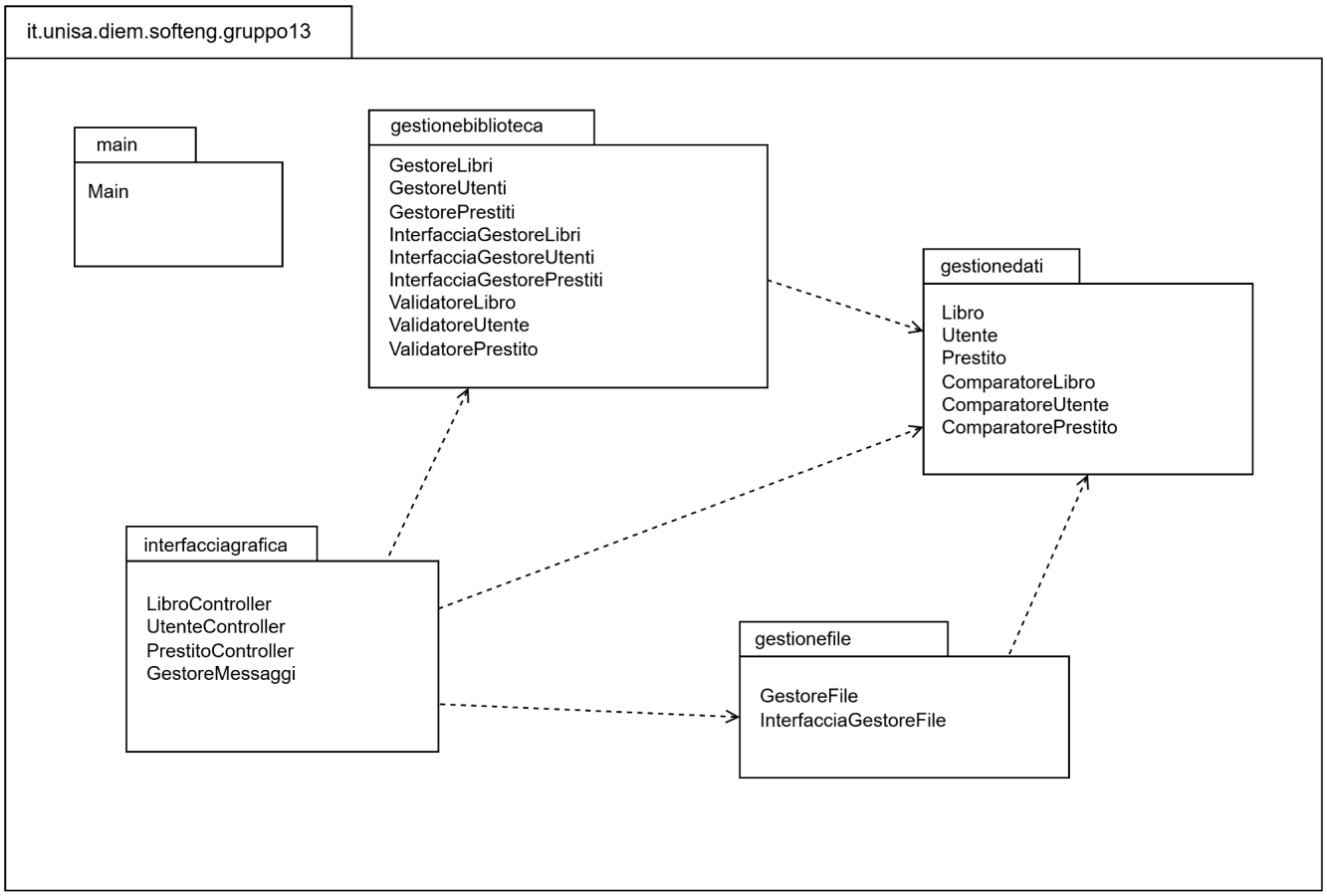
Design

DESIGN
Diagrammi e scelte di progettazione

Gruppo 13: Bello Daniel, Capaldo Daniele, De Santis Andrea, Ferrara Stefano

INDICE

DIAGRAMMA DEI PACKAGE



COMMENTO AL DIAGRAMMA

Il diagramma presentato illustra l'architettura del sistema software, organizzata secondo un'**architettura a livelli**. Questa suddivisione garantisce una netta **separazione delle responsabilità** e facilita la **manutenibilità** e l'**estendibilità** del codice.

- Il package **main** contiene la classe Main, che funge da punto di ingresso dell'applicazione avviando JavaFX.

- Il package **interfacciagrafica** è il livello più alto dell'applicazione, che gestisce l'interazione con l'utente.

  Questo package dipende da tutti gli altri package: **gestionebiblioteca** (per la logica), **gestionedati** (per manipolare gli oggetti da visualizzare) e **gestionefile** (per richiedere il salvataggio dei file).

- Il package **gestionebiblioteca** rappresenta il cuore funzionale del sistema. I **Gestori** manipolano le liste in memoria, i **Validatori** centralizzano se le regole di validazione dei dati, le **Interfacce** definiscono i contratti operativi.

  Utilizza il package **gestionedati** per definire e ordinare le strutture su cui operare. Non dipende né dalla GUI né dal gestore file.

- Il package **gestionefile** isola le logiche di salvataggio e caricamento dei dati su file esterni.

  Dipende solo dal package **gestionedati,** per sapere cosa salvare. È completamente isolato dalla logica di business.

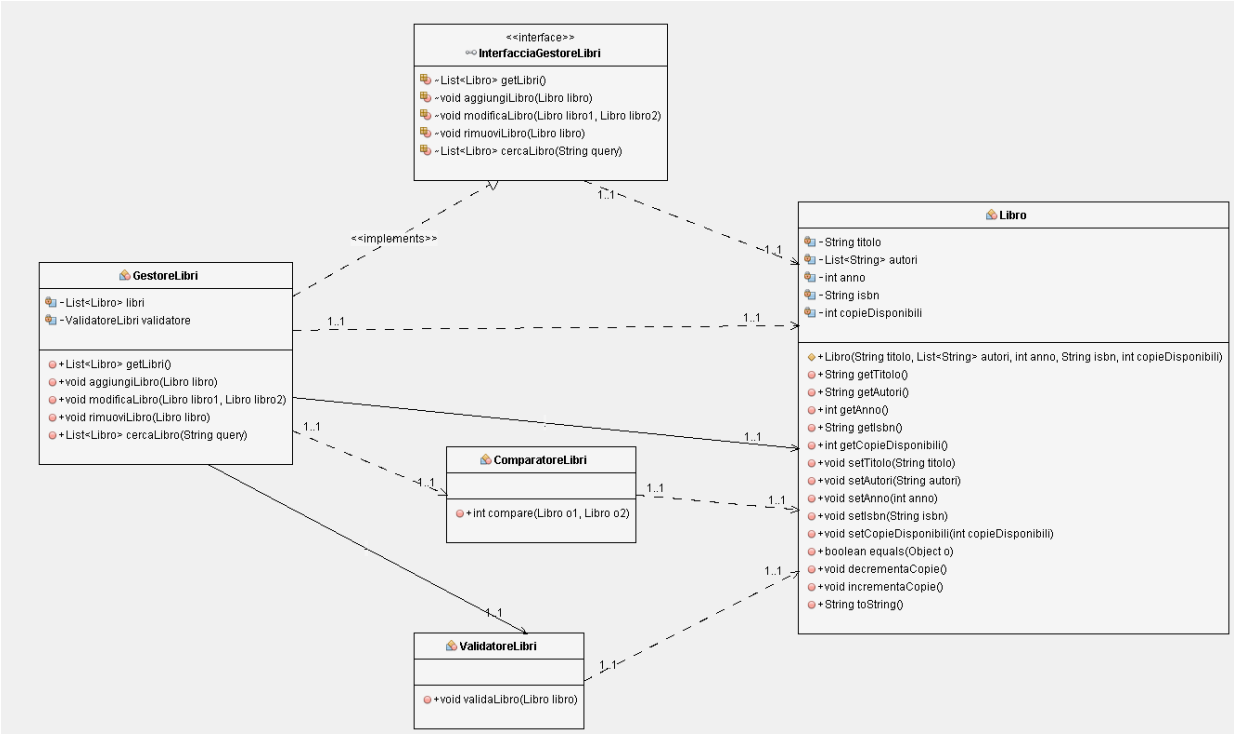
- Il package **gestionedati** è il livello più basso della gerarchia: fornisce le strutture dati utilizzate da tutti gli altri package.
  Essendo il livello base, **non ha dipendenze** verso l'esterno.

