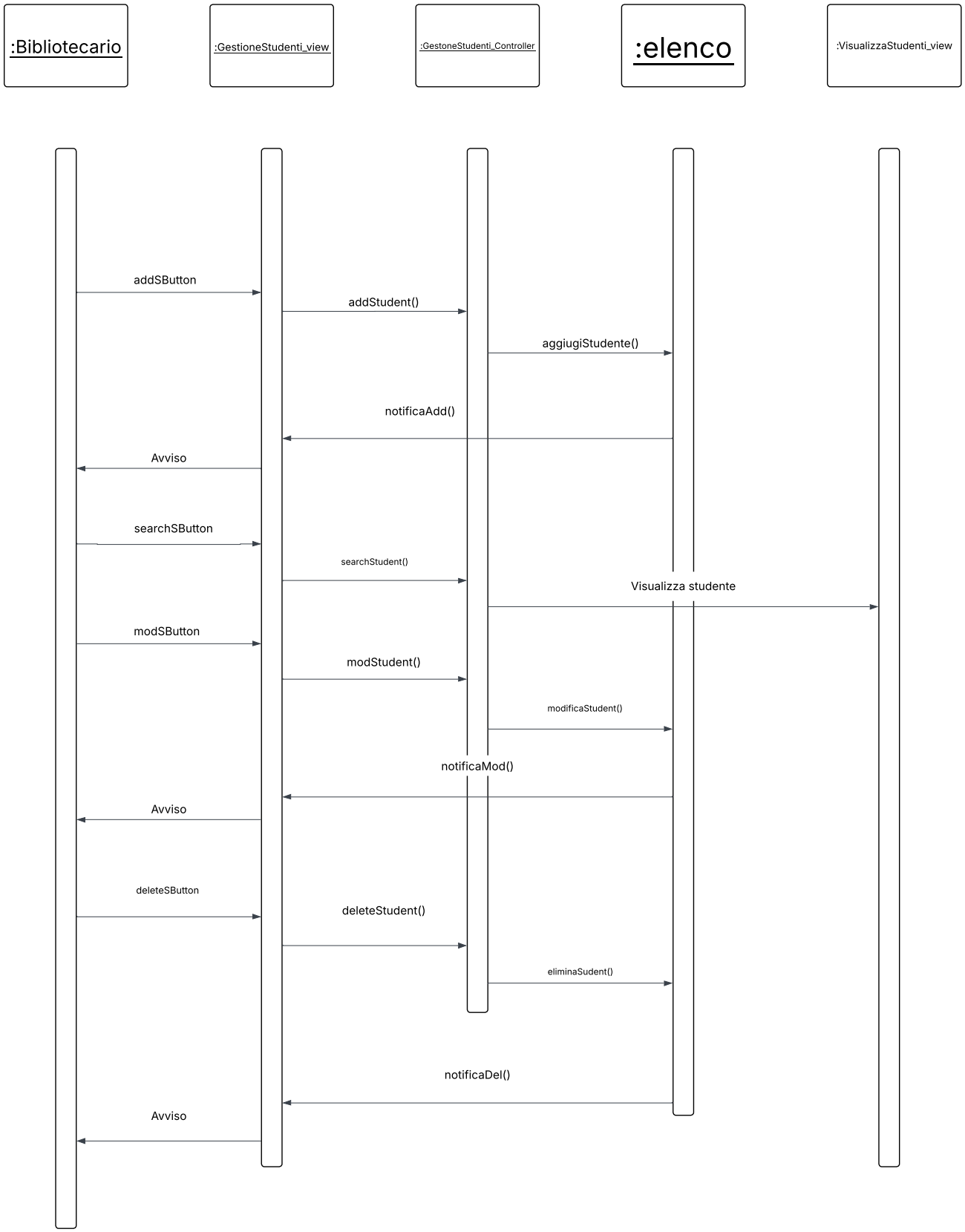
--Aggiunta libro: da addLButton si passa a addLibri() nel Bibliotecario, che chiama aggiugiLibro() e notificaAdd() tramite un Avviso.

