Sistema di Gestione Biblioteca

Documentazione Tecnica

Autori:
Berardino Bruno
Bacco Ferdinando
Annarumma Andrea
Gravina Angela

1. Panoramica del Sistema

Il **diagramma delle classi** rappresenta l'architettura software di un'applicazione desktop sviluppata in **JavaFX** per la gestione di una biblioteca. Il sistema permette la gestione **CRUD** (Create, Read, Update, Delete) di **Libri**, **Utenti** e **Prestiti**, garantendo la persistenza dei dati e una separazione tra logica di business e interfaccia utente.

2. Architettura Software

Il sistema segue un'architettura derivata dal pattern **MVC (Model-View-Controller)**, mediata da una classe coordinatrice centrale.

- **Model (Modello):** Rappresentato dalle classi entità (`Libro, Utente, Prestito`) e dalle classi di gestione dati (`Catalogo, Lista, ListaPrestiti`).
- **View (Vista):** L'interfaccia grafica è definita tramite file FXML gestiti dai rispettivi controller.
- **Controller:** I vari `*Controller` gestiscono l'interazione utente, mentre `MainApp` agisce come orchestratore principale.

3. Analisi dei Componenti

3.1 Inizializzazione e Orchestrazione

- **Main:** È il punto di ingresso dell'applicazione (`main(args)`), responsabile dell'avvio del ciclo di vita del software.
- **MainApp:** È la classe centrale del sistema.
  - **Responsabilità:** Inizializza lo `Stage` primario, istanzia i gestori dei dati (`Catalogo, ListaUtenti, ListaPrestiti`) e coordina la navigazione tra le diverse viste.
  - **Relazioni:** Ha una relazione di **composizione** (rombo pieno) con i gestori dei dati, indicando che `MainApp` possiede e controlla il ciclo di vita di questi oggetti.

3.2 Controller d'Interfaccia (UI Logic)

Le classi come: `HomeController`, `LibriController, UtentiController` e `PrestitiController` gestiscono la logica di presentazione.
La scelta di utilizzare metodi espliciti come `popolaForm(T)` e `showEditDialog()` riflette la priorità data all'esperienza utente (UX). Il controller funge da mediatore visivo, recuperando e visualizzando lo stato del Model prima di richiedere l'input, una scelta tipica di un'interfaccia user-friendly basata su form e dialog box.

- **Dependency Injection:** Tutti i controller presentano un metodo `setMainApp(MainApp)`,indicando che ricevono un riferimento all'applicazione principale per poter navigare tra le schermate e accedere ai dati condivisi.

- **Funzionalità:** Espongono metodi per la gestione degli eventi UI (es. **`clickNuovo`**, **`rimuovi`**, **`modifica`**) e per il passaggio di dati (**`setData`**, **`sendAttributi`**).

3.3 Gestione dei Dati (Business Logic Layer)

Il livello di gestione dati è strutturato gerarchicamente tramite l'uso di **Interfacce** e **Generics** per favorire il riuso del codice e il polimorfismo.

- **Interfaccia `Gestione<T>`:** Un'interfaccia astratta/generica che definisce le operazioni standard:
  1. **`getElenco()`:** Ritorna una **`ObservableList`**.
  2. **`aggiungi(T)`**, **`rimuovi(T)`**, **`modifica(T)`:** Operazioni di mutazione.
- **Interfacce Specifiche:** **`GestioneLibri`**, **`GestioneUtenti`**, **`GestionePrestiti`** estendono **`Gestione<T>`** aggiungendo metodi di ricerca specifici (es. **`ricercaTitolo`**).
- **Implementazioni Concrete:**
  1. **`Catalogo`**: Implementa **`GestioneLibri`**.
  2. **`Lista`** (o **`ListaUtenti`**): Implementa **`GestioneUtenti`**.
  3. **`ListaPrestiti`**: Implementa **`GestionePrestiti`**.
  4. Tutte implementano l'interfaccia **`Serializable`**, permettendo il salvataggio dello stato dell'oggetto.

