Università degli Studi di Salerno

Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di Laurea Triennale in Ingegneria Informatica

Ingegneria del Software

Gestione di una Biblioteca

Analisi Architetturale

Docenti:
Pasquale Foggia
Nicola Capuano

Studenti:
Gaetano Alessio Forino
Lucia Canzolino
Francesco Pio De Piano
Domenico Chiacchio

Anno Accademico 2025/2026

Gruppo 16

Indice

1 Introduzione

Il presente documento ha lo scopo di illustrare le scelte progettuali e l'architettura del sistema software **Gestione di una Biblioteca** realizzato dal gruppo 16 del corso di Ingegneria del Software, A.A. 2025/2026. Il sistema è un applicativo desktop, scritto in Java, per la gestione di una biblioteca, progettato per consentire l'amministrazione efficiente di tre entità fondamentali:

- **Libri**: Gestione del catalogo (inserimento, visualizzazione, disponibilità);
- **Utenti**: Gestione dell'anagrafica degli iscritti;
- **Prestiti**: Gestione del ciclo di vita del prestito (creazione, restituzione e controllo scadenze).

L'obiettivo della *progettazione* è stato quello di realizzare, come appreso durante il corso, un sistema che si configuri come modulare, estensibile e manutenibile, ponendo, nel dettaglio, particolare attenzione al disaccoppiamento tra la logica di business e l'interfaccia utente, affinchè si possano, in futuro, implementare cambiamenti, ove necessario, in maniera semplice e rapida.

1.1 Architettura del sistema

L'architettura del sistema segue il pattern architetturale MVC (Model-View-Controller), appreso nel corso di *Programmazione ad Oggetti*.

Questa scelta è stata dettata dalla necessità di separare in modo netto le *responsabilità* dei componenti software, adoperando, cioè, il principio di Separation of Concerns appreso nelle lezioni del corso di *Ingegneria del Software*, facilitando, dunque, la manutenzione e il testing dell'intero sistema.

Seguendo il pattern architetturale MVC, è possibile suddividere il sistema realizzato in diverse macro-categorie logiche:

- **Model (Modello)**: Esso rappresenta il componente che si occupa della logica di business e dei dati del sistema. Non possiede alcuna conoscenza dell'interfaccia grafica.
- **View (Vista)**: Esso rappresenta il componente dedicato alla presentazione dei dati all'utente e della cattura degli *eventi* di input, ossia delle azioni che l'utente esegue interagendo con la GUI.
- **Controller (Controllore)**: Il controller agisce da intermediario; nello specifico, egli interpreta gli input dell'utente dalla View, interroga o modifica il Model e aggiorna la View di conseguenza.

1.2 Mappatura dei Componenti - Diagramma ad Alto Livello

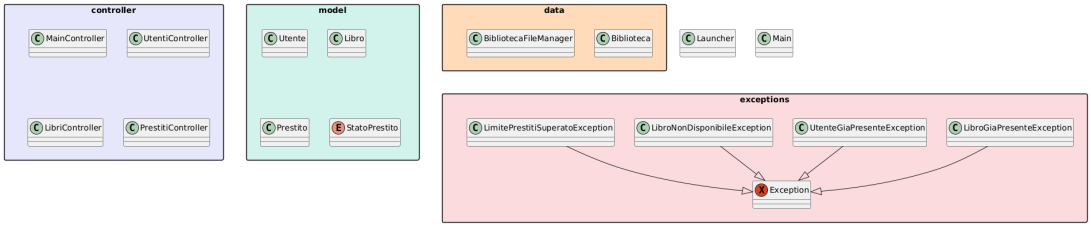

Figura 1: Diagramma ad alto livello del sistema. Il diagramma evidenzia la decomposizione in moduli del sistema.

La struttura dei *package Java* nel progetto riflette direttamente l'architettura MVC:

1.2.1 Model: `org.softeng.model` e `org.softeng.data`

Questo sottosistema gestisce lo stato dell'applicazione.

Il package `org.softeng.model` contiene le classi che modellano i dati. Nello specifico, la classe *Libro* rappresenta un volume con titolo, autore e stato (se in prestito o no), la classe *Utente* rappresenta un utente iscritto alla biblioteca e, infine, la classe *Prestito* associa un Utente a un Libro, definendo date di inizio e fine del prestito.

Il package `org.softeng.data` si compone delle classi *Biblioteca* e *BibliotecaFileManager*. Nel dettaglio:

- Biblioteca: È la classe principale che espone i metodi per manipolare i dati. Implementa la logica di business; infatti, ad esempio, effettua la verifica della disponibilità di un libro.
- BibliotecaFileManager: Gestisce la persistenza dei dati su file CSV, garantendo che il resto del sistema non debba preoccuparsi dei dettagli di I/O.

1.2.2 View: `src/main/resources/fxml`

L'interfaccia utente è realizzata mediante **JavaFX**. La struttura è definita in file .fxml dichiarativi, ottenendo, in tal modo, una separazione tra il layout e la logica di funzionamento dell'applicativo.

1.2.3 Controller: `org.softeng.controller`

Ogni vista FXML è associata a una classe Controller dedicata, che gestisce gli eventi, quali il click o l'inserimento di testo negli appositi campi.

Nel dettaglio, questo package si compone di un controller principale, detto *MainController* che gestisce la navigazione tra le diverse "schermate" dell'applicativo, e di altri "sotto-controller":

- LibriController: implementa le operazioni di creazione, lettura, aggiornamento ed eliminazione relative al catalogo bibliografico. ;
- UtentiController: Gestisce il ciclo di vita degli account utente (registrazione, modifica e cancellazione). ;

- PrestitiController: Regola la logica di business relativa alla circolazione (registrazione prestiti, restituzioni e controllo disponibilità).

1.3 Scelte Progettuali

La classe `org.softeng.data.Biblioteca` è stata realizzata affinchè si possa garantire che esista una e una sola istanza della biblioteca in memoria durante l'esecuzione del programma, in modo tale che tutti i controller (LibriController, PrestitiController e UtentiController) operino sugli stessi dati condivisi e coerenti. Infine, per quanto concerne la persistenza, con lo scopo di garantire la portabilità senza l'uso di database esterni complessi, essa è gestita su file di testo strutturati (CSV).

La classe `Biblioteca` funge da repository *"in-memory"* dell'applicazione. All'avvio, essa delega alla classe ausiliaria `BibliotecaFileManager` il compito di recuperare e caricare lo stato salvato precedentemente. La persistenza delle modifiche su disco avviene tramite salvataggio esplicito su richiesta dell'utente, garantendo così la continuità dei dati tra le diverse sessioni di utilizzo.