--Ricerca e visualizzazione: searchLButton porta a searchLibri() nella VisualizzaLibro_view , che mostra "Visualizza libro" .

--Modifica: modLButton attiva modificaLibro() e notificaMod() con Avviso.

--Cancellazione: deleteLButton invoca eliminaLibro() e notificaDel() con Avviso.

2. Diagramma di sequenza per Studente:



Il seguente diagramma di sequenza illustra il flusso per la gestione degli studenti nel nostro sistema software di una biblioteca. Coinvolgendo l'interfaccia GestioneStudenti_view , il controller GestoneStudentiController , ed un Elenco.

Presenta come "attore" il Bibliotecario ed effettua una "visita" secondaria all'interfaccia VisualizzaStudenti_view .

Flusso Principale:

Inizia con l'attivazione dei pulsanti nell'interfaccia GestionStudenti_view, poi:

Operazioni Dettagliate:

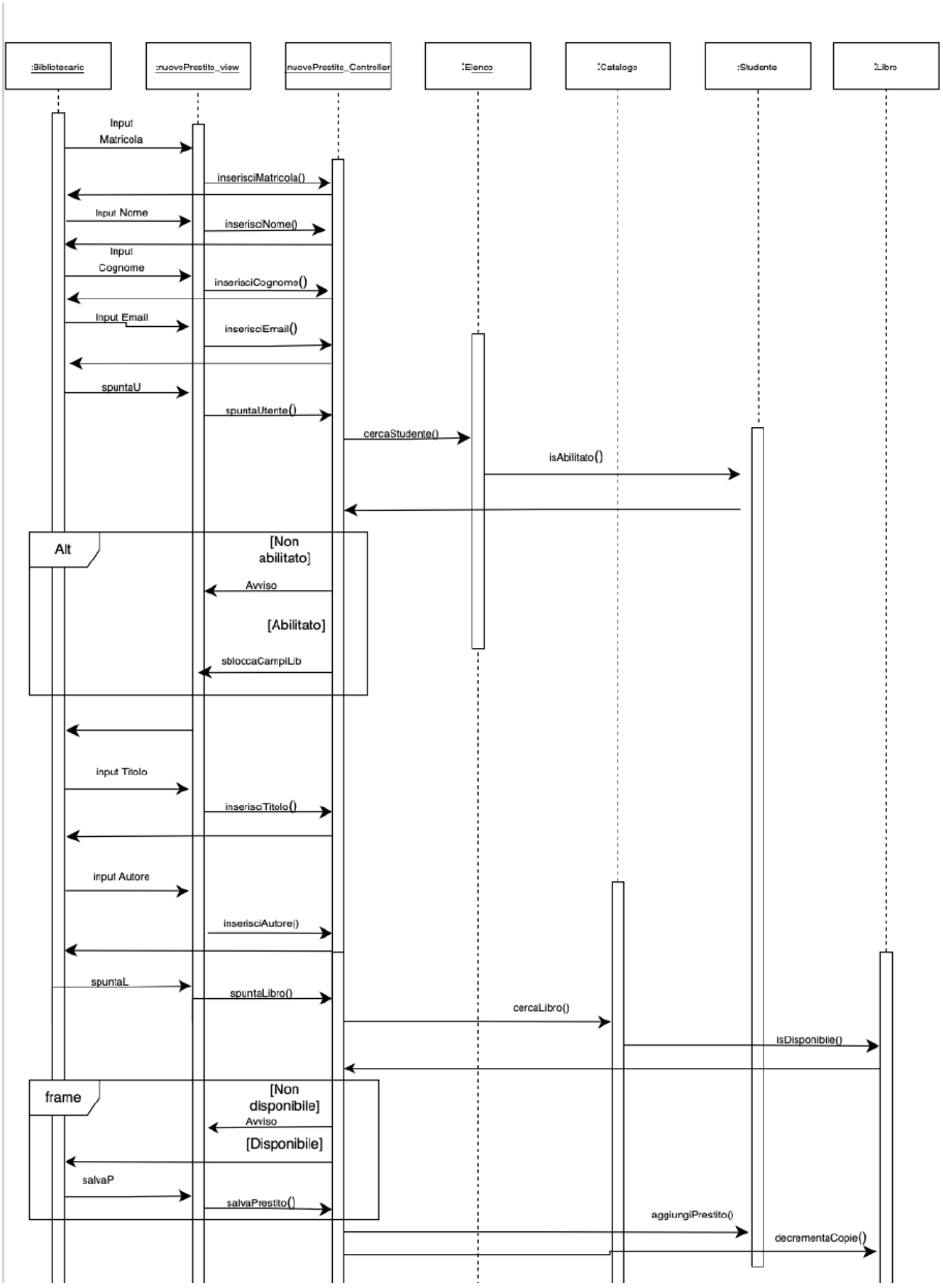
--aggiunta studente: addSButton porta a addStudent() nel Bibliotecario, che esegue aggiugiStudente() e notificaAdd() con Avviso.