DIAGRAMMA DELLE CLASSI

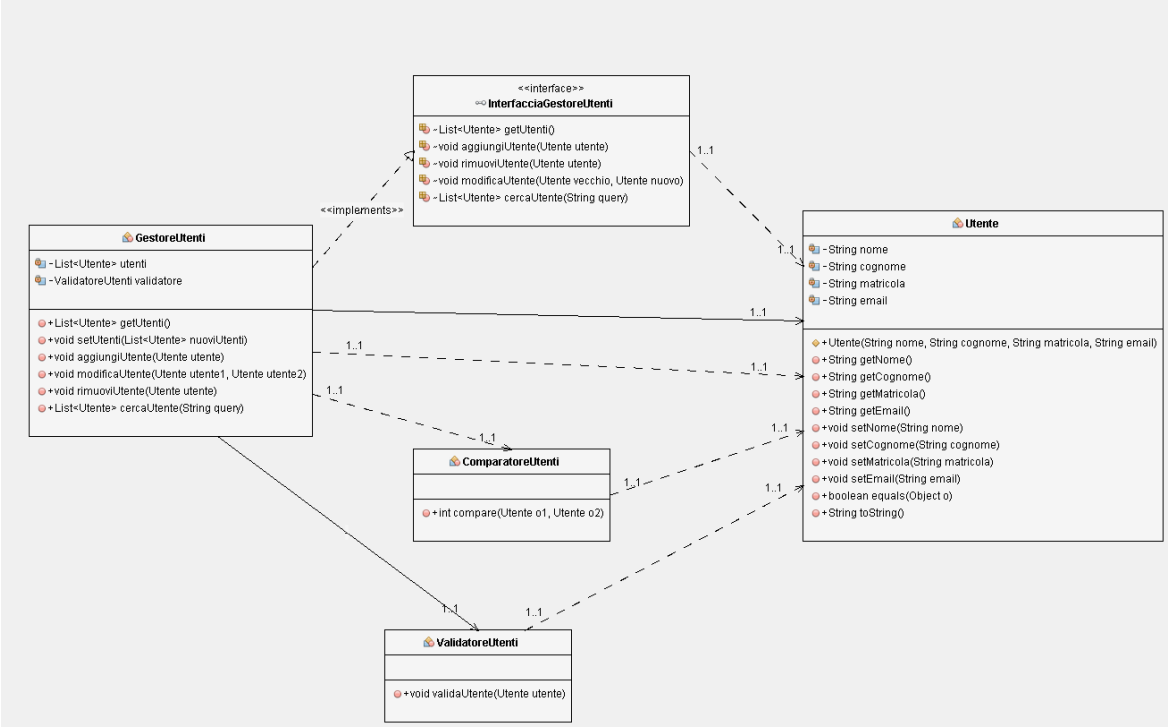
Il Diagramma delle Classi illustra le **relazioni statiche** tra i componenti del sistema.

Per maggiore chiarezza il diagramma principale è stato diviso in più parti.

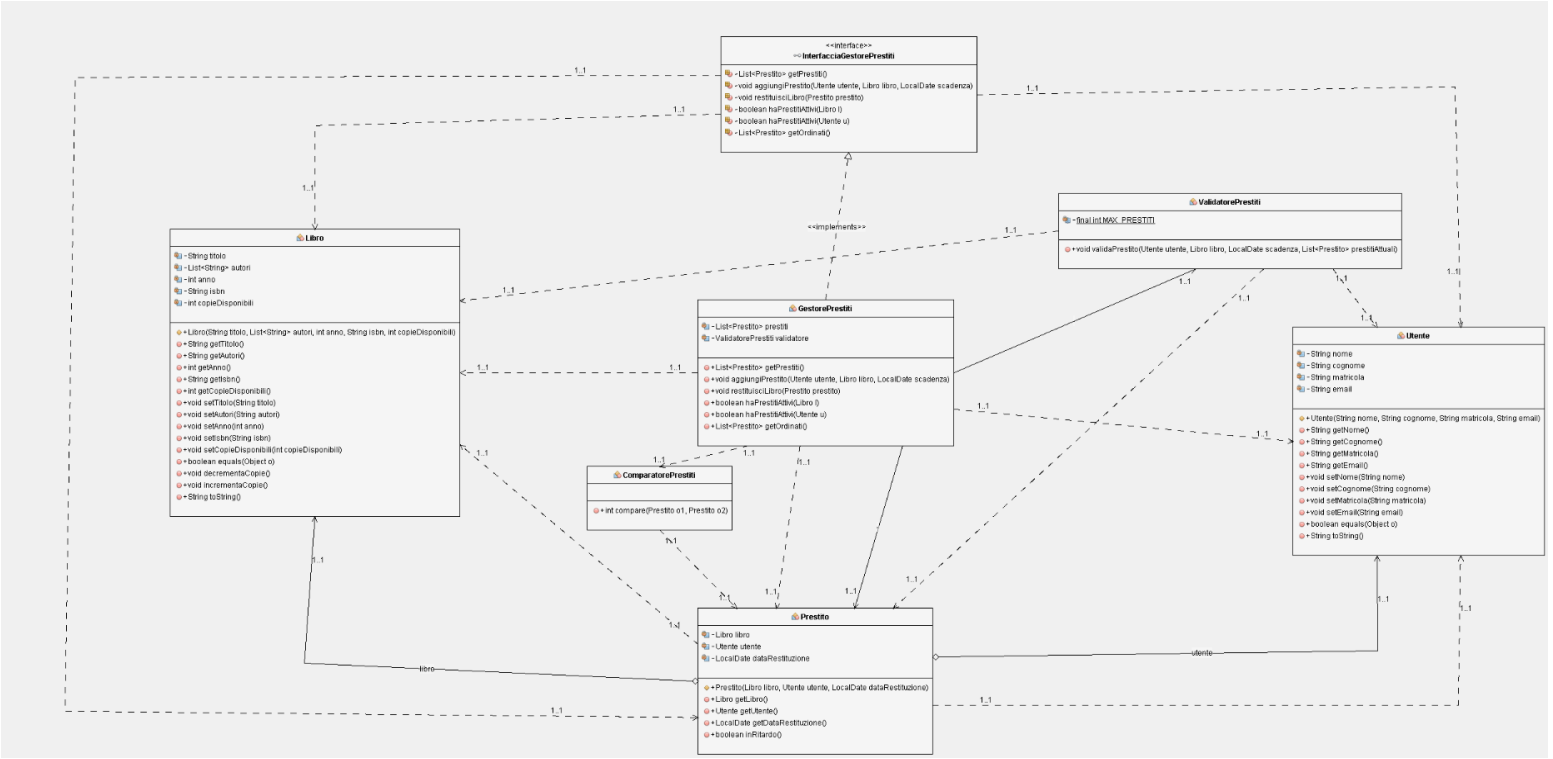
Il diagramma sottostante mostra le relazioni tra **Libro**, **GestoreLibri**, **InterfacciaGestoreLibri**, **ValidatoreLibro** e **ComparatoreLibro**.