2 Diagrammi delle Classi

2.1 Introduzione ai diagrammi

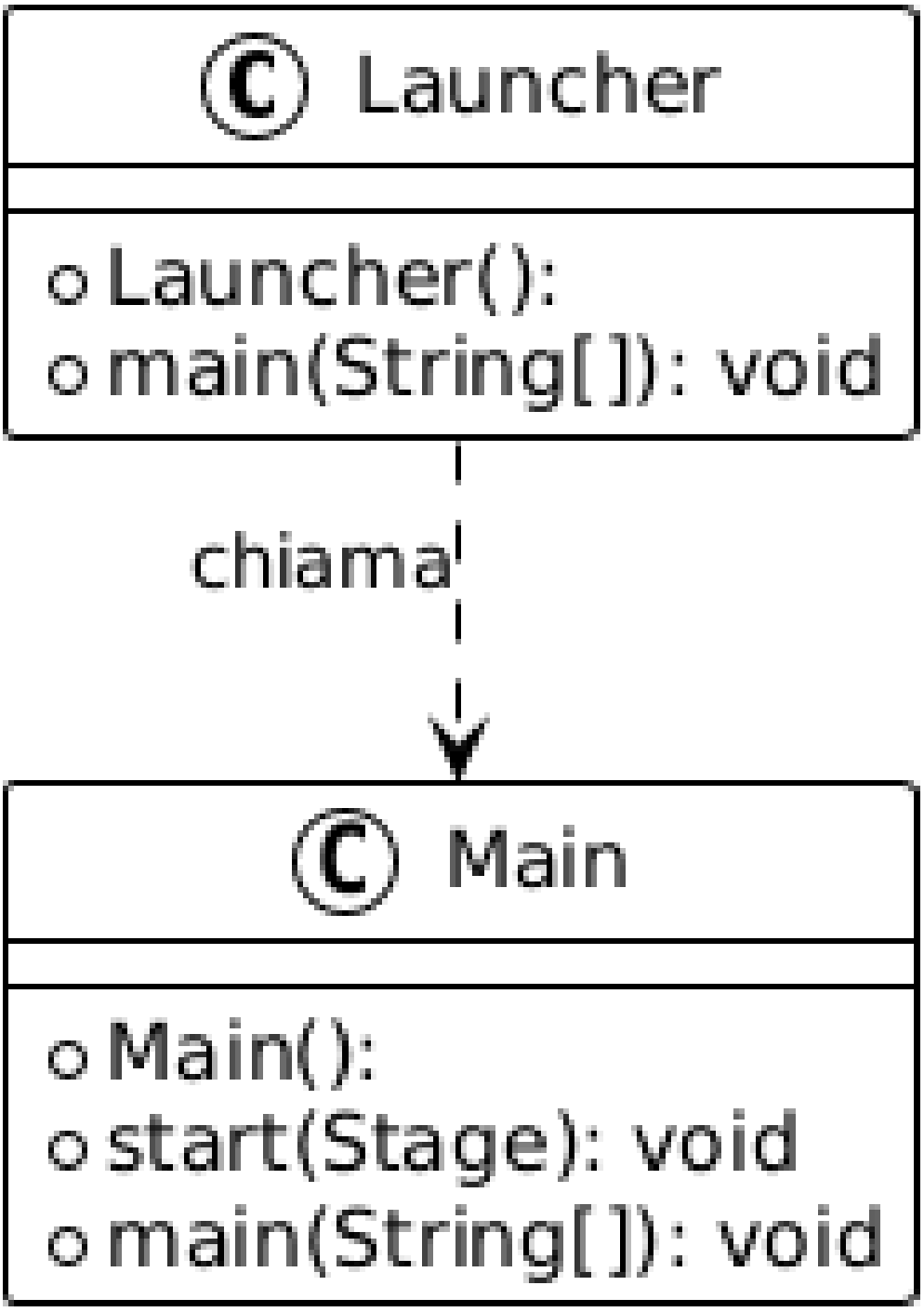

Figura 2: Diagramma delle classi: focus su Main e Launcher

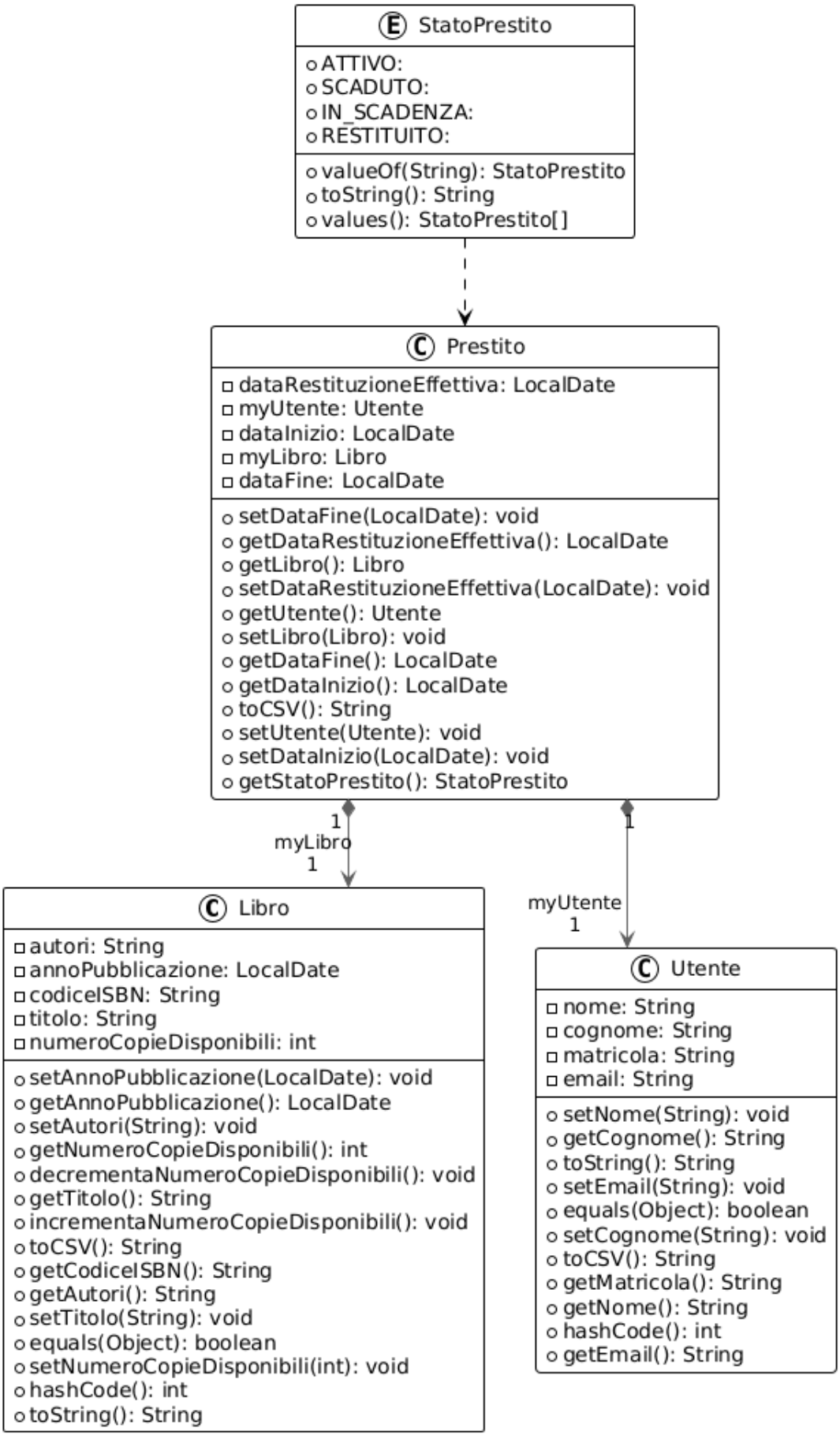

Figura 3: Diagramma delle classi: focus sul package model

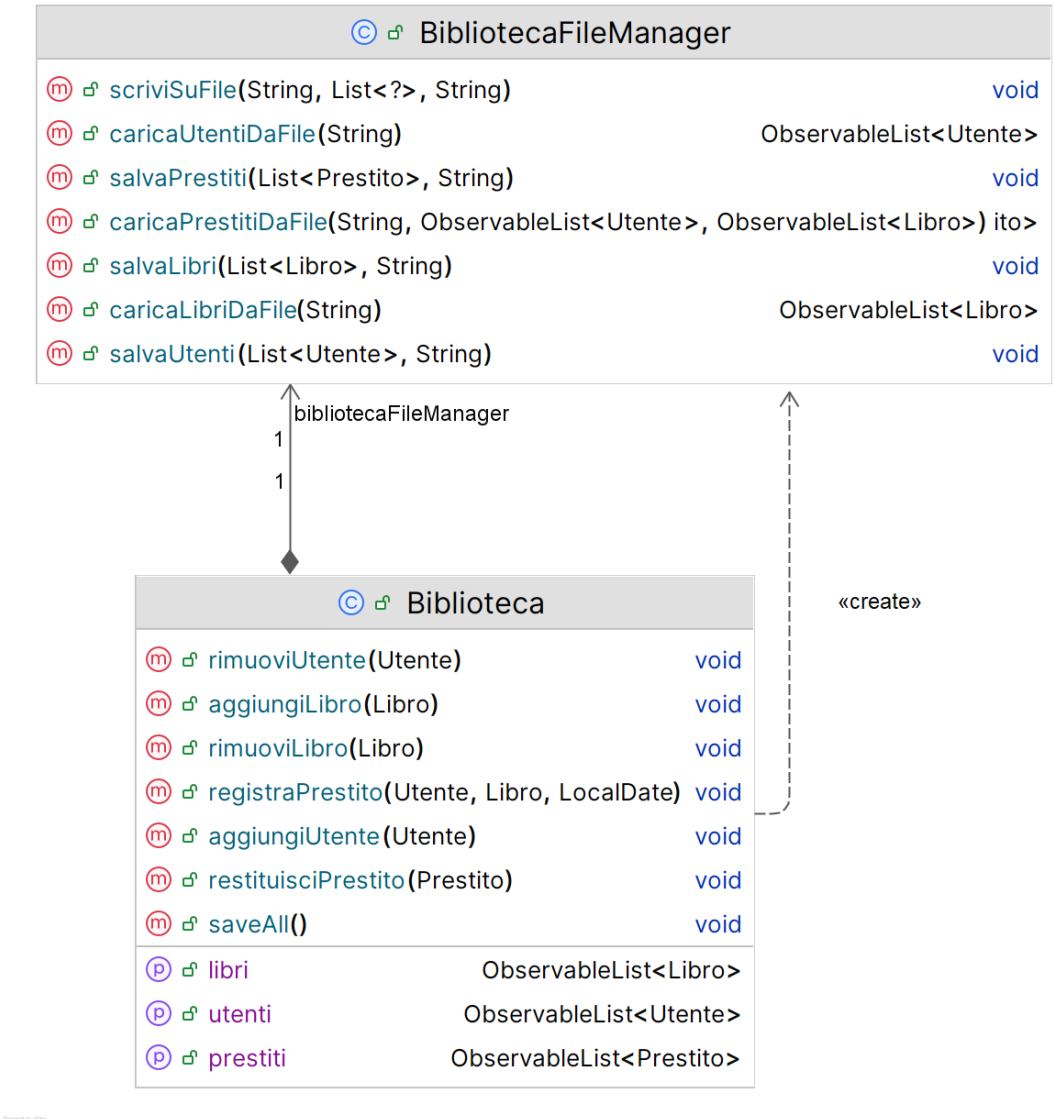

Figura 4: Diagramma delle classi: focus sul package data

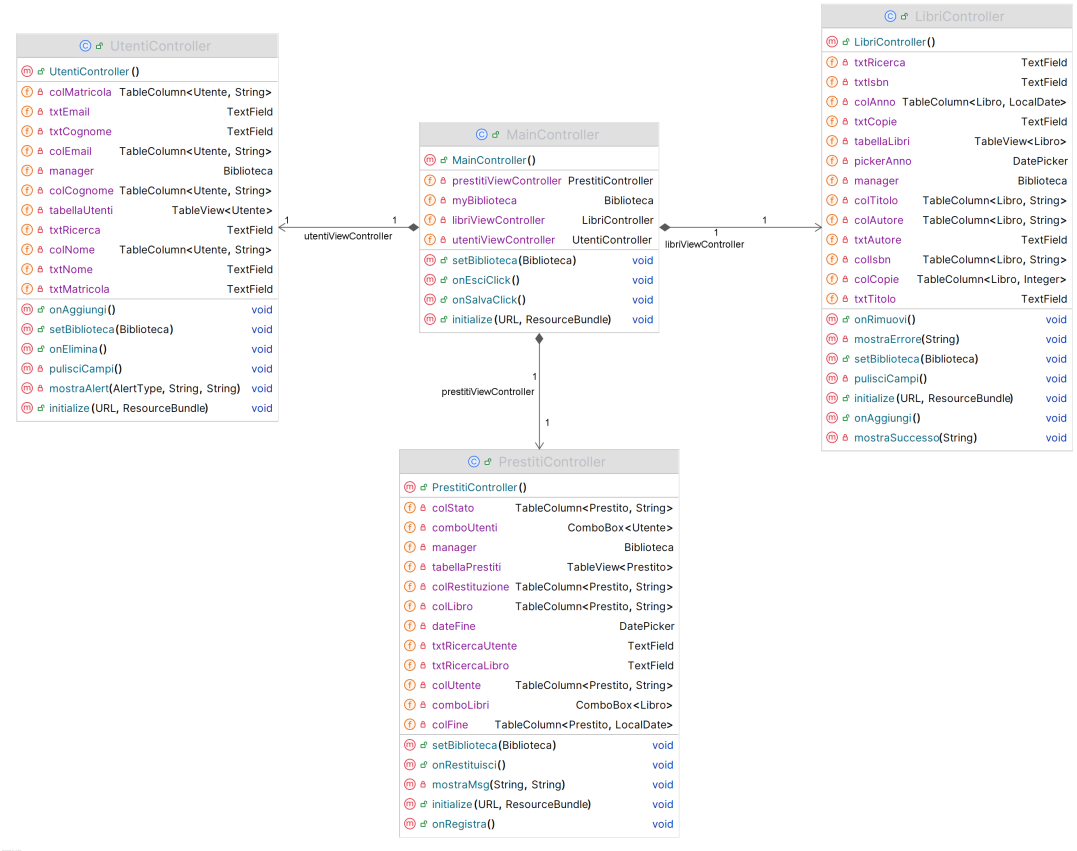

Figura 5: Diagramma delle classi: focus sul package controller

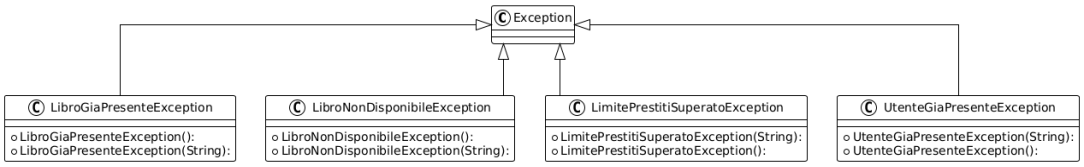

Figura 6: Diagramma delle classi: focus sul package exceptions

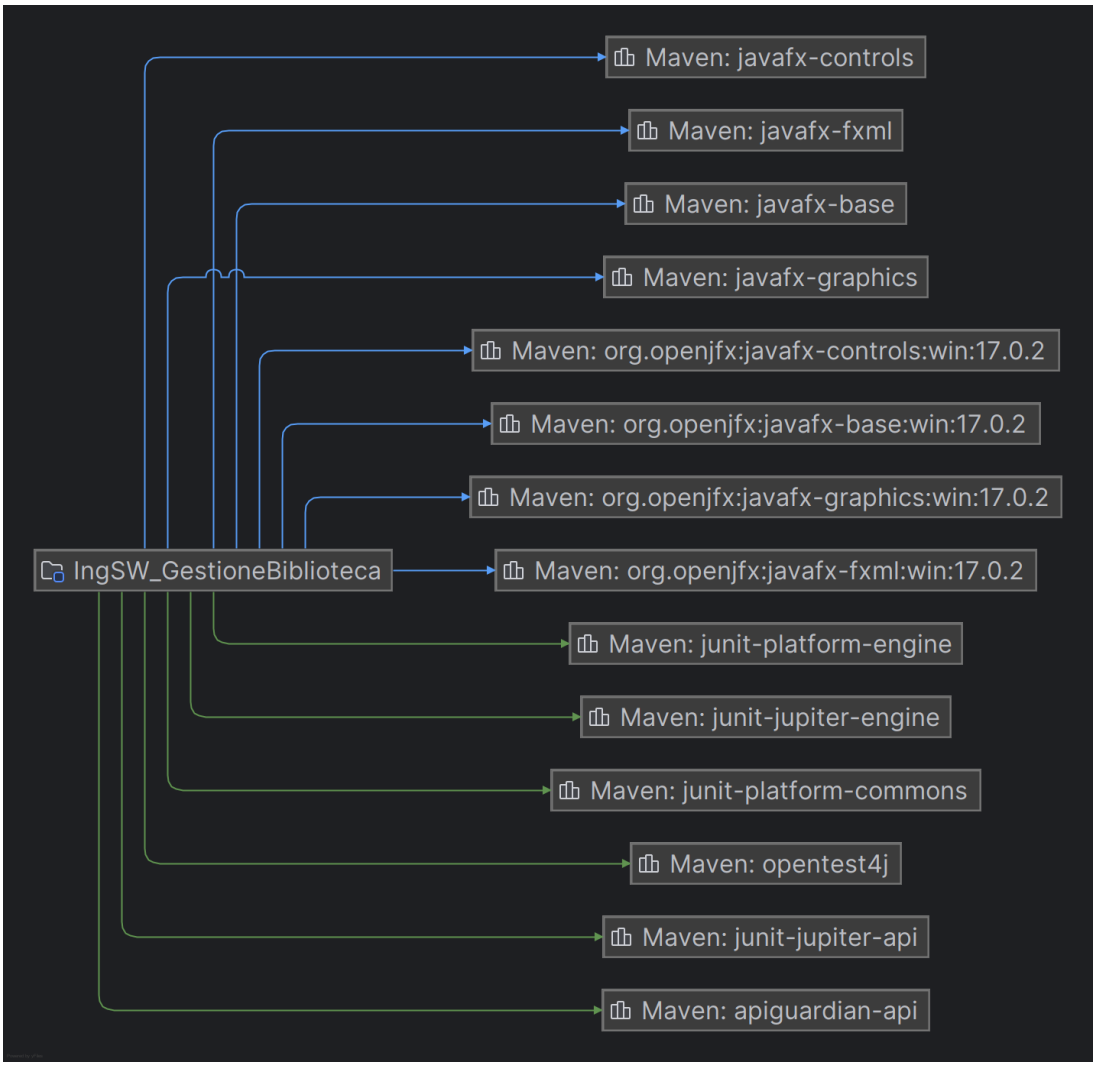

Figura 7: Diagramma delle dipendenze

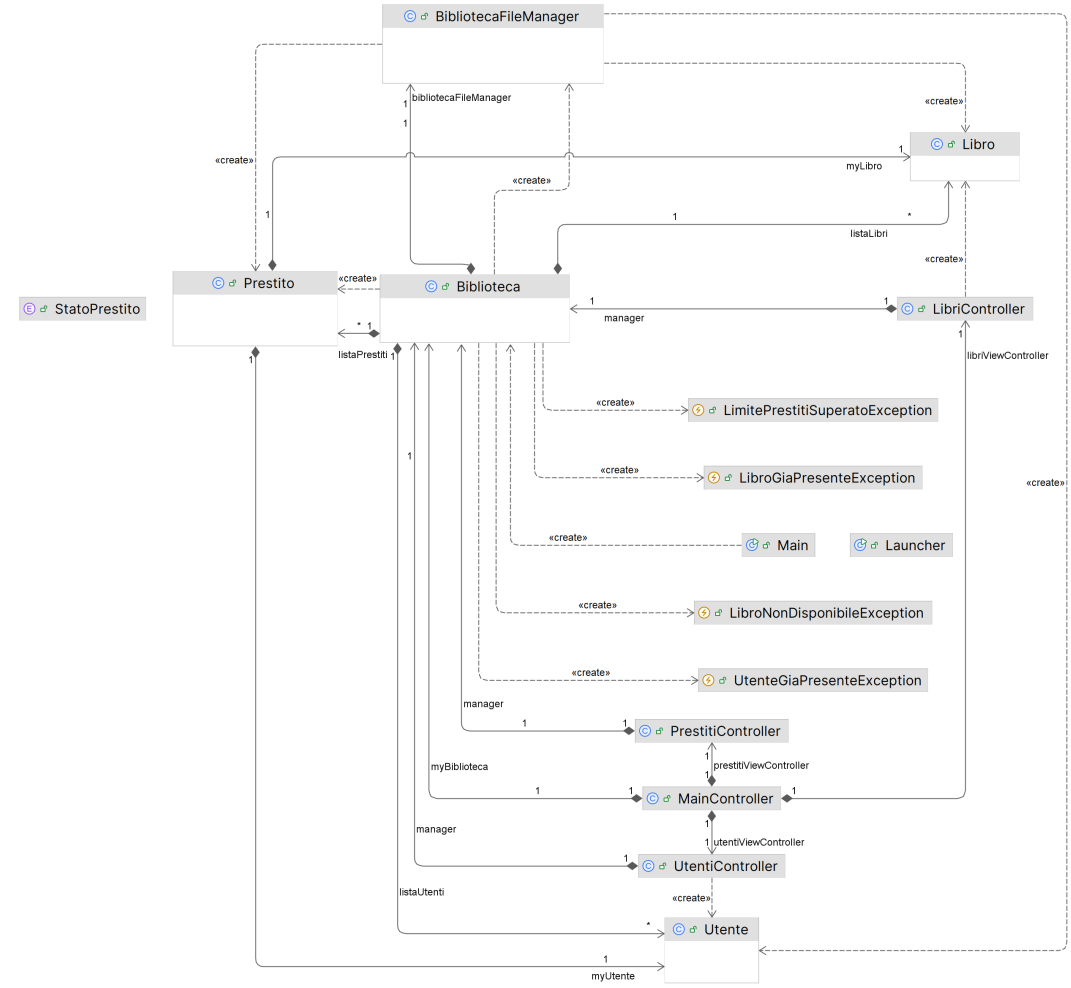

Figura 8: Diagramma delle classi dell'intero sistema: vista con dettagli

Il diagramma delle classi in esame descrive, dunque, la struttura del sistema software per la gestione bibliotecaria, realizzato in linguaggio Java e basato sul framework grafico JavaFX.

L'architettura del progetto è stata concepita seguendo il pattern Model-View-Controller (MVC), al fine di garantire una netta separazione tra la logica di business, la gestione dei dati e l'interazione con l'utente.

Da una prima analisi del diagramma, emergono tre macro-componenti fondamentali che collaborano per il funzionamento del sistema:

- Il Modello di Dominio (Model): La parte "principale" del sistema è rappresentata dalla classe Biblioteca; essa centralizza la gestione dello stato dell'applicazione mantenendo le liste osservabili (ObservableList) delle tre entità principali: Libro, Utente e Prestito, che compongono il package *Model*.

  Queste entità sono modellate come classi contenenti i metodi necessari per l'accesso e la manipolazione dei dati, inclusa la logica di validazione interna (come la gestione delle copie disponibili in Libro o lo stato del prestito tramite l'enumerativo StatoPrestito).

- Il Livello di Controllo (Controller): L'interazione con l'interfaccia grafica è gestita da una gerarchia di controller.

Il MainController funge da orchestratore principale, mentre controller specifici (LibriController, UtentiController, PrestitiController) gestiscono le rispettive viste di dettaglio.

Nel dettaglio, ogni controller possiede un riferimento all'istanza condivisa di Biblioteca, permettendo a tutte le viste di operare in modo coerente sullo stesso set di dati.

I metodi esposti dai controller (es. onAggiungi, onElimina, initialize) permettono una gestione degli eventi granulare e guidata dalle interazioni dell'utente, tipica, in sostanza, di un'applicativo realizzato mediante JavaFX.

- Persistenza e Gestione Errori: Per massimizzare la coesione del software, la responsabilità del salvataggio e caricamento dei dati è stata delegata alla classe BibliotecaFileManager, separando così la logica di I/O dalla logica di dominio.

  Il sistema implementa inoltre una gestione robusta delle eccezioni tramite classi personalizzate (es. LibroNonDisponibileException, LimitePrestitiSuperatoException), che permettono di intercettare e gestire errori specifici nell'ambito del dominio bibliotecario, garantendo l'integrità dei dati e un feedback preciso all'utente.

In sintesi, l'architettura proposta mira a un basso accoppiamento tra le componenti visive e il modello dati, favorendo così la manutenibilità e l'estendibilità futura del progetto.

2.2 Package Model