--ricerca e visualizzazione: searchSButton attiva searchStudent() nell'interfaccia VisualizzaStudenti_view, mostrando "Visualizza studente".

--modifica: modSButton invoca modificaStudent() e notificaMod() con Avviso.

--cancellazione: deleteSButton chiama eliminaStudente() e notificaDel() con Avviso.

3. Diagramma di sequenza per Prestito:



Il seguente diagramma di sequenza modella il processo di creazione di un nuovo prestito all'interno del nostro sistema software di una biblioteca. Coinvolgendo l'attore (che è il Bibliotecario), l'interfaccia nuovo Prestito_view, il controller nuovoPrestito_Controller, un Elenco, un Catalogo, e le entità Studento e Libro.

Flusso principale:

Inizia con l'inserimento dati dello studente: inputMatricola, InserisciNome(), inserisciCognome)(, etc. --> poi spuntaUtente() attiva cercaStudente().

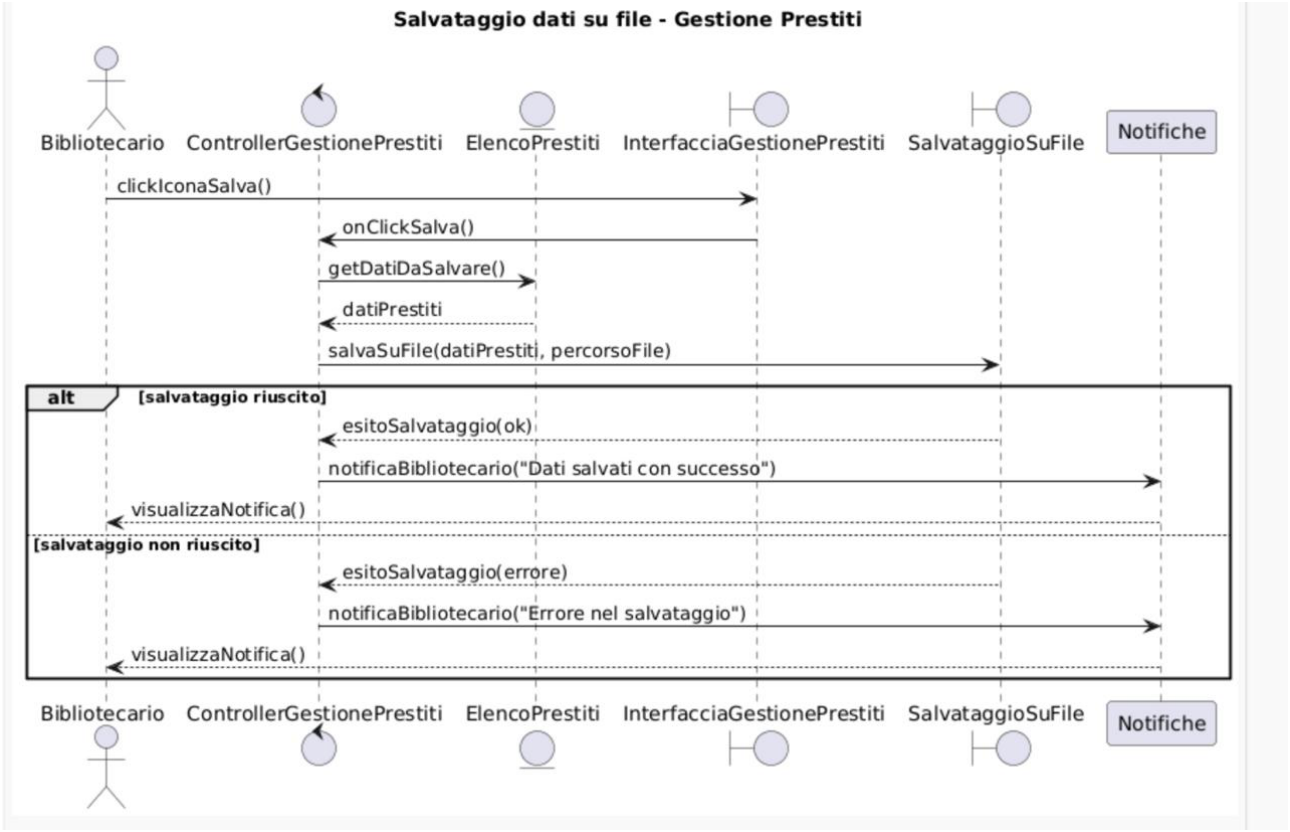
Operazioni dettagliate:

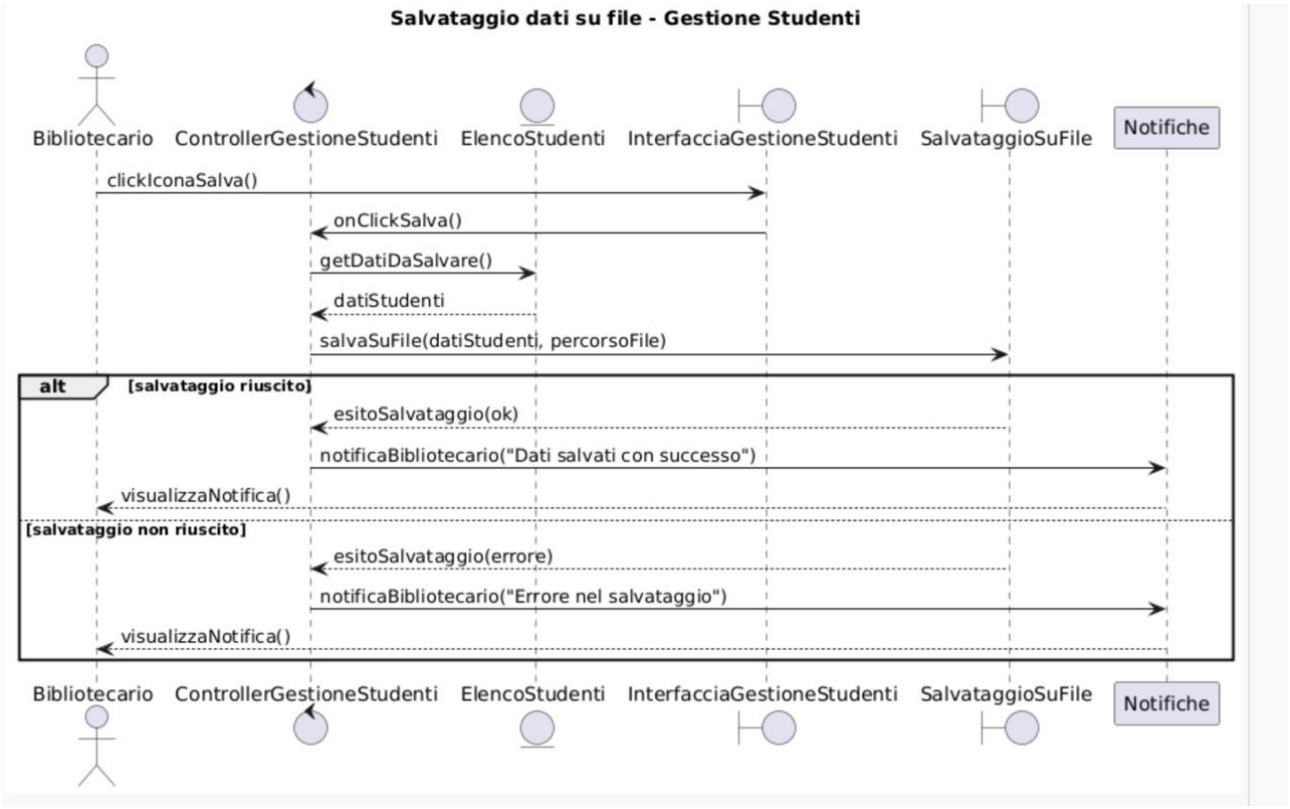
--un ALT verifica se lo studente è abilitato, tramite isAbilitato(): se non è abilitato, mostra Avviso(); se è abilitato chiama sbloccaCampiLib per procedere ai libri