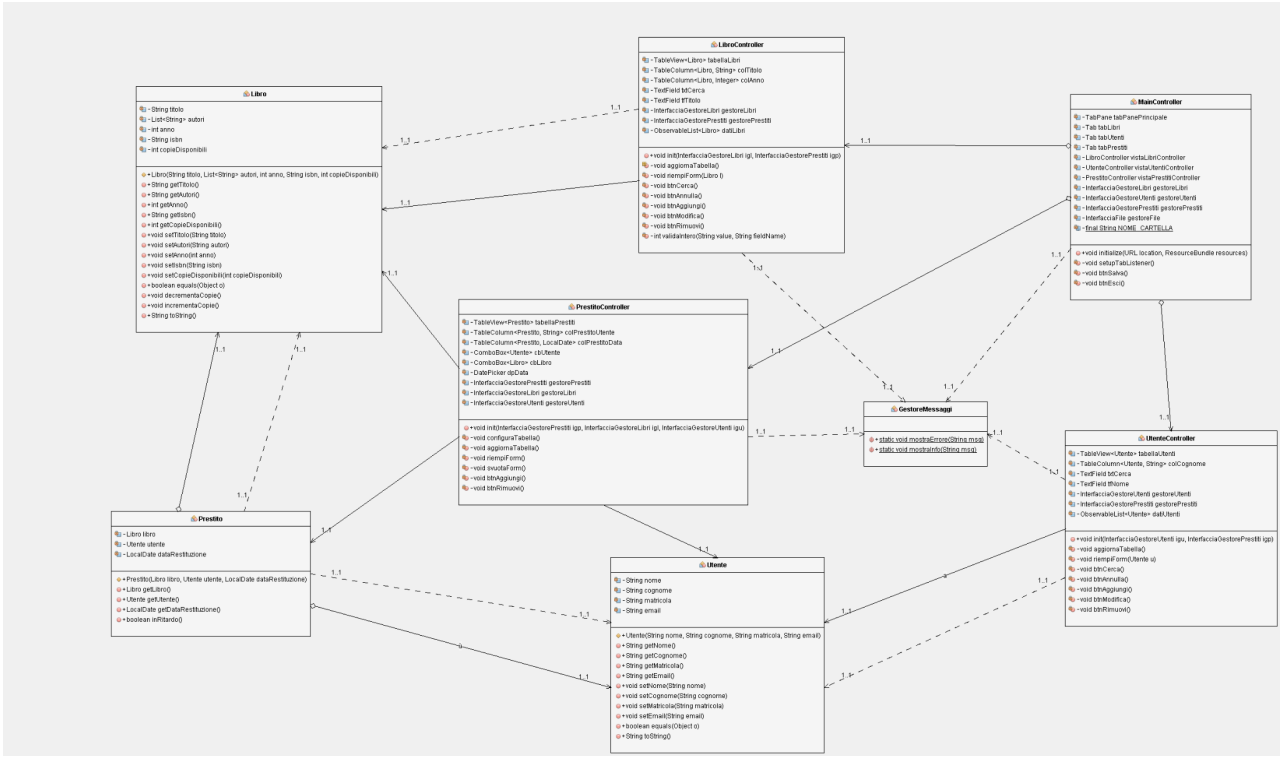
Il diagramma sottostante mostra le relazioni tra **Utente**, **GestoreUtenti**, **InterfacciaGestoreUtenti**, **ComparatoreUtente** e **ValidatoreUtente**.



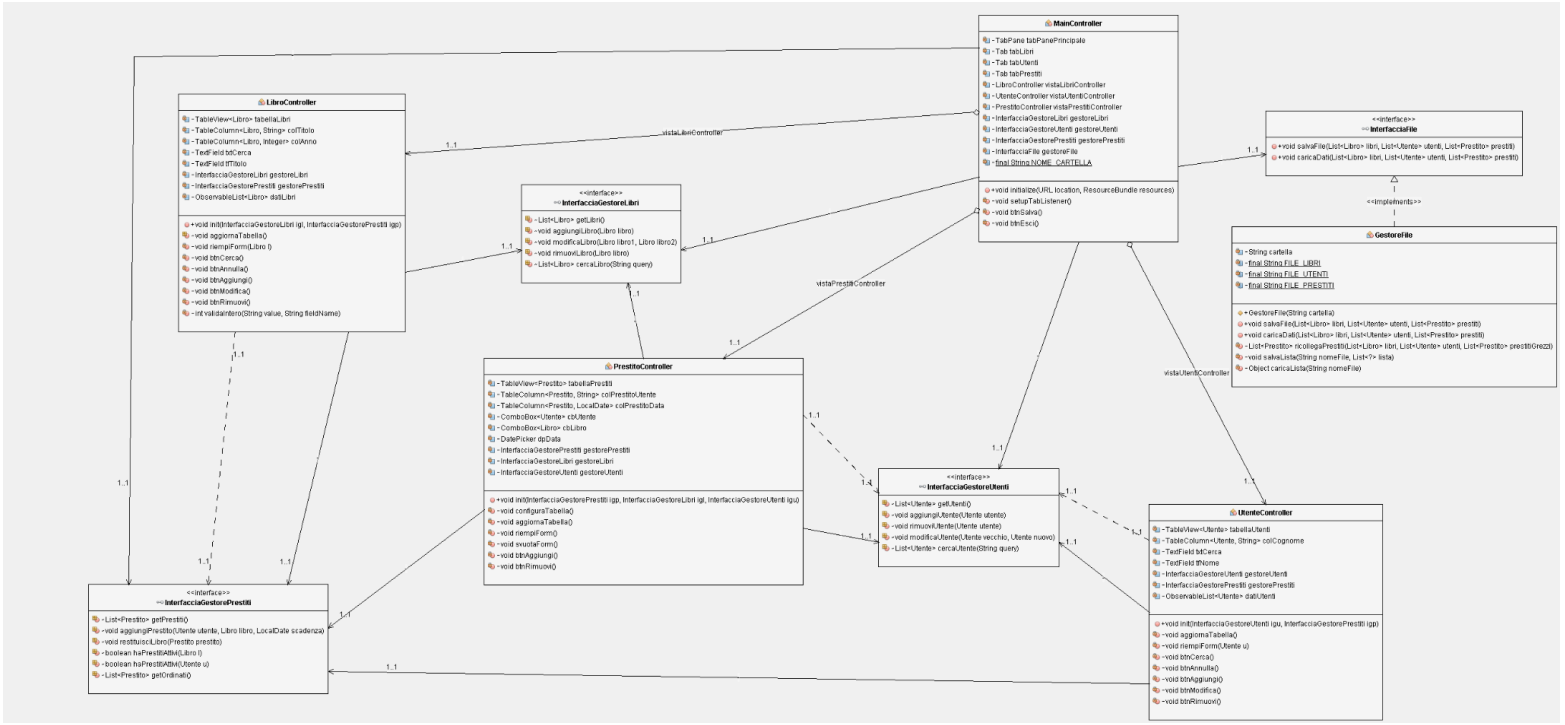
Il diagramma sottostante mostra le relazioni tra **Prestito**, **GestorePrestiti**, **InterfacciaGestorePrestiti**, **ValidatorePrestito**, **ComparatorePrestito**, **Libro** e **Utente.**