Il package `Model` costituisce il nucleo informativo dell'applicazione. Esso contiene le classi che mappano le entità del mondo reale. La progettazione di queste classi segue il principio dell'incapsulamento dei dati.

Di seguito si analizzano le tre classi cardine: `Libro`, `Utente` e `Prestito`.

2.2.1 Classe `Libro`:

La classe `Libro` rappresenta l'unità fondamentale del patrimonio bibliotecario. Oltre ai classici attributi descrittivi (`titolo`, `autori`, `annoPubblicazione`), la classe gioca un ruolo attivo nella gestione della disponibilità.

- **Identità Univoca:** L'attributo `codiceISBN` funge da chiave naturale per l'identificazione univoca del volume. I metodi `equals()` e `hashCode()` sono stati sovrascritti basandosi su questo campo, garantendo che non possano esistere duplicati logici all'interno delle collezioni.

- **Gestione della Disponibilità (Incapsulamento):** Un aspetto critico è la gestione dell'attributo `numeroCopieDisponibili`. Invece di esporre un semplice setter pubblico che permetterebbe di impostare valori arbitrari, la classe fornisce metodi di *business* specifici:

```
public void incrementaNumeroCopieDisponibili();
public void decrementaNumeroCopieDisponibili();
```

Questo approccio protegge l'*invariante di classe*: le copie non possono scendere sotto lo zero. Nel dettaglio, la logica di controllo viene delegata al chiamante, che verifica il numero di copie effettivamente disponibili prima di effettuare il decremento.

- **Esportazione in file CSV:** Il metodo `toCSV()` predispone l'oggetto per la persistenza, trasformando lo stato interno in una stringa formattata, pronta per essere scritta su file dal `BibliotecaFileManager`.

2.2.2 Classe `Utente`:

La classe `Utente` modella gli iscritti alla biblioteca. Sebbene strutturalmente più semplice, essa è fondamentale per l'associazione dei prestiti.

- **Dati Anagrafici:** Memorizza informazioni essenziali come `nome`, `cognome`, `email` e `matricola`.
- **Ruolo della Matricola:** L'attributo `matricola` agisce come identificatore primario. È essenziale per distinguere omonimie e per collegare univocamente un prestito a una persona fisica.
- **Interoperabilità:** Anche qui, l'implementazione corretta di `toString()` facilita il debugging e la visualizzazione rapida nelle componenti UI (es. nelle ComboBox o nelle liste di selezione).

2.2.3 Classe `Prestito`:

La classe `Prestito` è la più complessa dal punto di vista concettuale, in quanto è una classe di natura *associativa*.

- **Associazioni:** La classe non duplica i dati di libri e utenti, ma mantiene i riferimenti agli oggetti originali:

  ```
  private Libro libro;
  private Utente utente;
  ```

  Questo garantisce la coerenza dei dati: se il titolo di un libro viene corretto nell'oggetto `Libro`, la modifica si riflette automaticamente in tutti i prestiti che puntano a quell'oggetto (poiché Java passa gli oggetti per riferimento).
- **Gestione Temporale:** L'uso della classe `java.time.LocalDate` per i campi `dataInizio`, `dataFine` e `dataRestituzioneEffettiva` rappresenta una scelta moderna e robusta rispetto, ad esempio, ad una vecchia implementazione del progetto che adoperava la classe `Date`, facilitando calcoli sulle scadenze e sui ritardi.
- **Stato del Processo (Enum):** L'attributo `statoPrestito` utilizza l'enumerazione `StatoPrestito` (valori visibili nel diagramma come `IN_CORSO`, ecc.). Questa scelta rende impossibile assegnare uno stato non valido al prestito, garantendo così robustezza e coerenza all'intero sistema.