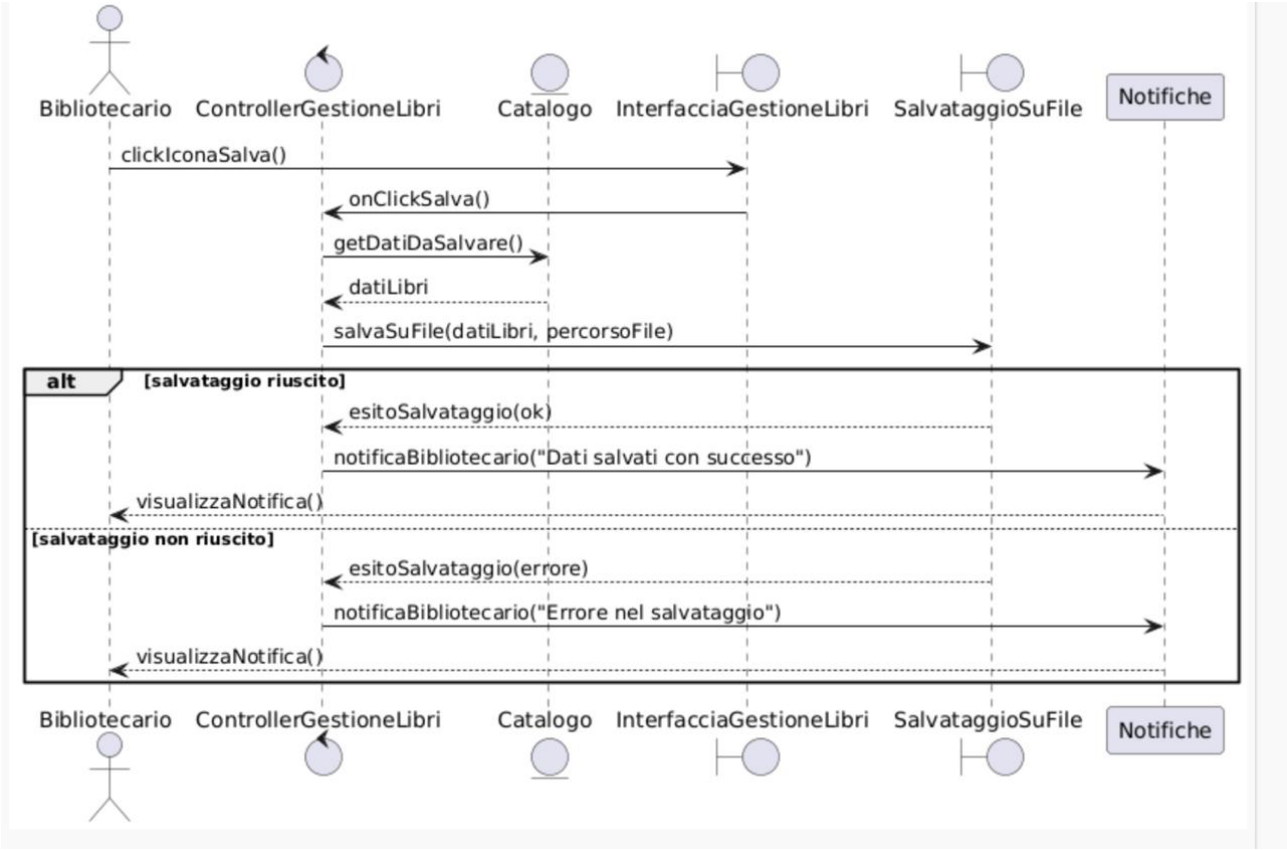
--selezione e verifica libro: si inseriscono input Titolo , inserisciTitolo() , input Autore, inserisciAutore(), seguiti da spuntaLibro() e cercaLibro(). Un frame controlla se iil libro è disponibile tramite IsDisponibile(): se non è disponibile, appare Avviso; se è disponibile , attiva salvaP .