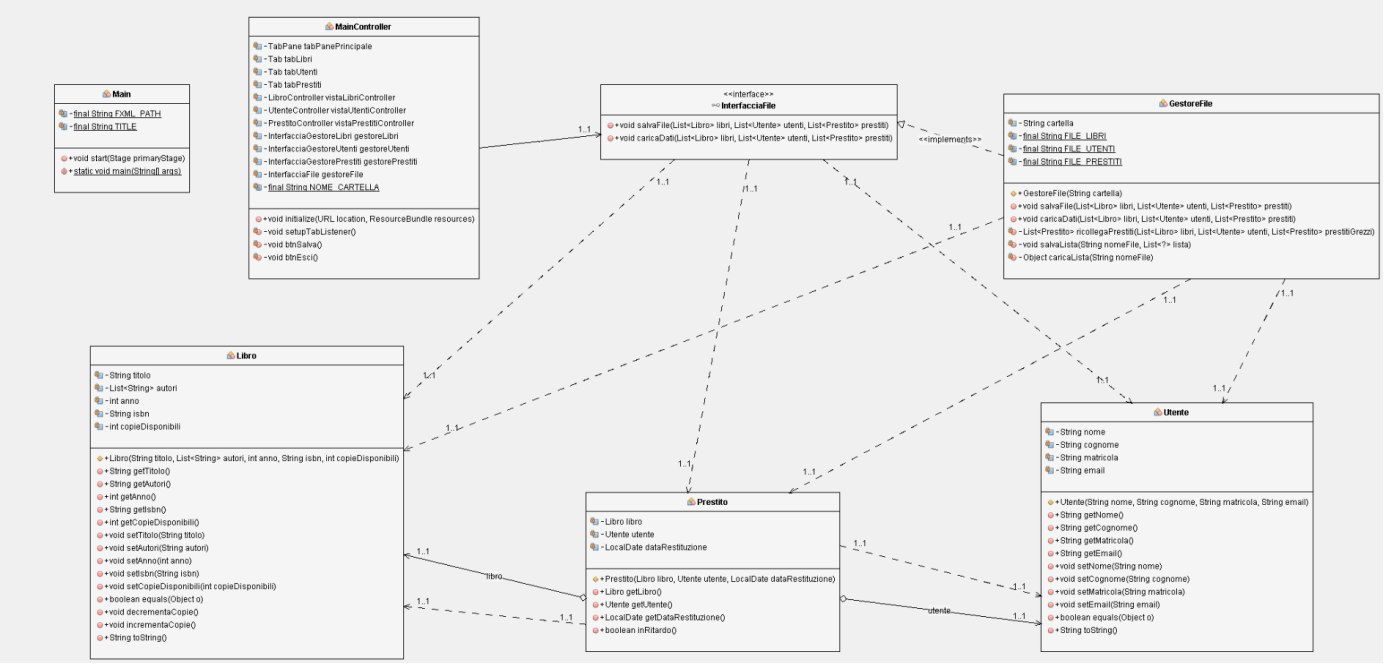
Il diagramma sottostante mostra le relazioni tra i **Controller** e i **Dati**.



Il diagramma sottostante mostra le relazioni tra i **Controller** e le **Interfacce dei Gestori.**



Il diagramma sottostante mostra le relazioni tra **Main**, il **Gestore File** e i **Dati.**



SCELTE DI PROGETTAZIONE

Per la progettazione architetturale è stato usato un approccio **Object Oriented**, suddividendo le classi in base alle entità che rappresentano il dominio del problema, come Libro, Utente e Prestito. Le operazioni delle classi invece sono organizzate in metodi con **approccio funzionale**, basato sulle operazioni che devono essere realizzate, come l'aggiunta, la modifica, la ricerca e l'eliminazione.

ATTRIBUTI DI QUALITÀ

L'intera architettura del sistema è stata definita a partire dalla necessità di soddisfare specifici **Attributi di Qualità** (Q.A.), sia interni che esterni.

Per quanto riguarda gli **attributi interni**, l'architettura presenta buoni livelli di **manutenibilità** e **riusabilità**: l'obiettivo primario è stato garantire che il software sia **facilmente modificabile** ed **estendibile** nel tempo.
Per ottenere questo risultato, il passo principale è stato dividere il sistema in livelli logici distinti, in modo che ogni classe avesse il minimo impatto sull'altra, aumentando così anche la **modularità.**
Ciò ha permesso di realizzare classi generiche e **riutilizzabili**, come i comparatori, i validatori, i gestori di file, il gestore delle finestre di dialogo.

Per quanto riguarda gli **attributi di qualità esterni**, sono stati privilegiati **robustezza** e **usabilità**, così che il sistema sia in grado di gestire situazioni anomale e input errati senza terminare in modo imprevisto, fornendo all'utente un'interazione fluida e priva di ostacoli.

PRINCIPI DI BUONA PROGETTAZIONE

Lo sviluppo del progetto è stato guidato da una rigorosa adesione ai principi fondamentali dell'ingegneria del software, con l'obiettivo primario di garantire una buona **qualità architetturale.**

La filosofia progettuale ha privilegiato la semplicità rispetto alla complessità, seguendo il principio **KISS.** Parallelamente, abbiamo usato il principio **DRY** per ridurre le duplicazioni e tenere il codice pulito: ogni logica ricorrente è stata centralizzata ed eliminata dalle classi principali. Da qui nasce, ad esempio, la scelta di delegare una classe specializzata (GestoreMessaggi) per la comparsa delle finestre di dialogo nei vari Tab.

L'architettura del sistema riflette il principio di **Separation of Concerns**, organizzando il codice in livelli logici ben distinti, separando grafica, logica, persistenza e dati. Un vantaggio di questo approccio è la **flessibilità**: un'eventuale modifica dell'interfaccia grafica non richiederebbe alcuna modifica alla logica di business o al salvataggio dati.

Questa organizzazione riflette fedelmente il **Single Responsibility Principle**. Un esempio tangibile di questa scelta è l'implementazione dei validatori come entità esterne ai gestori e ai modelli (Libro, Utente, Prestito). Questa separazione garantisce un'**elevata coesione**: se in futuro dovessero cambiare i criteri per la concessione di un prestito, sarà sufficiente intervenire su ValidatorePrestito, senza rischiare di arrecare problemi alle altre classi.