In conclusione, queste tre classi non sono meri contenitori di dati, ma oggetti *"intelligenti"* la cui proprietà cardine è quella di incapsulare regole di validazione e logiche di interazione fondamentali per la stabilità e funzionamento del sistema.

2.3 Package Controller

Il package `Controller` implementa la logica di presentazione e mediazione del sistema. In accordo con il pattern MVC, queste classi non contengono i dati (delegati al Model) né la definizione grafica (delegata alle viste FXML), ma gestiscono la reazione agli eventi generati dall'utente e coordinano l'aggiornamento della UI.

Di seguito sono analizzate le quattro classi principali evidenziate nel diagramma.

2.3.1 MainController:

La classe `MainController` agisce come "controllore di alto livello" e "punto di snodo" per la navigazione e per le operazioni globali.

- **Gestione del Ciclo di Vita Globale:** I metodi `onSalvaClick()` e `onEsciClick()` indicano che questo controller gestisce la barra dei menu o i comandi generali dell'applicazione. In particolare, `onSalvaClick()` invoca il metodo `saveAll()` della classe `Biblioteca`, innescando la persistenza dei dati su file per tutte le entità.
- **Orchestrazione delle Viste:** Le associazioni nel diagramma collegano il `MainController` ai sotto-controller (`libriViewController`, `utentiViewController`, `prestitiViewController`). Dunque, è lecito affermare che il `MainController` è responsabile dell'inizializzazione della dashboard principale e dell'iniezione dell'istanza del Model (`Biblioteca`) nei controller figli.

2.3.2 LibriController:

Questa classe è specializzata nella gestione relativa all'entità `Libro`.

- **Manipolazione Dati:** Il metodo `onAggiungi()` raccoglie l'input dai campi di testo (titolo, autore, ISBN, copie), istanzia un nuovo oggetto `Libro` e lo passa al Model. Il metodo `onRimuovi()` gestisce l'eliminazione, recuperando l'elemento selezionato nella `TableView`.
- **Feedback Utente e Pulizia:** Metodi come `mostraSuccesso(String)` e `mostraErrore(String)` indicano un'attenzione alla UX, incapsulando la logica di visualizzazione di alert o messaggi di stato. Il metodo `pulisciCampi()` garantisce che il form venga resettato dopo ogni operazione, prevenendo inserimenti duplicati accidentali.

2.3.3 UtentiController:

Analogamente al controller dei libri, `UtentiController` gestisce l'interfaccia per gli iscritti.

- **Operazioni di Dominio:** I metodi `onAggiungi()` e `onElimina()` interfacciano la vista con i metodi corrispondenti della classe `Biblioteca`. È qui che vengono intercettate eccezioni specifiche come `UtenteGiaPresenteException`, convertendole in un avviso visivo tramite `mostraAlert(AlertType, String, String)`.
- **Validazione:** La presenza di `mostraAlert` con parametri dettagliati permette una gestione granulare degli errori di validazione (es. campi vuoti, formato email non valido, matricola duplicata).

2.3.4 PrestitiController:

Il `PrestitiController` è il componente più complesso in termini di logica relazionale, poiché deve coordinare oggetti di tipo diverso (`Utente` e `Libro`).

- **Registrazione e Restituzione:** I metodi `onRegistra()` e `onRestituisci()` non si limitano ad aggiungere o rimuovere righe. Essi devono invocare la logica di business che verifica:
  1. La disponibilità delle copie del libro (tramite eccezione `LibroNonDisponibileException`).
  2. Il rispetto del limite prestiti per l'utente (tramite eccezione `LimitePrestitiSuperatoExcepti`
- **Interazione col Model:** A differenza degli altri controller che gestiscono entità singole, questo controller deve interagire con le liste di Utenti e Libri (per popolare i menu a tendina/ComboBox) oltre che con la lista dei Prestiti.

2.4 Package Data

2.4.1 Analisi del Package Data: Stato e Persistenza

Il package `Data` rappresenta il cuore logico del sistema. Esso non si limita a contenere le strutture dati, ma orchestra le operazioni di dominio e garantisce la persistenza delle informazioni tra le diverse esecuzioni del software.

Le due classi principali che compongono questo modulo, `Biblioteca` e `BibliotecaFileManager`, collaborano strettamente separando nettamente la gestione dello stato in memoria (RAM) dalla gestione dello stato su disco (File System).

2.4.2 Classe `Biblioteca`:

La classe `Biblioteca` agisce come **punto di accesso unificato** per l'intera logica di business. Essa maschera la complessità delle relazioni tra oggetti, offrendo ai Controller un'interfaccia di alto livello molto semplice e, allo stesso tempo, coerente.

- **Gestione dello Stato Reattivo (ObservableList):** Per quanto concerne le strutture dati, dopo un'attenta analisi degli attributi, emerge una scelta architetturale fondamentale per l'integrazione con JavaFX:

  ```
  private ObservableList<Libro> libri;
  private ObservableList<Utente> utenti;
  private ObservableList<Prestito> prestiti;
  ```

  L'utilizzo di `ObservableList` trasforma il modello dati in un soggetto attivo. Grazie al pattern *Observer* implementato nativamente da queste collezioni, qualsiasi modifica apportata ai dati (es. tramite `aggiungiLibro`) notifica automaticamente le componenti della GUI collegate (*Data Binding*), eliminando la necessità di codice per l'aggiornamento manuale delle tabelle.

- **Centralizzazione della Business Logic:** La classe non è un semplice contenitore passivo. Metodi come `registraPrestito` e `restituisciPrestito` incapsulano le regole cardine del sistema:

  - Verifica della disponibilità delle copie (collaborazione con la classe `Libro`).
  - Controllo dei vincoli utente (collaborazione con la classe `Utente`).
  - Gestione delle eccezioni di dominio in caso di violazione delle regole.

- **Coordinamento della Persistenza:** Il metodo `saveAll()` funge da "trigger". Quando invocato (solitamente alla chiusura o al salvataggio manuale), la `Biblioteca` non scrive direttamente su disco, ma delega il compito al componente specializzato, mantenendo il proprio codice pulito e focalizzato sulle regole di gestione.

2.4.3 Classe `BibliotecaFileManager`:

La classe `BibliotecaFileManager` è stata introdotta per rispettare il principio di **Alta Coesione (High Cohesion)** . La logica di "come salvare i dati" (formattazione, gestione file, parsing) è completamente separata dalla logica di "cosa sono i dati".

- **Persistenza su file .csv:** La classe espone metodi speculari di caricamento (`caricaLibriDaFile`, ecc.) e salvataggio (`salvaLibri`, ecc.). Dopo un'analisi effettuata sulle firme dei metodi, è possivile notare come la classe manipola stringhe e liste convertendo oggetti in un formato testuale e viceversa.

- **Robustezza e Helper Methods:** La presenza del metodo `scriviSuFile(String path, List<?> list, String header)` indica un design progettato con il fine di evitare la duplicazione del codice (DRY - Don't Repeat Yourself). Questo metodo, adoperando la wildcard `<?>`, gestisce l'apertura degli stream di output e la gestione delle eccezioni di I/O (es. `IOException`) in un unico punto, rendendo il sistema più facile da manutenere.

2.4.4 Relazione tra i Componenti

La relazione tra `Biblioteca` e `BibliotecaFileManager` è di tipo associativo direzionale: la `Biblioteca` "usa" il `FileManager`. Questa struttura permette, in un'ottica di evoluzione futura, di modificare il meccanismo di persistenza (ad esempio passando da CSV a un Database vero e proprio) modificando esclusivamente la classe `BibliotecaFileManager`, senza dover modificare il codice della classe `Biblioteca` o dei Controller.

3 Diagrammi di Sequenza

3.1 Introduzione ai Diagrammi di Sequenza

3.1.1 Definizione e Contesto nell'UML

Nell'ambito dell'ingegneria del software e della modellazione orientata agli oggetti, i **Diagrammi di Sequenza** rappresentano uno degli artefatti più significativi del linguaggio UML (*Unified Modeling Language*). Essi appartengono alla famiglia dei *diagrammi di interazione* e hanno l'obiettivo primario di descrivere la dinamica del sistema focalizzandosi sull'ordine temporale dei messaggi scambiati tra le varie entità che lo compongono.

A differenza dei diagrammi delle classi, che offrono una visione statica e strutturale dell'architettura (quali classi esistono e come sono collegate), i diagrammi di sequenza, nel loro complesso, "danno vita" al sistema, mostrando come queste classi collaborano concretamente per realizzare uno specifico Caso d'Uso.

3.1.2 Struttura e Funzionamento

Il funzionamento di un diagramma di sequenza si basa su una rappresentazione bidimensionale:

- **Asse Orizzontale (Partecipanti):** In alto sono disposti gli attori (utenti umani o sistemi esterni) e gli oggetti software (classi, componenti) coinvolti nel processo. Nel contesto dell'applicativco in esame, questi corrispondono tipicamente alle Viste (`View`), ai Controllori (`Controller`) e al Modello (`Model` o `Entity`).
- **Asse Verticale (Tempo):** Il tempo scorre linearmente dall'alto verso il basso. Ogni partecipante possiede una propria *lifeline* tratteggiata che rappresenta la sua esistenza durante l'interazione.

La comunicazione avviene tramite **Messaggi** (frecce) che connettono le linee di vita:

- **Messaggi Sincroni (Frecce piene):** Indicano una chiamata a metodo in cui il chiamante attende una risposta (es. il Controller che invoca `manager.getLibri()`).
- **Messaggi di Ritorno (Frecce tratteggiate):** Rappresentano la risposta o il valore restituito al termine dell'esecuzione di un metodo.
- **Barre di Attivazione:** Rettangoli sovrapposti alla linea di vita che indicano il periodo in cui un oggetto è attivo e sta eseguendo un'elaborazione (ha il controllo del flusso).

3.1.3 Metodologia di Realizzazione: Dal Codice al Modello

Per questo progetto, i diagrammi di sequenza presentati non sono stati disegnati su basi ipotetiche, ma sono stati realizzati seguendo un rigoroso approccio di aderenza al codice sorgente (*Source-Code Driven*). Ogni diagramma è la traduzione grafica fedele della logica implementata nei file Java dell'applicativo.

La realizzazione di tali diagrammi è avvenuta analizzando passo dopo passo il flusso di esecuzione:

1. **Identificazione dell'evento "trigger":** Si parte dall'evento scatenante nell'interfaccia utente (es. click sul bottone "Registra" in `PrestitiView`).
2. **Mapping delle Chiamate:** Si traccia ogni invocazione di metodo dal Controller verso le classi del dominio (es. `Biblioteca`, `Utente`, `Prestito`).
3. **Gestione dei Flussi Alternativi:** Sono stati modellati esplicitamente i blocchi condizionali (`if/else`) e la gestione delle eccezioni (`try/catch`), evidenziando come il sistema reagisce a errori specifici (es. `LibroNonDisponibileException`) o a input non validi.

3.1.4 Scopo e Utilità nella Progettazione Software

L'inclusione di questi diagrammi nella documentazione risponde a diverse esigenze cruciali per la qualità del software:

- **Verifica della Logica di Business:** Permettono di validare visivamente se il flusso implementato soddisfa i requisiti funzionali. Ad esempio, visualizzare chiaramente che il decremento delle copie di un libro avviene *solo dopo* aver superato i controlli di disponibilità.
- **Identificazione delle Responsabilità:** Chiariscono il "chi fa cosa". Nel caso in esame, evidenziano nettamente la separazione dei compiti (Pattern MVC): la *View* gestisce l'input, il *Controller* orchestra il flusso e gestisce le eccezioni grafiche (Alert), mentre il *Model* (`Biblioteca`) detiene la logica di persistenza e i vincoli di integrità.
- **Documentazione dei Casi Limite:** A differenza di una semplice descrizione testuale, il diagramma costringe a considerare esplicitamente i rami di errore (es. cosa succede se l'utente è null? Cosa succede se il libro è già in prestito?), garantendo che nessuna eccezione rimanga non gestita.
- **Manutenibilità Futura:** Forniscono a futuri sviluppatori una mappa immediata del flusso di esecuzione, rendendo più semplice individuare dove intervenire per modifiche o debugging senza dover leggere prima tutte le righe di codice.