3.4 Modello di Dominio (Entities)

Le entità rappresentano i dati fondamentali: **Libro**, **Utente**, **Prestito**.

- Implementano **`Comparable`** (per l'ordinamento) e **`Serializable`** (per la persistenza).
- **Relazioni:** Un **`Prestito`** associa un **`Libro`** e un **`Utente`**. Dal diagramma si evince una relazione tra **`Prestito`** e **`Libro`**(molteplicità 1 a 0.3) e tra **`Prestito`** e **`Utente`**.

3.5 Persistenza

- **Interfaccia `SalvataggioDati`:** Definisce il contratto per il salvataggio e caricamento (**`salva`**, **`carica`**). Questo applica il principio di *Inversion of Control*, disaccoppiando la logica di business dal supporto di memorizzazione fisico.
- **`GestoreFile`:** Implementa **`SalvataggioDati`**. Si occupa della serializzazione delle liste di oggetti su file binari o di testo.

3.6 Gestione delle Eccezioni

Sono definite eccezioni personalizzate per gestire errori di dominio specifici, lanciate dai metodi delle classi di gestione:

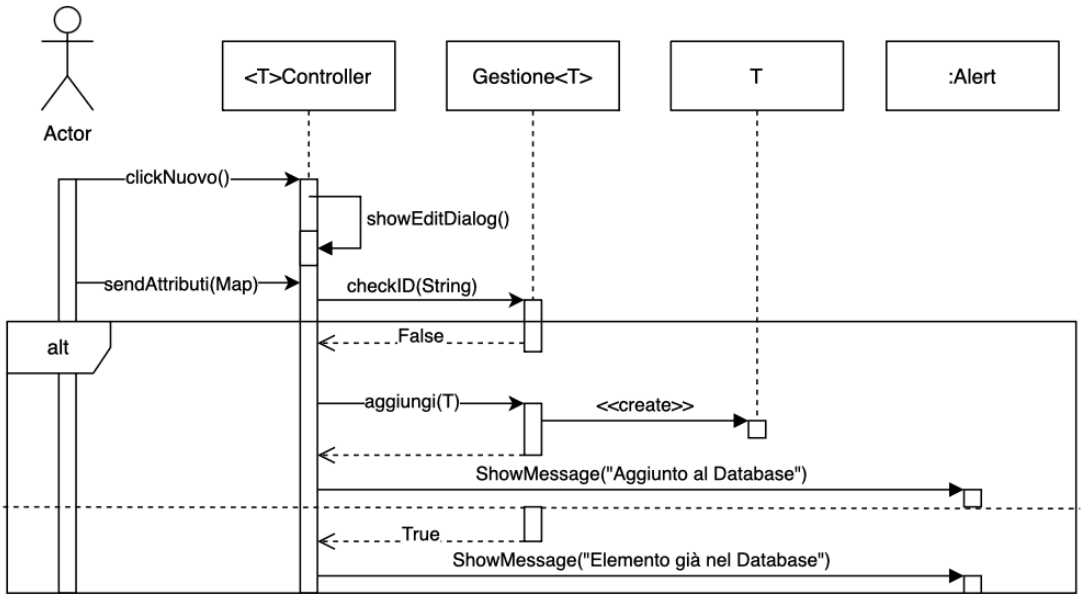
- `PrestitoNonValidoException`
- `EliminazioneNonValidaException`

4. Design Pattern Rilevati

1. **Facade / Mediator:** `MainApp` agisce come una facciata semplificata per i controller e media la comunicazione tra le parti del sistema.
2. **Observer Pattern:** Utilizzato implicitamente tramite `ObservableList` di JavaFX. Quando i dati nel Catalogo cambiano, la UI si aggiorna automaticamente.
3. **Strategy Pattern (parziale):** L'uso dell'interfaccia `SalvataggioDati` permette di cambiare la strategia di persistenza senza modificare l'applicazione.
4. **Template Method / Generic Programming:** L'uso di `Gestione<T>` definisce uno scheletro comune per le operazioni CRUD.