È stato rispettato l'**Open-Closed Principle** rendendo il sistema **aperto all'estensione** ma **chiuso alla modifica**. Questo è possibile grazie all'uso strategico dell'incapsulamento e delle **interfacce**. Qualora fosse necessario introdurre una nuova modalità di persistenza o una nuova logica di gestione, sarebbe sufficiente creare una nuova classe che implementi l'interfaccia esistente. Il codice dei controller, dipendendo solo dalle interfacce, non richiederebbe alcuna modifica per accogliere la nuova funzionalità.

La correttezza di questo meccanismo di sostituzione è assicurata dal rispetto del **Liskov Substitution Principle**. Le interfacce sono state progettate definendo **contratti chiari e precisi**; ciò assicura che qualsiasi futura implementazione possa sostituire quella attuale senza alterare il funzionamento del sistema. Inoltre, invece di definire un'unica interfaccia per l'intero sistema, sono state create **interfacce specifiche** per i vari contesti (**Interface Segregation Principle).**

COESIONE E ACCOPPIAMENTO

L'obiettivo del design è stato quello di **massimizzare la coesione** verso il livello più alto, quindi facendo in modo che le parti dei moduli fossero molto legate tra loro. Allo stesso tempo, un ulteriore obiettivo è stato quello di raggiungere il livello ideale di Accoppiamento per Dati (**Data Coupling**), evitando forme più rigide attraverso anche l'uso di un corretto incapsulamento e variabili non globali.

Per **LibroController**, **UtenteController** e **PrestitoController** la coesione è **comunicazionale** perché includono funzionalità che lavorano sugli stessi dati. L'accoppiamento con gestori e modelli è **per dati** in quanto passano i dati ai gestori e chiamano i metodi pubblici delle interfacce per eseguire le operazioni.

Il **MainController** ha coesione **procedurale**: i metodi come initialize o quelli che gestiscono il cambio Tab sono raggruppati perché fanno parte della procedura di esecuzione dell'interfaccia. L'accoppiamento è **per dati**: coordina il passaggio di liste.

Il **GestoreMessaggi**, svolgendo il solo compito di mostrare informazioni, ha coesione **funzionale** e accoppiamento **per dati,** visto che scambia come parametro solo la stringa da utilizzare.

La coesione del **GestoreFile** è **sequenziale**. La classe salva e carica dati, ricevendo le liste di dati come parametri (accoppiamento **per dati**).

GestoreLibri, **GestoreUtenti** e **GestorePrestiti** hanno coesione **funzionale** perché si focalizzano sulle rispettive collezioni di dati. Il livello di accoppiamento è **per dati** in quanto ricavano da modelli, validatori e comparatori unicamente le informazioni strettamente necessarie per il funzionamento

ValidatoreLibro, **ValidatoreUtente** e **ValidatorePrestito** svolgono il singolo compito di validazione, portando ad avere una coesione massima (**funzionale**) e accoppiamento **per dati** con i modelli**,** visto che ricevono i dati da validare e restituiscono il risultato.

Coesione **funzionale** e accoppiamento **per dati** caratterizzano anche **ComparatoreLibro**, **ComparatoreUtente** e **ComparatorePrestito** per motivi analoghi.

Le entità **Libro**, **Utente**, **Prestito** comunicano con l'esterno solo tramite passaggio di parametri semplici nei getter/setter (accoppiamento **per dati**) e svolgono l'unico compito di definire le strutture dati su cui le altre classi operano (coesione **funzionale**).

Il **Main** è **funzionale** dal momento che contiene i metodi per l'esecuzione dell'applicazione.

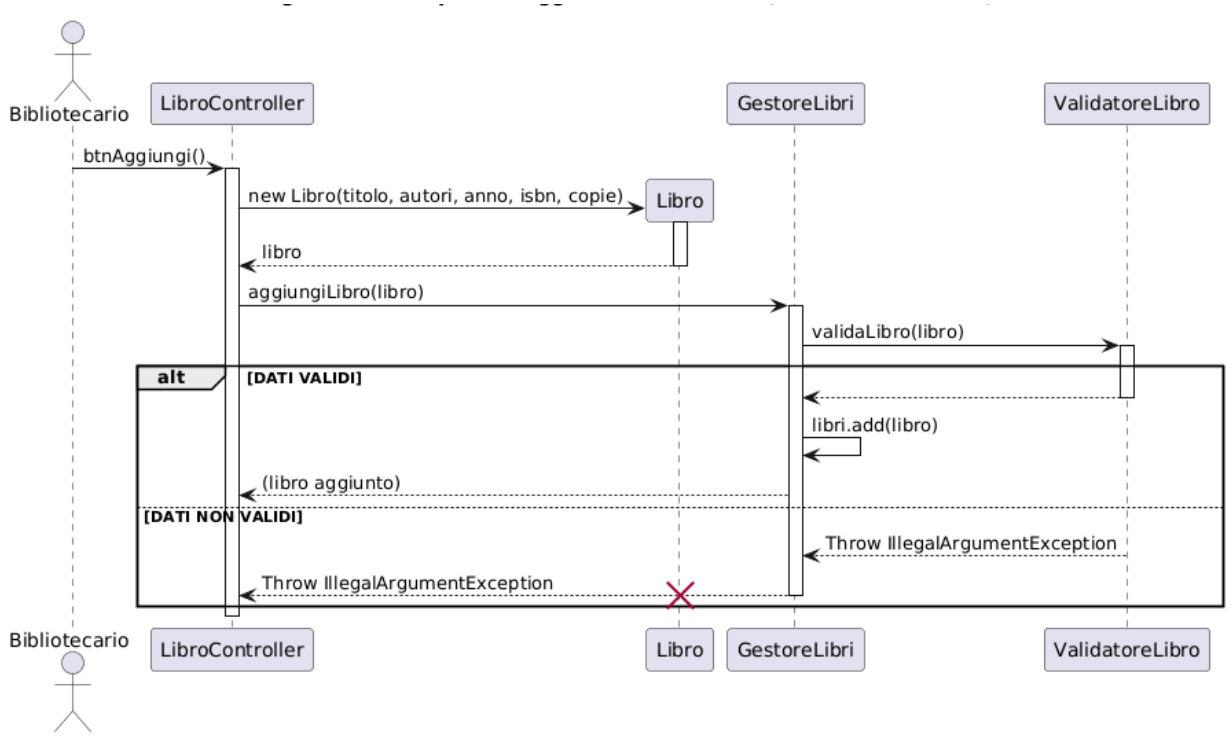
DIAGRAMMI DI SEQUENZA

I diagrammi di sequenza illustrano il **flusso logico** e le interazioni tra i componenti dell'architettura. Al fine di garantire una maggiore visibilità, talvolta sono stati omessi i dettagli implementativi di basso livello. In particolare, sono stati omessi i metodi accessori (getter e setter), le operazioni di aggiornamento delle viste (come l'aggiornamento delle tabelle) e la gestione grafica dei messaggi.