3.2 Diagrammi di Sequenza

In questa sezione sono riportati i diagrammi di sequenza relativi ai casi d'uso del sistema, modellati in conformità con il codice sorgente Java.

3.2.1 Gestione Libri

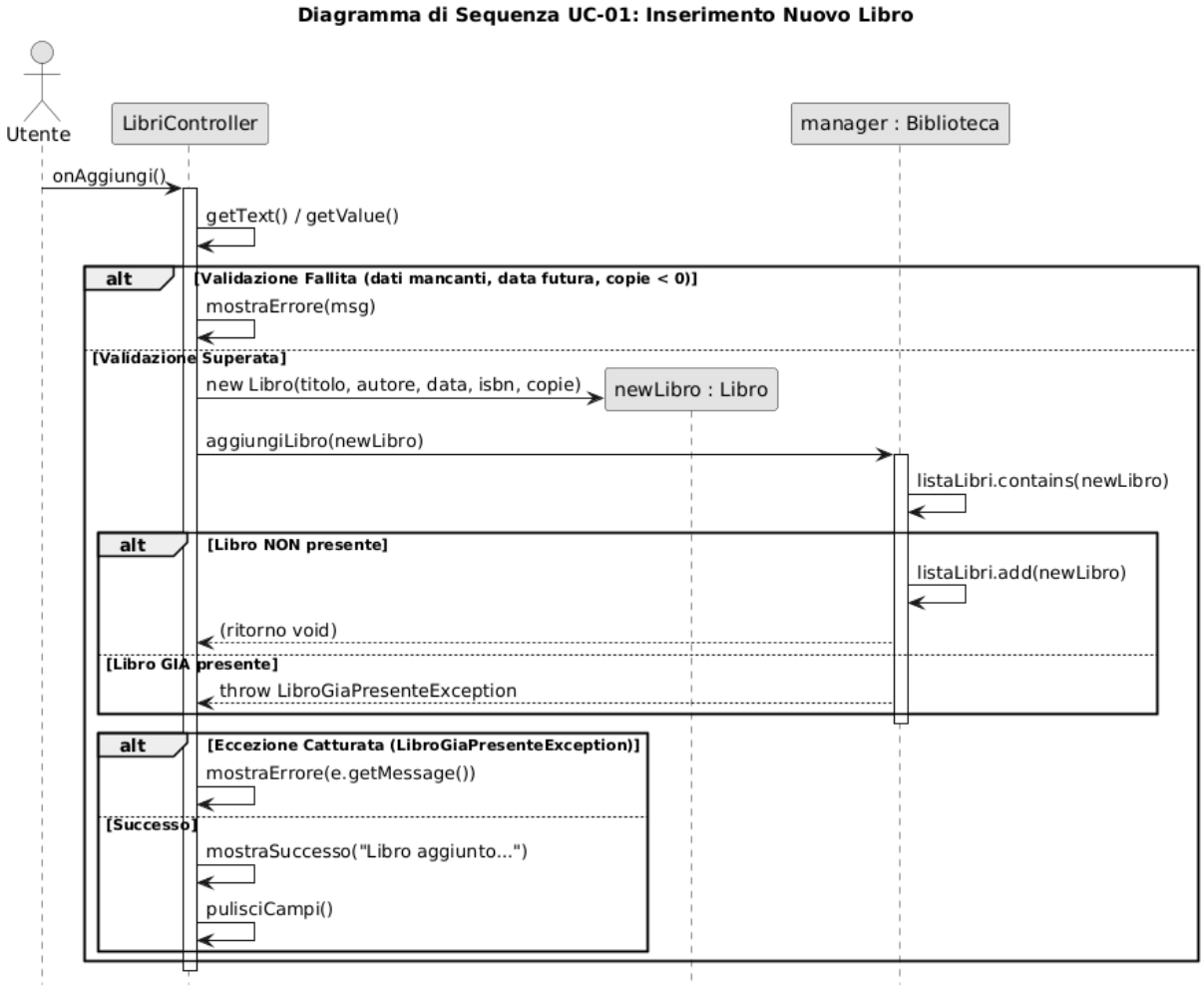

Figura 9: UC01 - Visualizzazione Catalogo Libri. Il diagramma mostra la sequenza di inizializzazione del `LibriController`, il recupero della lista dal `Model` e la configurazione della `TableView`.

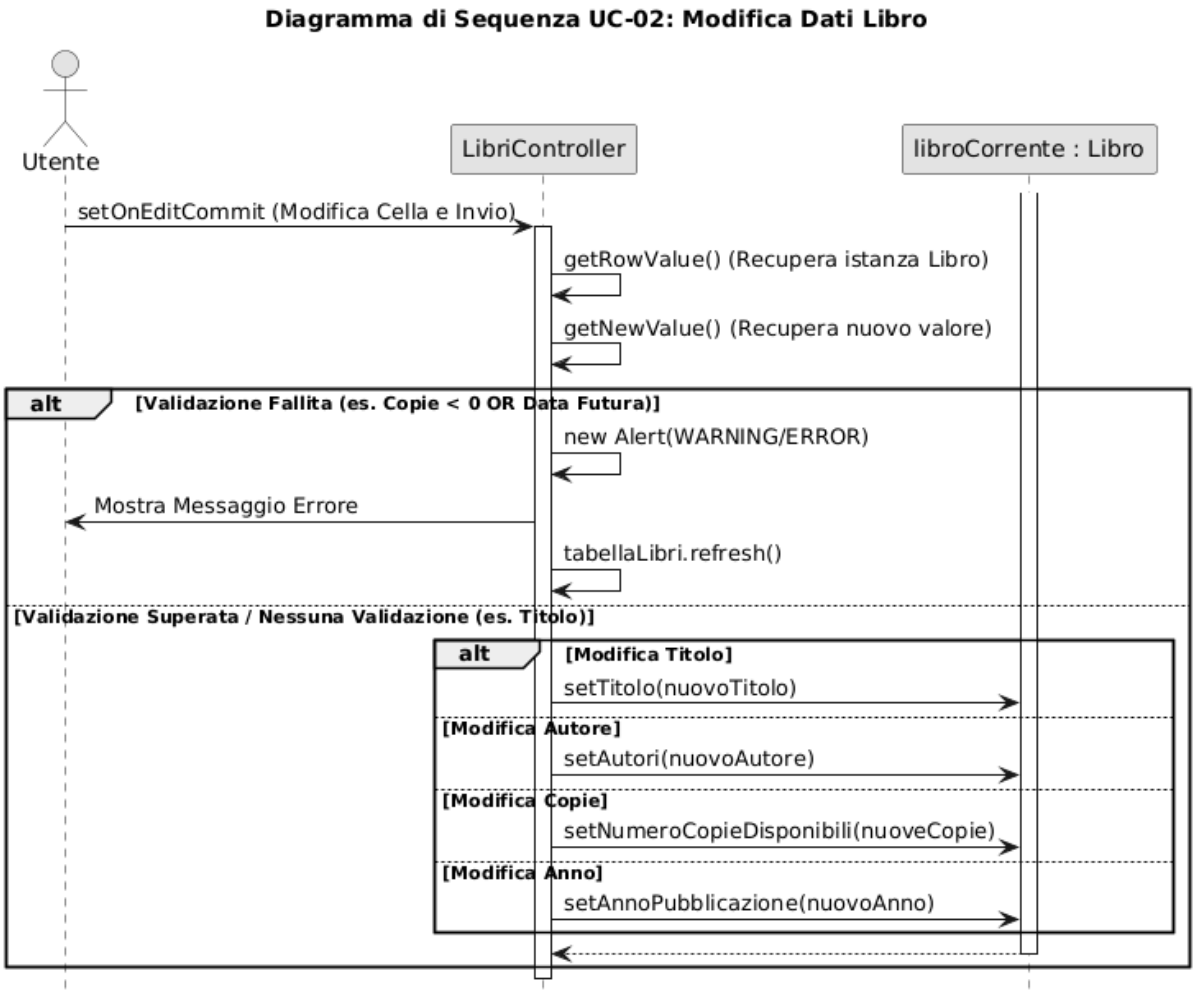

Figura 10: UC02 - Inserimento Nuovo Libro. Viene evidenziata la gestione dell'eccezione `LibroGiaPresenteException` nel caso di duplicazione del codice ISBN.

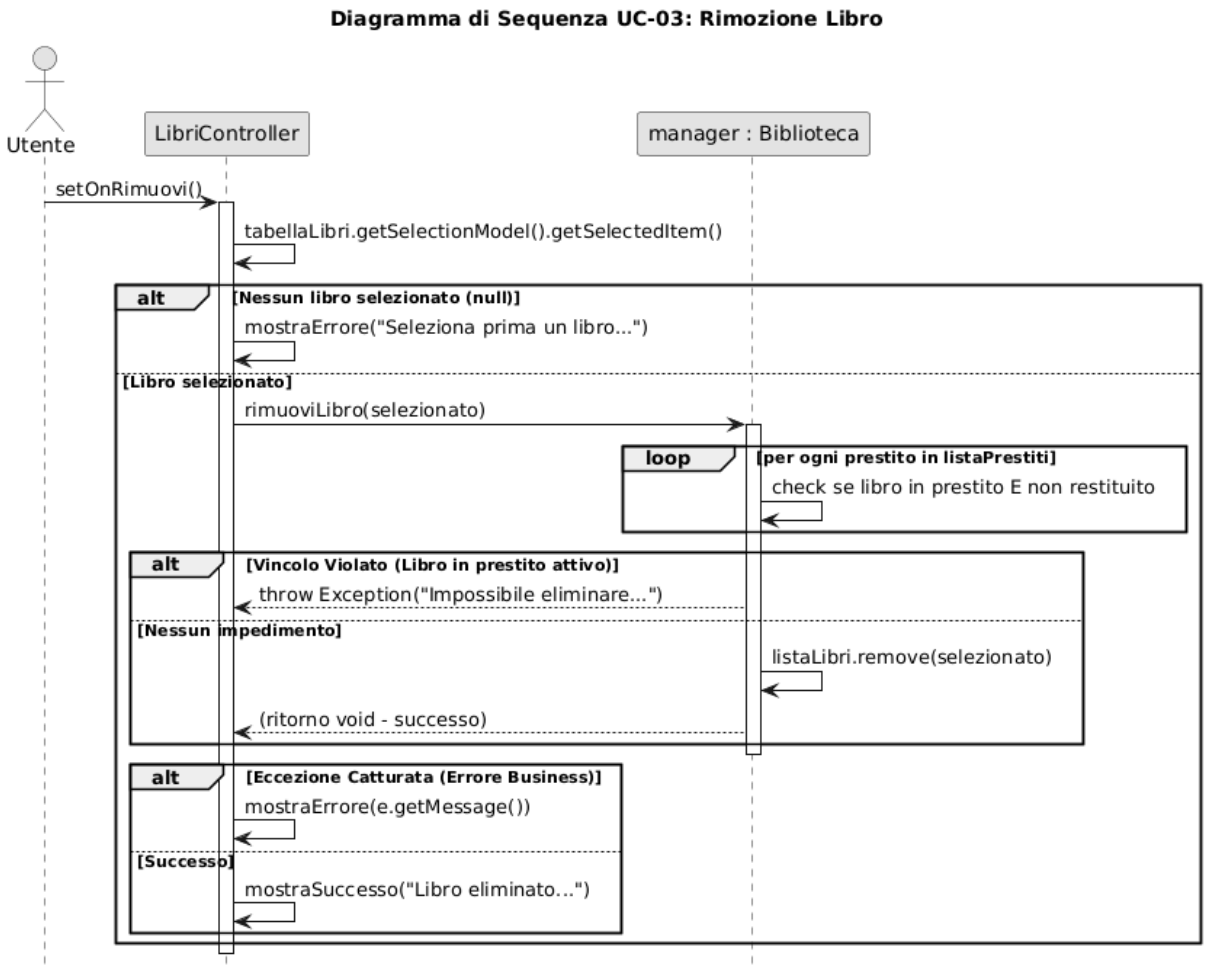

Figura 11: UC03 - Rimozione Libro. Il sistema impedisce la rimozione se il libro risulta attualmente in prestito (controllo di integrità referenziale).