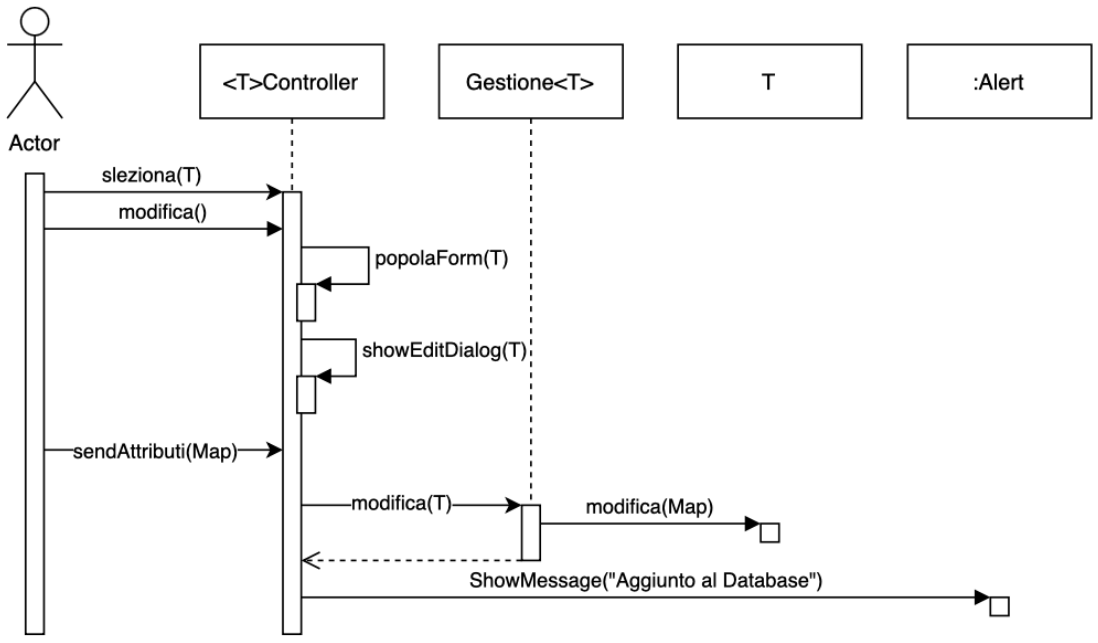
5. Vista Dinamica e Flussi Operativi *(Sequence Diagrams)*