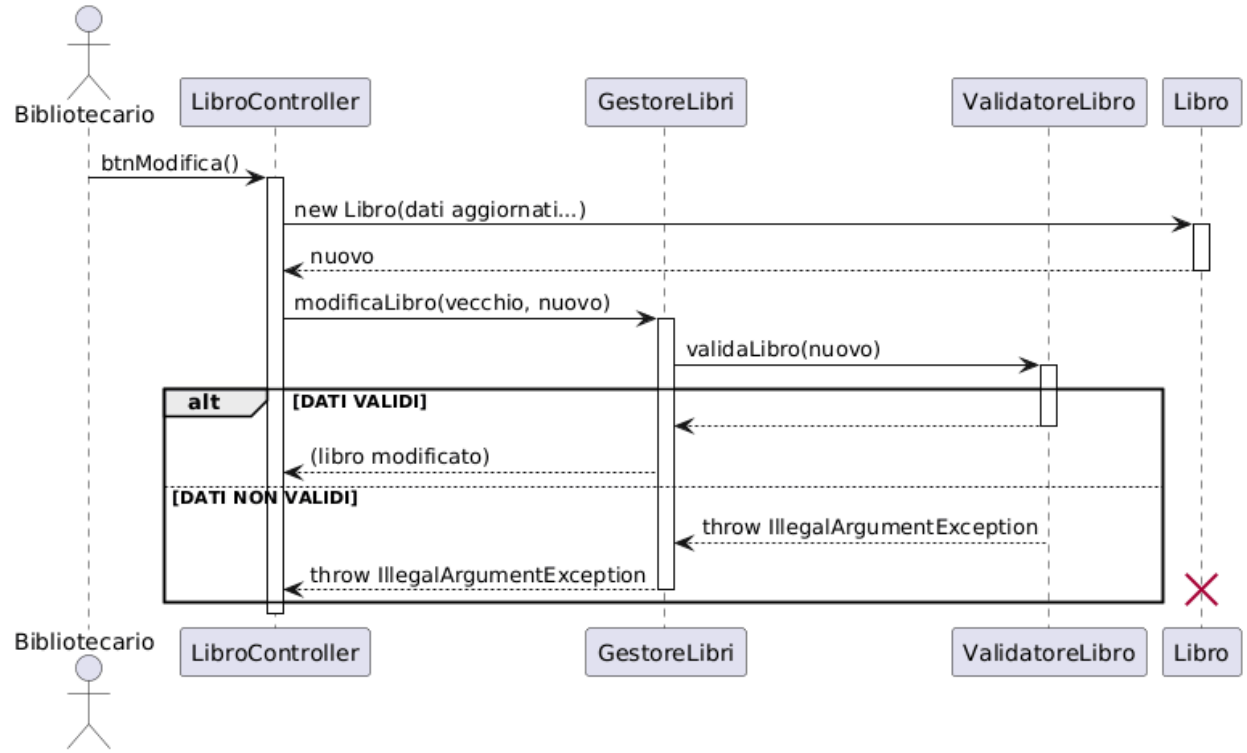
Come scenari più significativi sono stati considerati le **aggiunte**, le **modifiche**, le **eliminazioni** e il **salvataggio su file**.

Per le modifiche e le eliminazioni si assume come precondizione che gli oggetti coinvolti siano stati preventivamente selezionati dall'utente.