3.2.2 Gestione Utenti

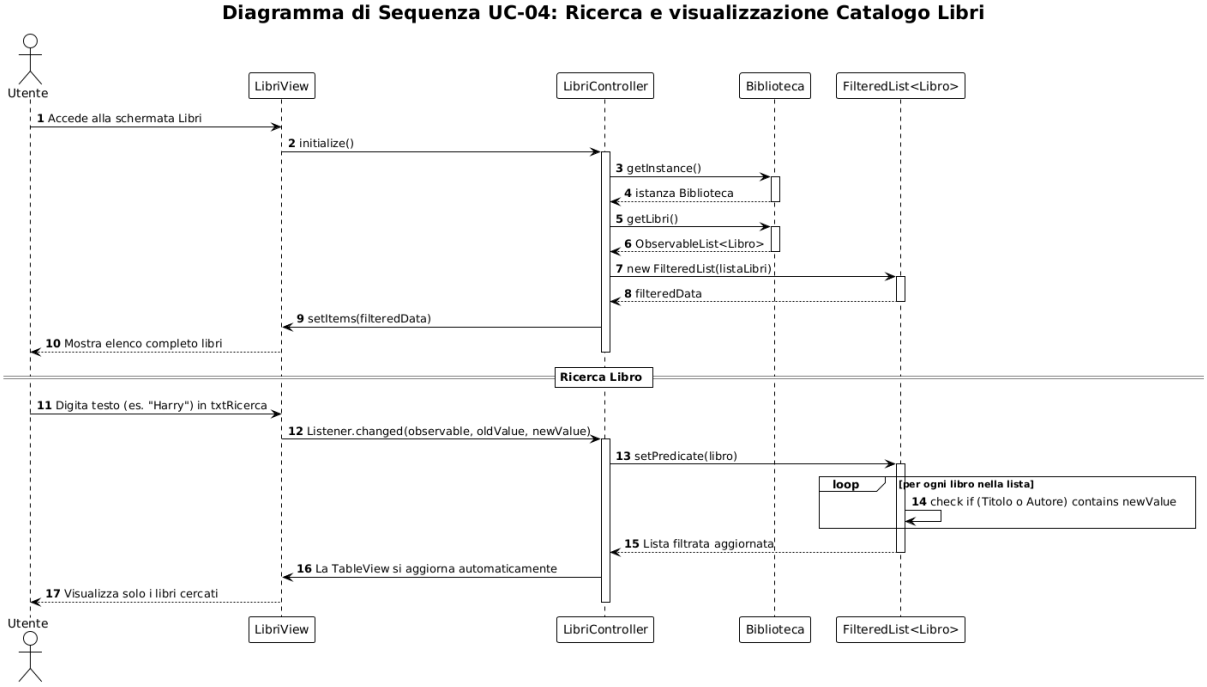

Figura 12: UC04 - Ricerca Utenti. Illustra il filtraggio dinamico (case-insensitive) su Nome, Cognome e Matricola attivato dal Listener sulla barra di ricerca.

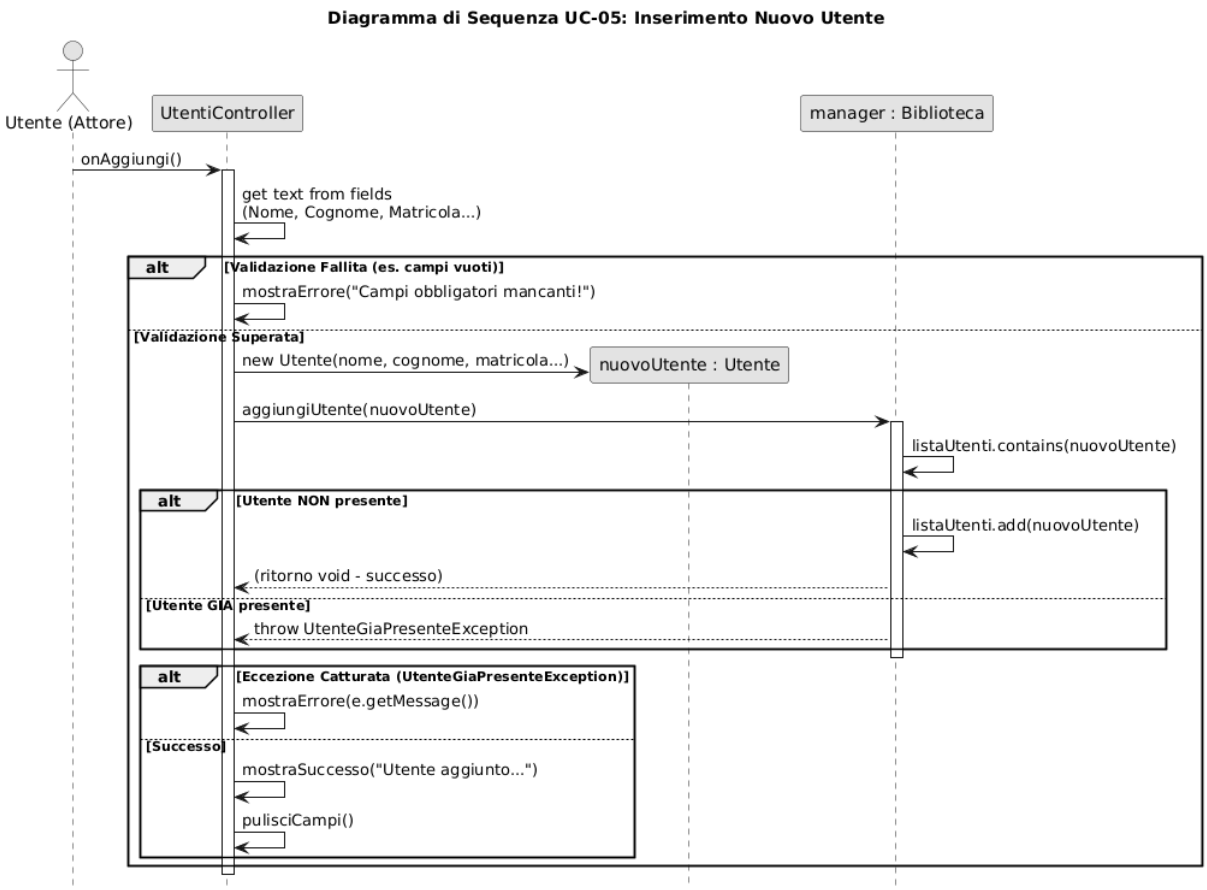

Figura 13: UC05 - Inserimento Utente. Mostra la validazione dei campi obbligatori e il controllo di univocità della matricola gestito dal `Model`.

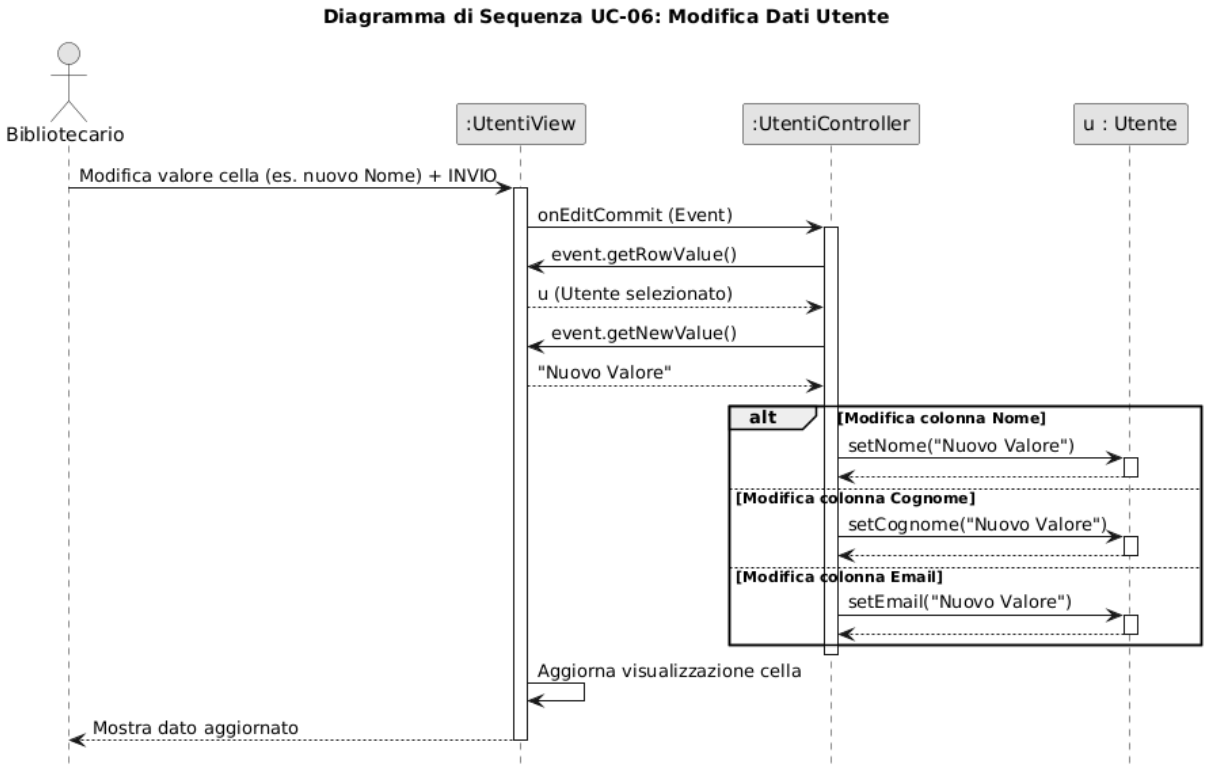

Figura 14: UC06 - Modifica Utente. Diagramma del flusso di editing diretto sulle celle (*edit on commit*) per l'aggiornamento in memoria dei dati anagrafici.