5.1 Inserimento Nuova Entità (Generico)



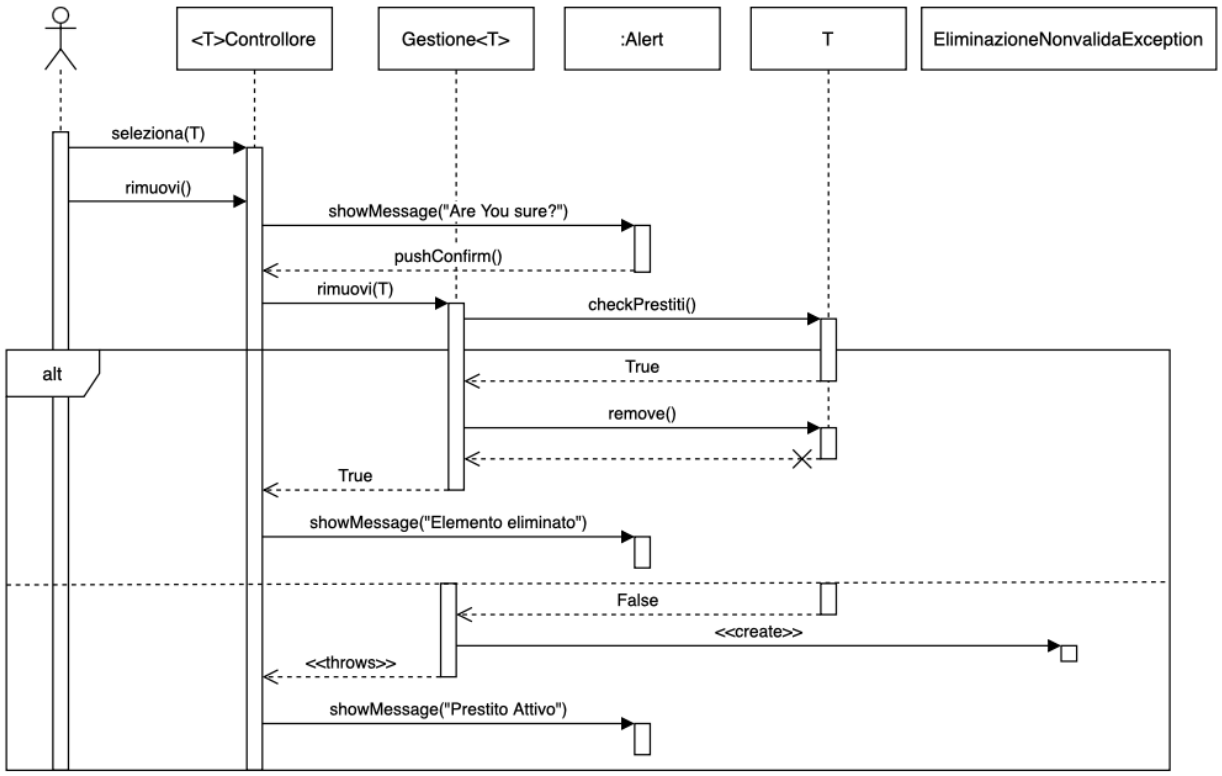
- **Diagramma:** *Add Generic*
- **Flusso:** L'attore inizia l'azione `clickNuovo()`. Il sistema mostra un dialog di inserimento.
- **Validazione:** Alla conferma (`sendAttributi`), il gestore controlla l'unicità tramite `checkID(String)` (es. ISBN o Matricola).
- **Esito:**
  - *False (ID non presente):* L'oggetto viene creato **(`<<create>>`),** aggiunto e salvato.
  - *True (ID duplicato):* Operazione abortita, messaggio di errore ("Elemento già nel Database").

5.2 Modifica Entità (Generico)



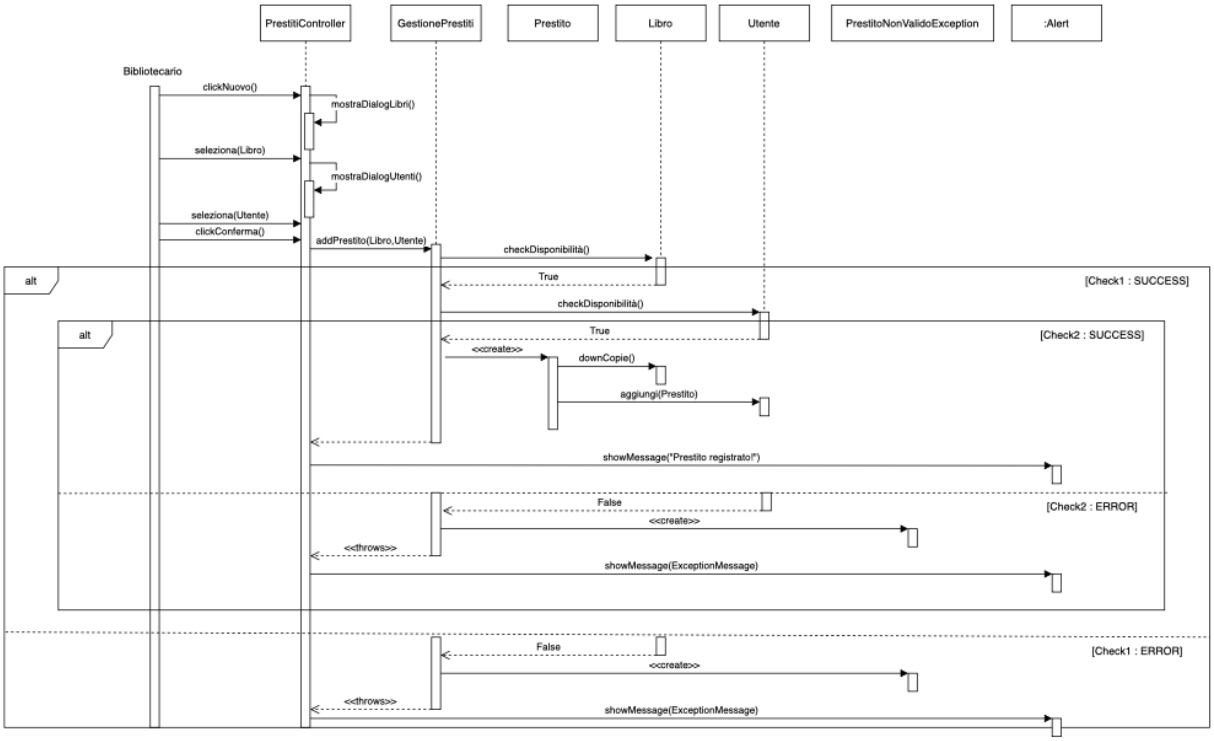
- **Diagramma:** *Modify Generic*
- **Flusso:** Selezione elemento e click su `modifica()`.
- **Binding:** Il controller recupera i dati (`popolaForm(T)`) per il dialog.
- **Persistenza:** Le modifiche sono inviate tramite una `Map` (DTO implicito). Il gestore aggiorna il modello e notifica il successo