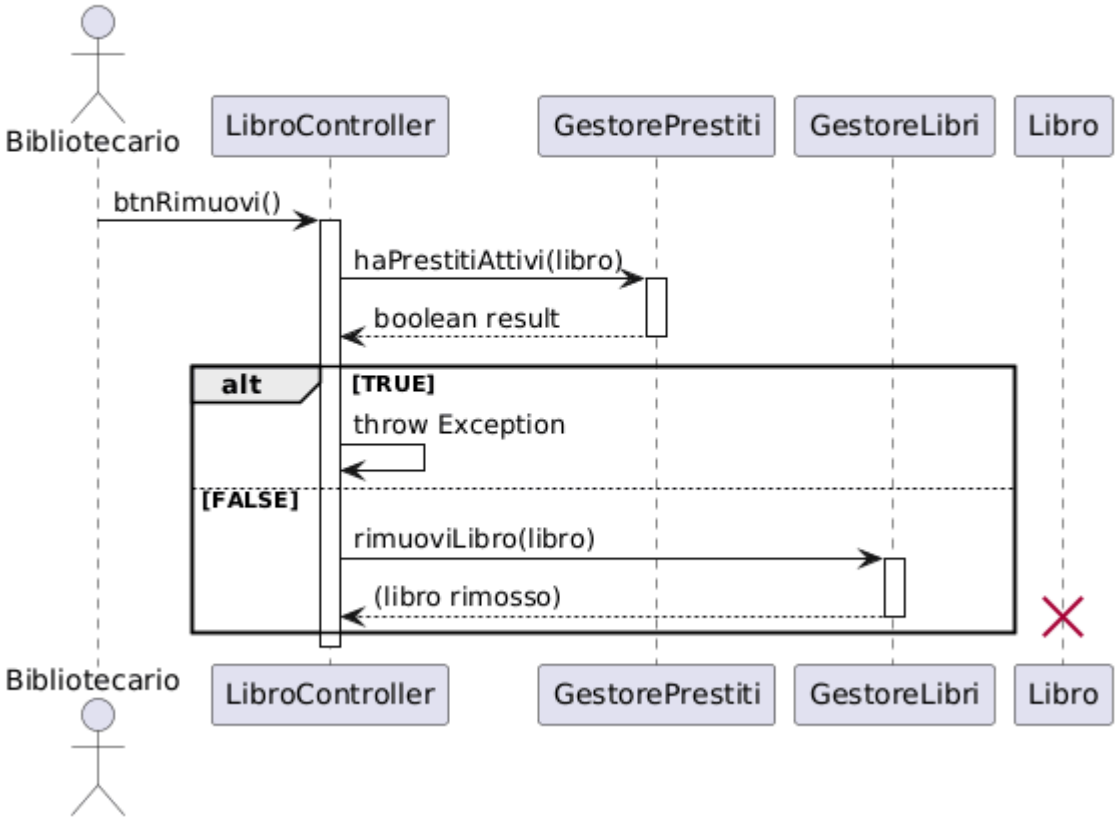
AGGIUNTA DI UN NUOVO LIBRO



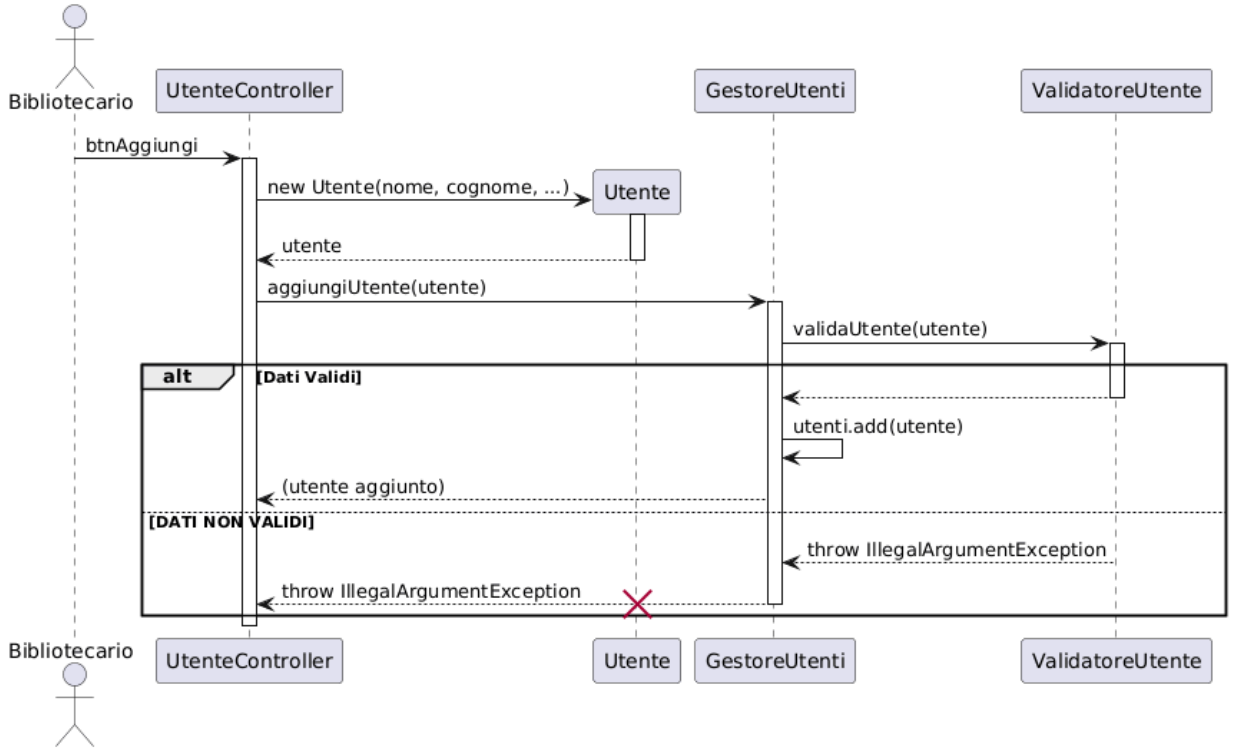
MODIFICA DI UN LIBRO



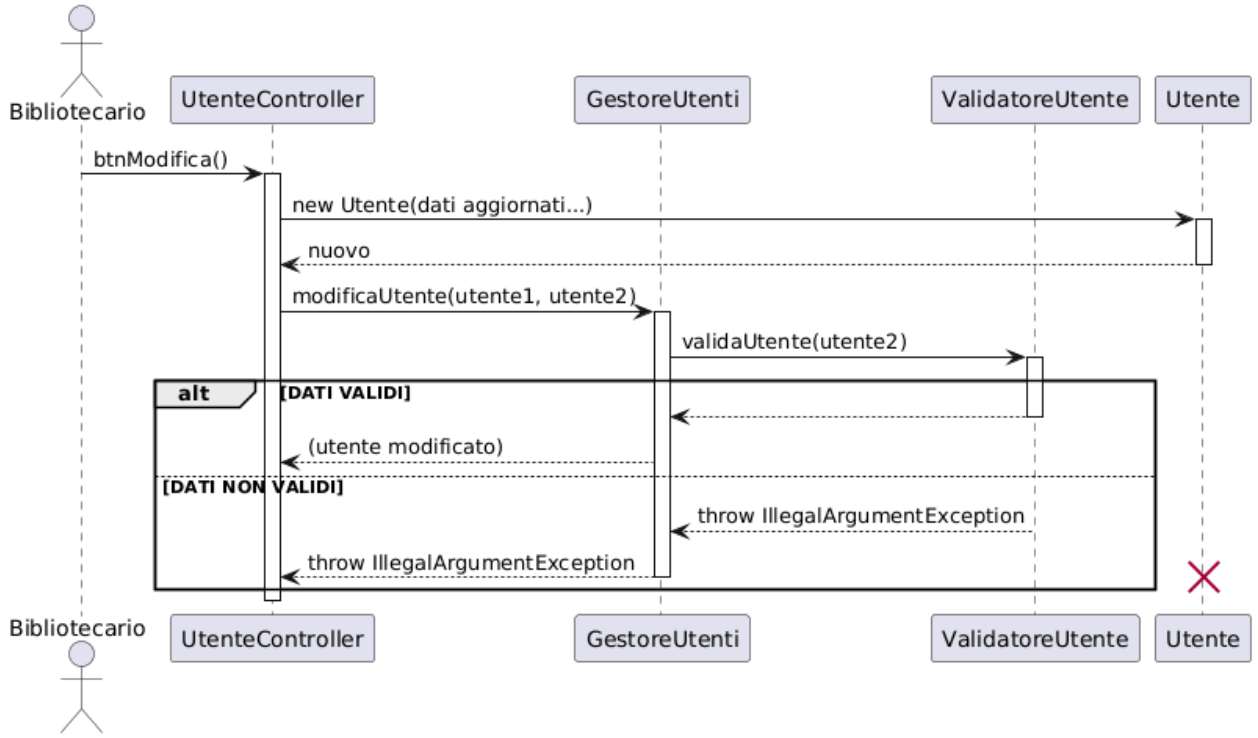
ELIMINAZIONE DI UN LIBRO



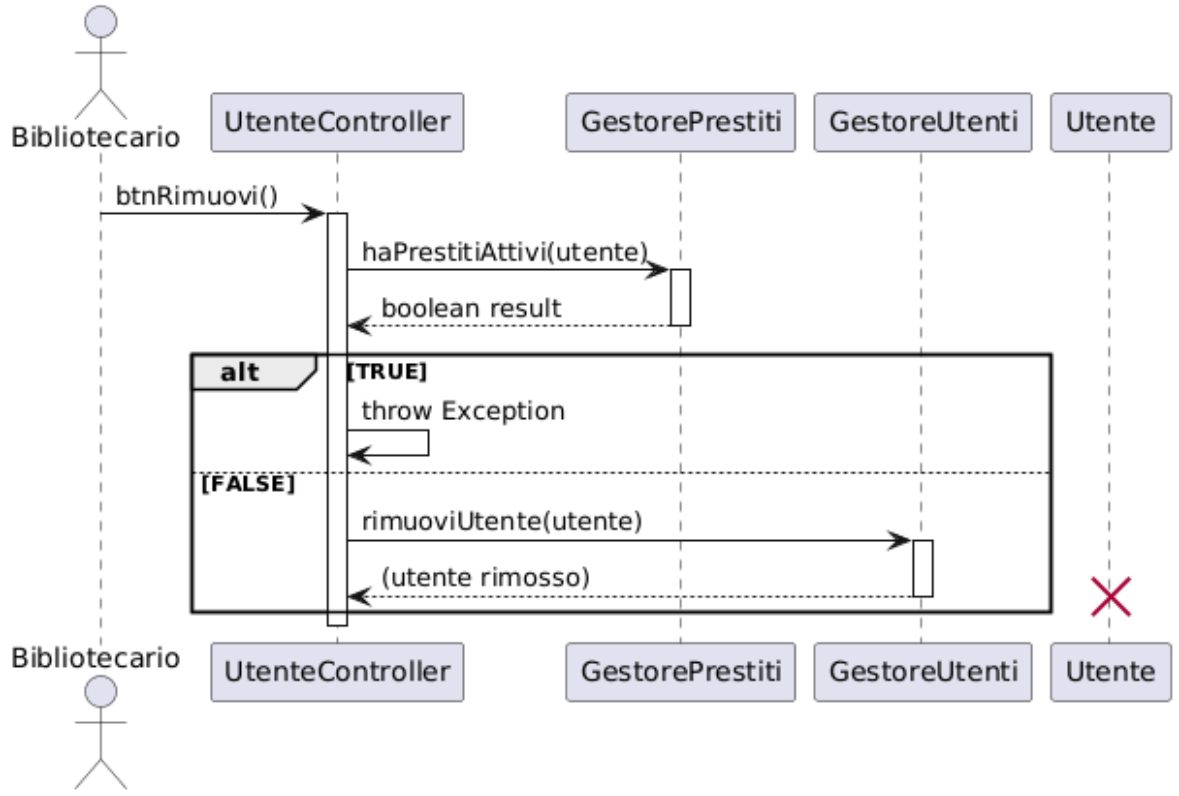