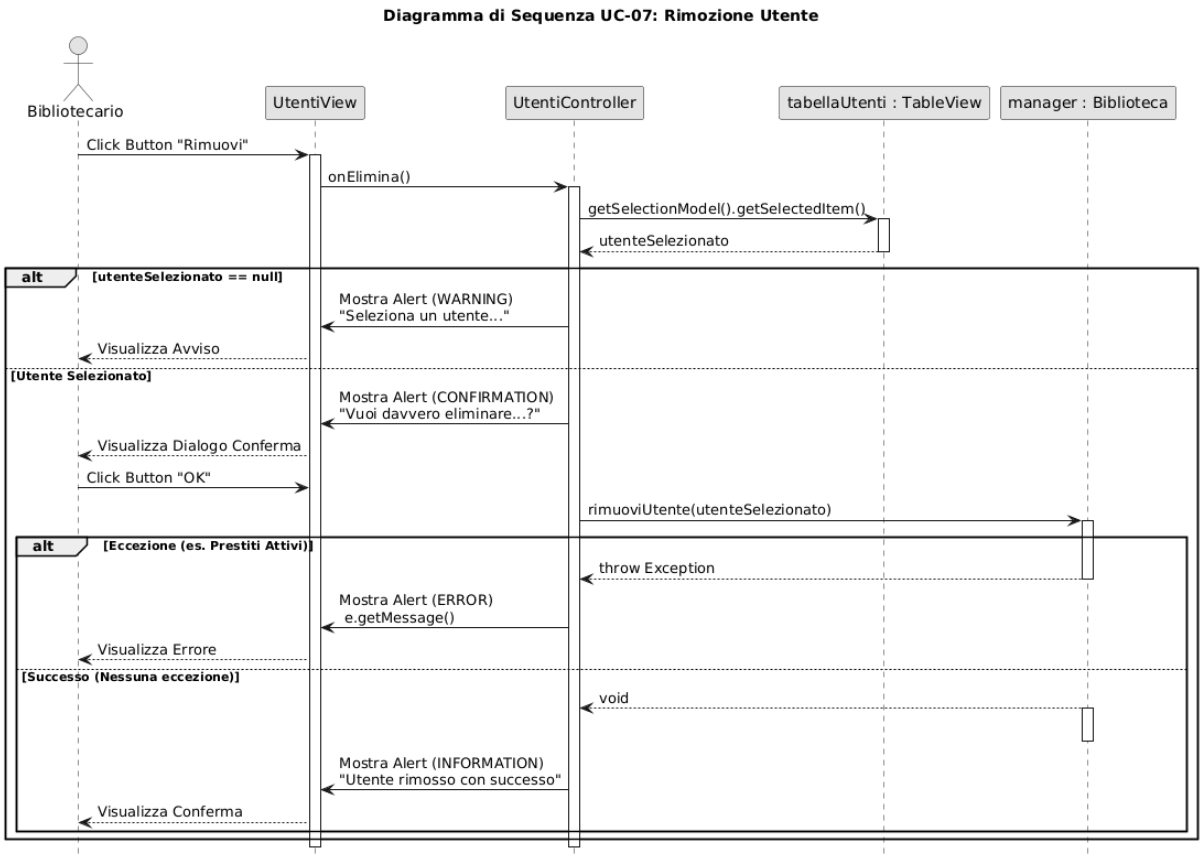

Figura 15: UC07 - Rimozione Utente. Sequenza che include la richiesta di conferma all'operatore e il controllo bloccante in caso di prestiti non ancora restituiti.

3.2.3 Gestione Prestiti

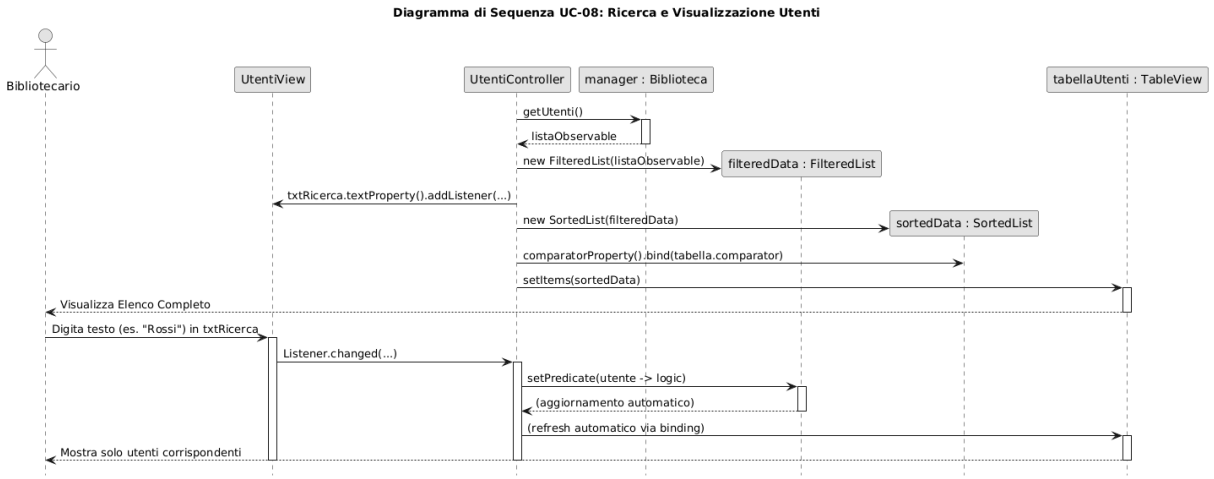

Figura 16: UC08 - Inizializzazione Prestiti. Caricamento delle liste `ObservableList` per popolare le ComboBox di selezione Utenti e Libri all'apertura della vista.

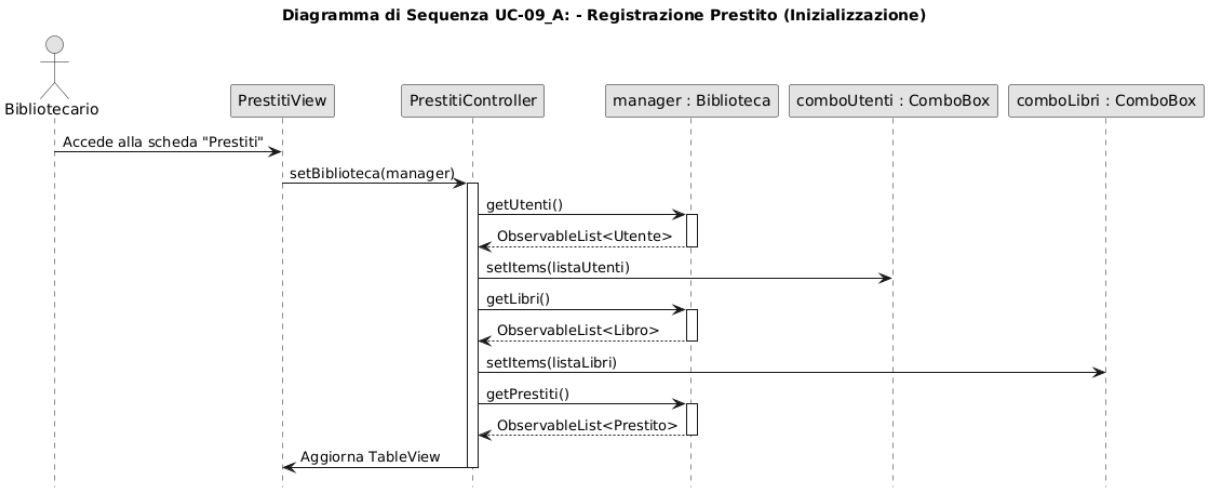

Figura 17: UC09 - Monitoraggio Prestiti. Rappresentazione della logica di inizializzazione di un prestito.

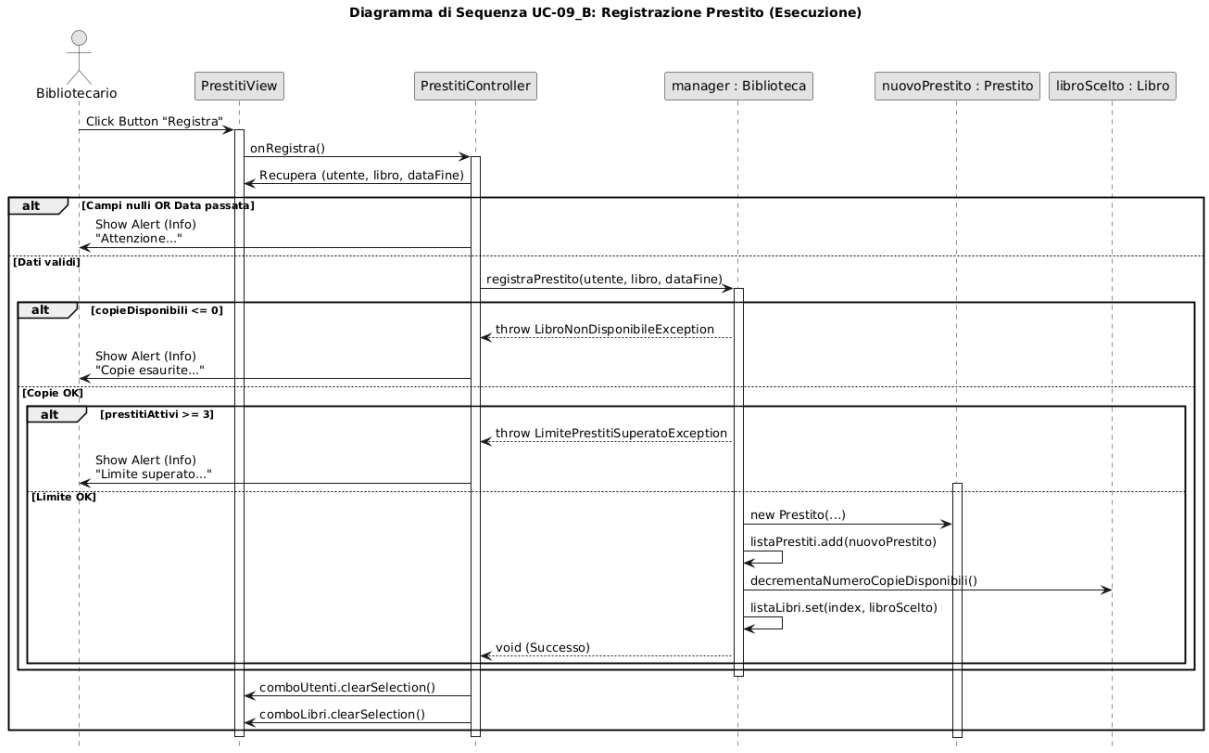

Figura 18: UC09 - Monitoraggio Prestiti. Rappresentazione della logica di rendering condizionale (`RowFactory`) che colora le righe in base allo stato (Scaduto, In Scadenza, Restituito).

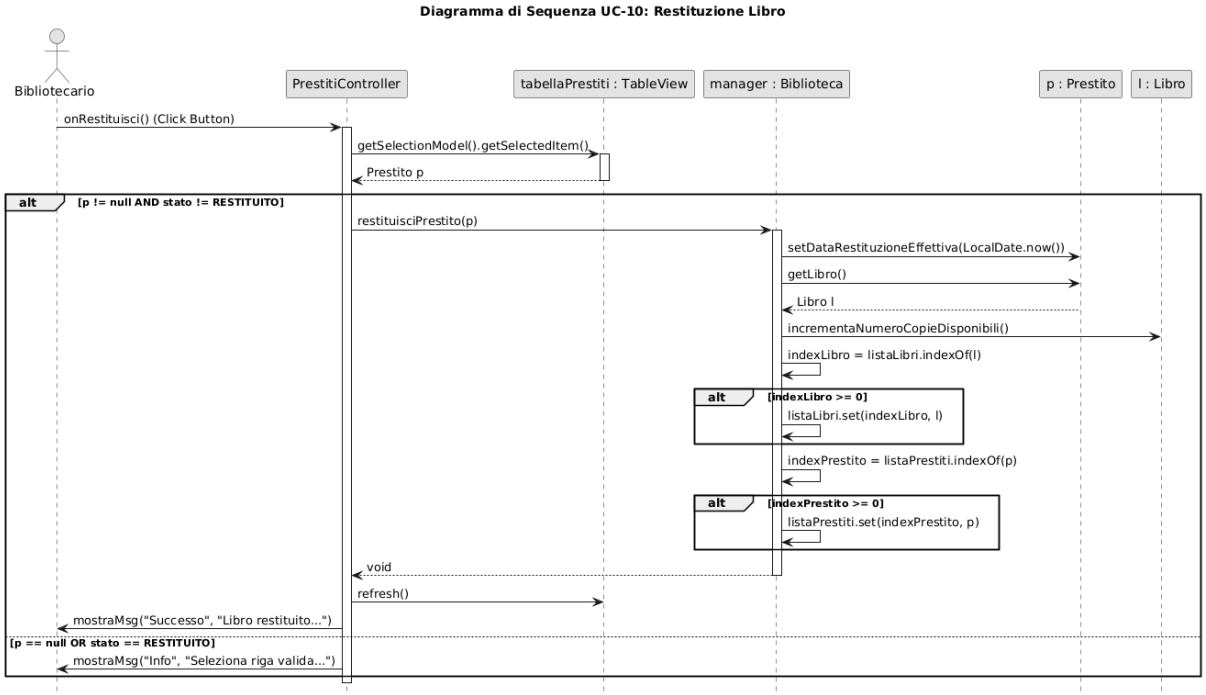

Figura 19: UC10 - Registrazione Prestito. Il flusso include i controlli critici di business: disponibilità delle copie e verifica del limite massimo di prestiti per utente.