5.3 Rimozione Entità e Integrità Referenziale



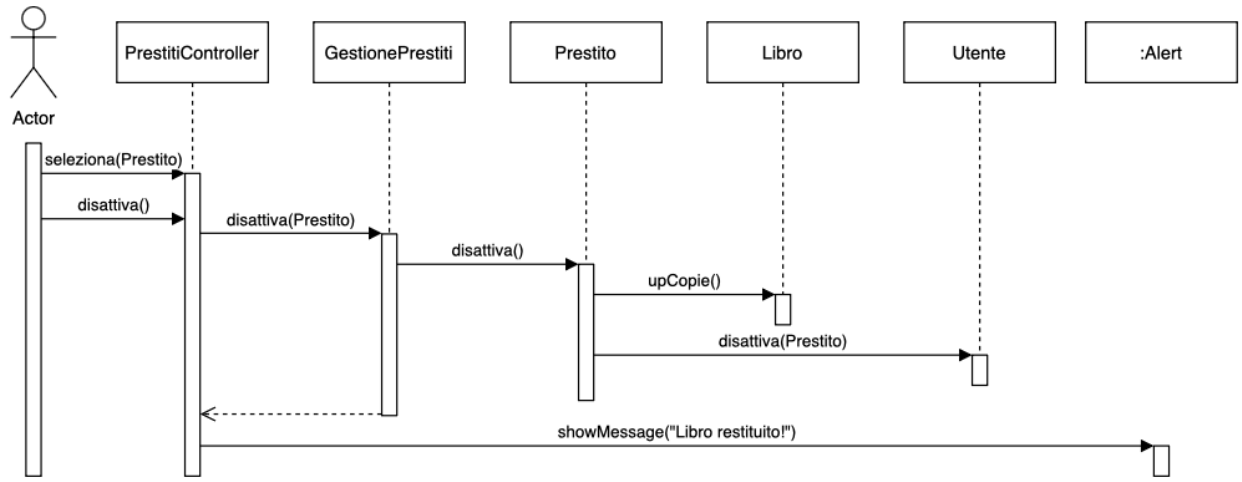
- **Diagramma:** *Delete Generic*
- **Flusso:** Richiesta `rimuovi()` e conferma esplicita ("Are you sure?").
- **Guardia:** `checkPrestiti()` verifica se l'entità ha dipendenze attive.
    - *True (Vincoli ok):* Oggetto rimosso fisicamente.
    - *False (Violazione):* Viene lanciata `EliminazioneNonValidaException`. Il controller mostra errore ("Prestito Attivo"), prevenendo inconsistenze