AGGIUNTA DI UN NUOVO UTENTE



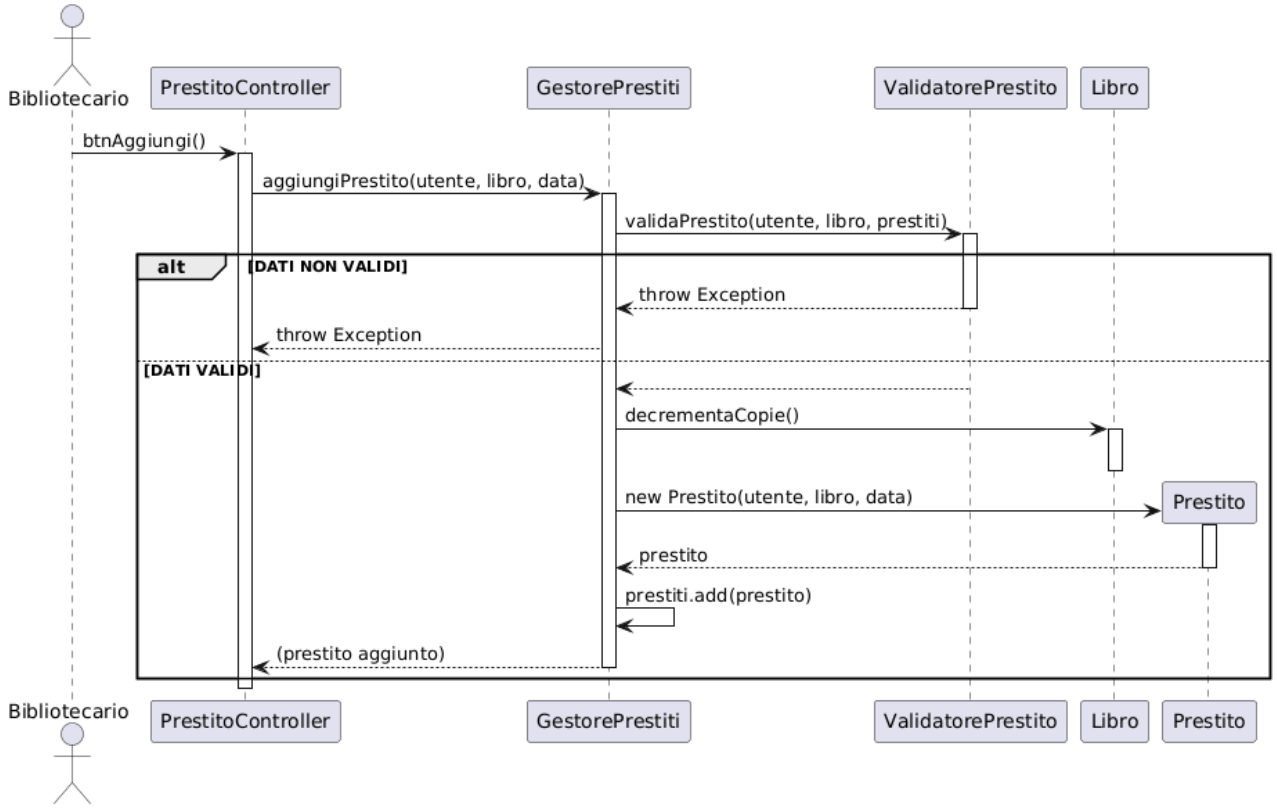
MODIFICA DI UN UTENTE



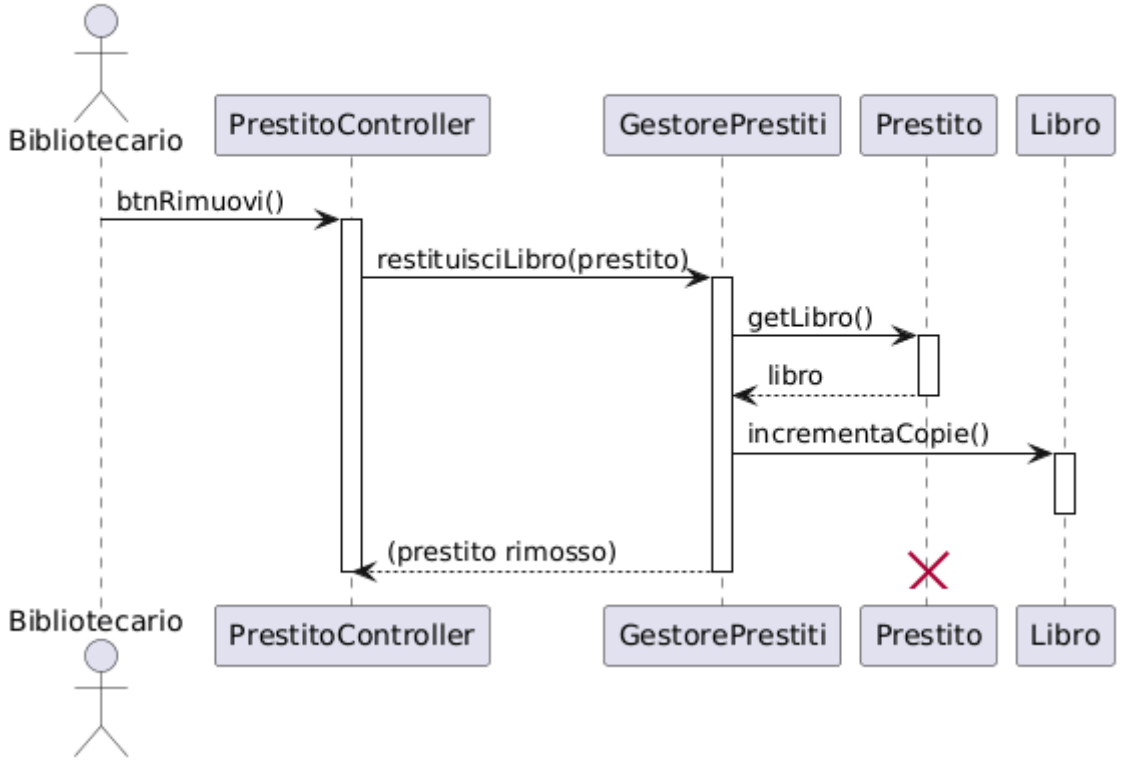
ELIMINAZIONE DI UN UTENTE



REGISTRAZIONE DI UN PRESTITO



ELIMINAZIONE DI UN PRESTITO



SALVATAGGIO SU FILE