--salvataggio prestito: salvaPrestito() invoca aggiungiPrestito() nell'Elenco e decrementaCopie() sul Libro.

4. Diagramma di sequenza su Salvataggio su File (per prestiti, studenti e libri):







Questi tre diagrammi seguono tutti il pattern MVC, per gestire eventuali errori.

Sono molto simili tra di loro, poiché tutti e 3 si concentrano sul salvataggio di una serie di dati su un file, tutto ciò ovviamente nel nostro sistema software di una biblioteca.

A) Il primo diagramma modella il salvataggio dei dati dei libri su file, con attore ovviamente il bibliotecario. Comprende il controller ControllerGestioneLibri, un Catalogo (di libri), l'interfaccia InterfacciaGestioneLibri, e una sezione legata alle Notifiche/Avvisi, oltre che una per il salvataggio effettivo su file.

Inizia con clickIconaSalva che attiva onClickSalva() nel controller, seguito da getDatiDaSalvare() per ottenere datiLibri e salvaSuFile(datiLibri, percorsoFile).

--Un ALT gestisce esitoSalvataggio: se ok porta a notificaBibliotecario("Dati salvati con successo") e visualizzaNotifica ; se errore porta a notificaBibliotecario("Errore nel salvataggio") e visualizzaNotifica.

B) il secondo diagramma modella il salvataggio dei dati degli studenti su file, con attore ovviamente il bibliotecario. Comprende il controller ControllerGestioneStudenti, un Elenco (di studenti), l'interfaccia InterfacciaGestioneStudenti e una sezione legata alle Notifiche/Avvisi, oltre che una per il salvataggio effettivo su file.

Inizia con clickIconaSalva che attiva onClickSalva() nel controller, seguito da getDatiDaSalvare() per ottenere datiStudenti e salvaSuFile(datiStudenti, percorsoFile).

--Un ALT gestisce esitoSalvataggio: successo attiva notificaBibliotecario("Dati salvati con successo") e visualizzaNotifica ; errore invia notificaBibliotecario("Errore nel salvataggio") e visualizzaNotifica.

C) il terzo diagramma modella il salvataggio dei dati dei prestiti su file, con attore ovviamente il bibliotecario. Comprende il controller ControllerGestionePrestiti, un Elenco (di studenti), un Catalogo (di prestiti), che abbiamo deciso di fondere assieme per essere più leggibile nel diagramma, l'interfaccia InterfacciaGestionePrestiti, e una sezione legata alle Notifiche/Avvisi, oltre che una per il salvataggio effettivo su file.

Inizia con clickIconaSalva che attiva onClickSalva() nel controller, seguito da getDatiDaSalvare() per ottenere i datiPrestiti e salvaSuFile(datiPrestiti, percorsoFile).

--Un ALT gestisce esitoSalvataggio: se ok porta a notificaBibliotecario("Dati salvati con successo") e visualizzaNotifica ; se errore porta a notificaBibliotecario("Errore nel salvataggio") e visualizzaNotifica.