5.4 Registrazione Prestito (Workflow Transazionale)



- **Diagramma:** *Add Loan*
- **Interazione:** Selezione sequenziale di Libro e Utente.
- **Doppia Validazione**: **`checkPrestiti()`** verifica se l'entità ha dipendenze attive.
  - **`checkDisponibilità()`** *su Libro (copie>0)*
  - **`checkDisponibilità()`** *su Utente (limite prestiti/sanzioni)*
- **Aggiornamento:** Se entrambi i check hanno successo: creazione `Prestito`, decremento copie libro (**`downCopie()`**), registrazione prestito.
- **Errore:** Se un check fallisce, viene sollevata **`PrestitoNonValidoException`**.
- **UX**:La selezione del Libro e dell'Utente avviene tramite finestre di dialogo modali (**`mostraDialogLibri`**, **`mostraDialogUtenti`**), una scelta architetturale per guidare l'utente e minimizzare gli errori di input (es. digitazione scorretta di ISBN o Matricola)

5.5 Restituzione Prestito



- **Diagramma:** *Return Loan*
- **Flusso:** Selezione prestito e comando `disattiva()`.
- **Side-effects:**
  i. `upCopie()` sul Libro (incremento disponibilità).
  ii. Aggiornamento stato prestito.
  iii. Notifica "Libro restituito".

6. Giustificazione delle Scelte Progettuali e Analisi Qualitativa

La progettazione mira a massimizzare la qualità interna del software, con focus su **manutenibilità** e **modularità**.

6.1 Decomposizione e Modularità

Il sistema usa una decomposizione **Object-Oriented** mappando le entità del dominio in classi. L'architettura a layer rispetta la **Separazione delle Preoccupazioni (SoC)**:

1. **View/Controller:** Gestione interazione.
2. **Domain Logic:** Regole di business (`Gestione<T>`).
3. **Data:** Stato (`Libro`, `Utente`).
4. **Persistenza:** Storage fisico (`GestoreFile`)

6.2 Analisi della Coesione (Cohesion)

Il progetto punta alla Coesione Funzionale (il livello più alto).

1. **Gestione:** Contiene solo metodi per la gestione della collezione (CRUD), senza logica UI o I/O di basso livello.
2. **Entità:** Metodi strettamente legati agli attributi.
3. **GestoreFile:** Coesione focalizzata solo sulla serializzazione/deserializzazione.

6.3 Analisi dell'Accoppiamento (Coupling)

Obiettivo: Basso Accoppiamento.

1. **Accoppiamento per Dati:** I controller passano solo i dati necessari (`Map` o oggetto `T`) ai gestori.
2. **Disaccoppiamento tramite Interfacce:** L'uso di `SalvataggioDati` applica l'Inversione della Dipendenza. Le classi di gestione dipendono dall'astrazione, non dal `GestoreFile` concreto (principio Open-Closed).
3. **Associazione vs Ereditarietà:** Privilegiata l'associazione (es. `Prestito` -> `Libro`) per ridurre l'accoppiamento forte tipico dell'ereditarietà.

6.4 Robustezza e "Design by Contract"

La progettazione incorpora elementi di Design by Contract. Le eccezioni personalizzate agiscono come guardie UI: il loro lancio interrompe la transazione e permette al Controller di mostrare un Alert semplice e specifico (es. 'Impossibile rimuovere utente con prestiti ancora attivi'), migliorando l'usabilità rispetto a messaggi di errore generici.

1. **Precondizioni:** Verifica di disponibilità (`checkDisponibilità`) prima di creare prestiti.
2. **Eccezioni:** `PrestitoNonValidoException` impedisce violazioni delle invarianti (es. prestare un libro con 0 copie).
3. **Integrità:** `checkPrestiti()` agisce da guardia contro stati inconsistenti.

6.5 Principio DRY

L'uso di **`Gestione<T>`** applica il principio DRY (Don't Repeat Yourself), centralizzando la logica CRUD ed evitando duplicazioni tra gestori di entità diverse.