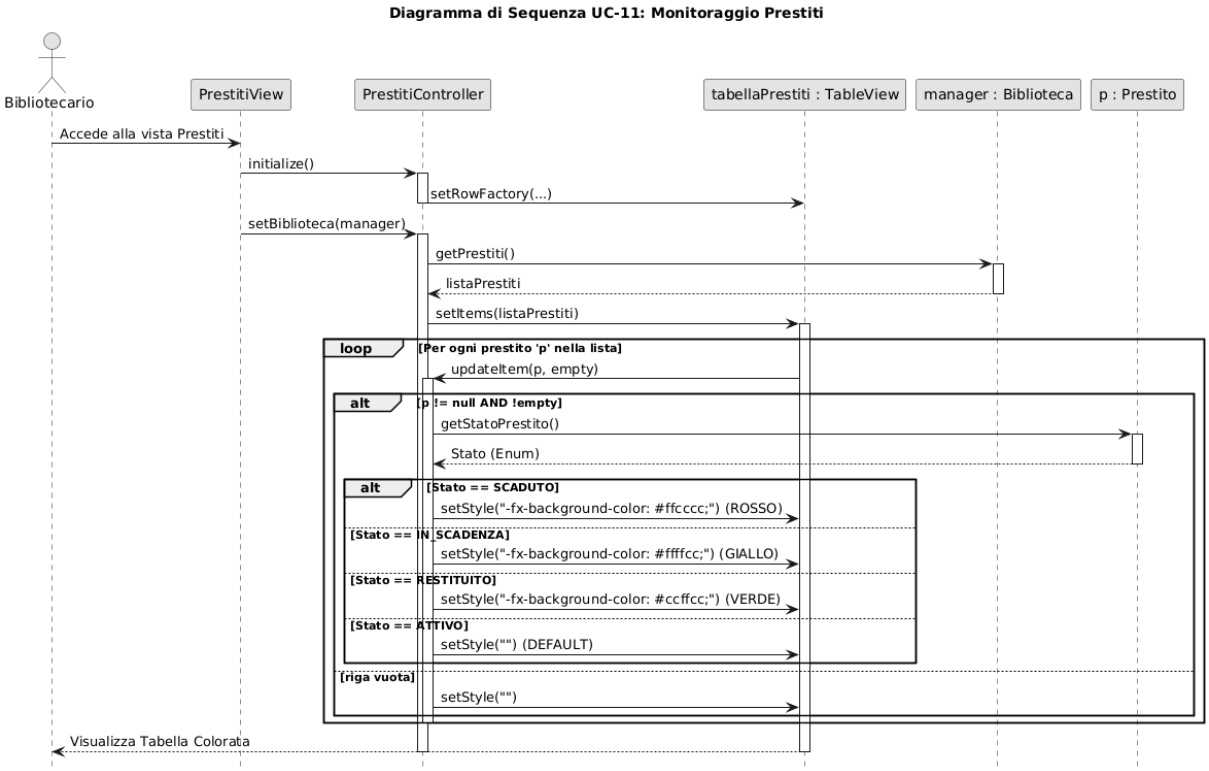

Figura 20: UC11 - Restituzione Prestito. Aggiornamento della data di restituzione effettiva e incremento automatico delle copie disponibili del libro associato.

4 Analisi di Coesione ed Accoppiamento

In questa sezione viene presentata un'analisi qualitativa dell'architettura del sistema *Gestione Biblioteca*, valutando due dei principali attributi di qualità del software: la **Coesione** (il grado di affinità tra gli elementi interni di un modulo) e l'**Accoppiamento** (il grado di interdipendenza tra moduli diversi).

4.1 Analisi della Coesione (Intra-Modulo)

La coesione misura quanto le responsabilità di un singolo modulo (classe o package) siano "concentrate". L'obiettivo progettuale è di massimizzare la coesione verso il livello *Funzionale*.

4.1.1 Package `org.softeng.model`

Questo package contiene le entità del dominio: `Libro`, `Utente`, `Prestito` e `StatoPrestito`.

- **Classi `Libro` e `Utente`**
  - **Livello di Coesione:** *Funzionale* (Alto).
  - **Valutazione:** Queste classi operano come entità non legate tra loro. I metodi presenti (es. `getTitolo()`, `decrementaCopie()`) lavorano esclusivamente sugli attributi dell'istanza per definire lo stato dell'oggetto.
  - **Approfondimento:** È presente una leggera impurità dovuta ai costruttori che accettano stringhe CSV (es. `new Libro(String rigaCSV)`) e al metodo `toCSV()`. Questo introduce una tendenza verso una coesione di tipo *Comunicazionale* limitata alla persistenza, poiché la logica di formattazione dati è "mischiata" alla logica di dominio. Tuttavia, il livello complessivo rimane alto. Infine, per migliorare la coesione di `Libro` ed eliminare la dipendenza dal formato CSV, si potrebbe introdurre una classe separata che adoperi la Serializzazione, quale, ad esempio, `LibroCSVSerializer`. In questo modo `Libro` rimarrebbe puramente Funzionale e il formato di salvataggio potrebbe cambiare senza dover ricompilare le classi del dominio. In sintesi, per migliorare la coesione, dunque, si potrebbe spostare la logica di conversione CSV in una classe di utilità esterna, lasciando le entità del modello pulite.
- **Classe `Prestito`**
  - **Livello di Coesione:** *Funzionale* .
  - **Valutazione:** La classe è altamente coesa poiché incapsula perfettamente la logica relazionale tra un utente e un libro. Il metodo `getStatoPrestito()` utilizza esclusivamente i dati interni (`dataFine`, `dataRestituzione`) per derivare un'informazione complessa, senza dipendere da logiche esterne.

4.1.2 Package `org.softeng.data`

Questo package gestisce la logica di business e la persistenza dei dati.

- **Classe `BibliotecaFileManager`**

  - **Livello di Coesione:** *Funzionale*.
  - **Valutazione:** La classe possiede un'unica responsabilità ben definita: la gestione dell'Input/Output su file. Tutti i metodi (`salvaLibri`, `caricaUtenti`) contribuiscono a questo singolo scopo, rispettando pienamente il *Single Responsibility Principle* (SRP).

- **Classe `Biblioteca`**
  - **Livello di Coesione:** *Comunicazionale*.
  - **Valutazione:** La classe agisce da *colonna portante* del modello. I suoi metodi (`aggiungiLibro`, `registraPrestito`, `rimuoviUtente`) sono raggruppati perché operano sulla stessa infrastruttura dati condivisa (le `ObservableList`). Non raggiunge la coesione puramente funzionale poiché gestisce entità eterogenee (Libri, Utenti e Prestiti) in un'unica classe, ma, per la scala del progetto, è una scelta architetturale accettabile.

4.1.3 Package `org.softeng.controller`

Questo package gestisce l'interazione tra l'interfaccia utente (View) e la logica (Model).

- **Controller Specifici (`LibriController`, `UtentiController`, `PrestitiController`)**
  - **Livello di Coesione:** *Sequenziale / Funzionale*.
  - **Valutazione:** Ogni controller gestisce un flusso di lavoro specifico: intercetta l'evento utente, valida l'input e invoca il modello. I metodi interni (es. `onAggiungi`) eseguono questi passi in una sequenza obbligata, garantendo una buona coesione procedurale e sequenziale.

4.2 Analisi dell'Accoppiamento (Inter-Modulo)

L'accoppiamento misura la dipendenza tra moduli diversi. L'obiettivo è minimizzarlo, favorendo l'accoppiamento *per Dati* (passaggio di parametri) rispetto all'accoppiamento *per Contenuto* o *Controllo*.

4.2.1 Relazione Controller → Model (`Biblioteca`)

- **Tipo:** *Accoppiamento per Timbro (Stamp Coupling)* e *per Controllo*.
- **Valutazione:** I controller dipendono dalla classe `Biblioteca`. È un accoppiamento *Stamp* perché al controller viene passato l'intero oggetto complesso `Biblioteca` tramite il metodo `setBiblioteca()`, anche se il controller ne usa solo una piccola parte (es. `LibriController` usa solo la lista dei libri).

4.2.2 Relazione `Biblioteca` → `BibliotecaFileManager`

- **Tipo:** *Accoppiamento per Dati (Data Coupling)*.
- **Valutazione: Ottimo**. La classe `Biblioteca` invoca il file manager passando liste di dati come parametri espliciti (`List<Libro>`, ecc.). Non c'è condivisione di variabili globali né dipendenze nascoste. Questo permette di modificare l'implementazione del salvataggio senza impattare sulla logica della biblioteca.

4.2.3 Relazione Entità (`Prestito` → `Utente`, `Libro`)

- **Tipo:** *Accoppiamento per Dati (Data Coupling).*
- **Valutazione:** La classe `Prestito` mantiene riferimenti alle istanze di `Utente` e `Libro`. Questo è un accoppiamento necessario per navigare le associazioni del dominio. Poiché l'accesso avviene tramite metodi pubblici e non manipolando direttamente la memoria interna, l'accoppiamento è mantenuto a un livello basso e sicuro.

4.2.4 Relazione Model → Persistenza (CSV)

- **Tipo:** *Accoppiamento per Dati (Data Coupling).*
- **Valutazione: Critico**. Le classi `Libro` e `Utente` contengono costruttori che accettano una stringa CSV grezza e metodi `toCSV()`. Questo crea una dipendenza implicita tra la definizione dell'entità e il formato fisico di salvataggio. Se il formato del file cambiasse (es. ordine dei campi), sarebbe necessario modificare il codice delle entità, violando parzialmente la separazione dei livelli.

4.3 Tabella Riepilogativa

Modulo / Relazione	**Livello Rilevato**	**Commento**
COESIONE (Obiettivo: Alta)		
Package `Model`	Funzionale (con aspetti Comunicazionali)	Le classi sono focalizzate sull'entità, ma gestiscono anche la propria formattazione CSV (impurità).
Classe `Biblioteca`	Comunicazionale	I metodi (registraPrestito, rimuoviUtente) operano sull'insieme condiviso delle liste dati.
Classe `FileManager`	Funzionale	Singola responsabilità (I/O).
Package `Controller`	Sequenziale	Esegue in ordine: lettura UI -> validazione -> chiamata al Modello.
ACCOPPIAMENTO (Obiettivo: Basso)		
Controller $\rightarrow$ Model	Stamp / Control	Standard per architetture MVC.
Data $\rightarrow$ FileManager	Data Coupling	Scambio dati tramite parametri.
Prestito $\rightarrow$ Entità	Data Coupling	Associazioni necessarie al dominio.
Model $\rightarrow$ CSV	Data Coupling	Dipendenza dal formato file nelle entità.

Tabella 1: Sintesi dell'analisi di Coesione ed Accoppiamento del sistema.