Documento di Design

Sistema di Gestione Biblioteca Universitaria

Gruppo 18
Corso di Ingegneria del Software

Dicembre 2025

Indice

1 Architettura del Sistema

1.1 Introduzione

Il Sistema di Gestione della Biblioteca Universitaria è stato sviluppato adottando una struttura che suddivide le funzioni del sistema in più livelli, così da migliorarne la chiarezza e la manutenibilita'. L'organizzazione segue il modello Model–View–Controller (MVC), che separa i dati, la presentazione e la gestione del funzionamento dell'applicazione.

1.2 Diagramma dei Package

Il sistema è organizzato in quattro package principali che riflettono la struttura MVC:

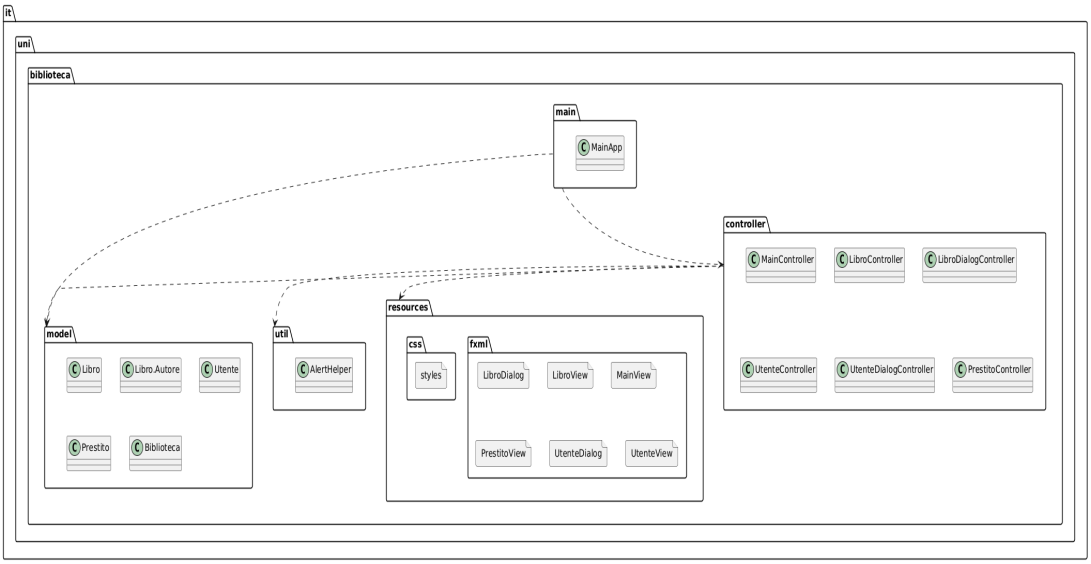

Figura 1: Diagramma dei package

1.2.1 Descrizione dei Package

- **main**: Contiene la classe `MainApp` che funziona come punto di ingresso dell'applicazoine JavaFX e gestisce il ciclo di vita del sistema.
- **model**: Contiene tutte le classi del dominio applicativo (`Libro`, `Utente`, `Prestito`, `Biblioteca`). Queste classi rappresentano le entita' di business e la logica di gestione dati.
- **controller**: Contiene i controller JavaFX che gestiscono le interazioni utente e coordinano Model e View (`MainController`, `LibroController`, `UtenteController`, `PrestitoController`, e i relativi DialogController).
- **util**: Contiene classi di utilita' come `AlertHelper` per la gestione centralizzata dei messggi all'utente.
- **resources**: Contiene i file FXML che definiscono le interfacce grafiche e i file CSS per lo styling.

1.3 Scelte Architetturali

1.3.1 Pattern Model-View-Controller (MVC)

Il sistema adotta il pattern MVC per garantire una chiara separazione delle responsabilita':

Model (Modello) Il package `model` contiene le classi che rappresentano il dominio applicativo:

- `Libro`: Rappresenta un libro con i suoi attributi (ISBN, titolo,autori, anno, copie)
- `Utente`: Rapprsenta un utente della biblioteca con i suoi prestiti attivi
- `Prestito`: Rappresenta un prestito con le relative date e stato
- `Biblioteca`: Classe centrale che coordina la gestione dilibri,utenti e prestiti

Il Model è completamente indipendente dalla View e contiene tutta la logica applicativa,incluse le validzioni e le regole di integrita' dei dati.

View (Vista) Le View sono definite tramite file FXML:

- `MainView.fxml`: Layout principale dell'applicazione
- `LibroView.fxml`: Interfaccia per la gestione dei libri
- `UtenteView.fxml`: Interfaccia per la gestione degli utenti
- `PrestitoView.fxml`: Interfccia per la gestione dei prestiti
- `LibroDialog.fxml`, `UtenteDialog.fxml`: Dialog per inserimento/modifica

Le View sono completamente dichiarative e non contengono logica di business.

Controller (Controllore) I Controller mediano tra Model e View:

- `MainController`: Gestisce la navigazione principale
- `LibroController`, `UtenteController`, `PrestitoController`: Gestiscono le operazioni specifiche di ciascuna entità
- `LibroDialogController`, `UtenteDialogController`: Gestiscono i dialog di inserimento/modifica

I Controller ricevono gli eventi dall'interfaccia, invocano i metodi appropriati del Model e aggiornano la View con i risultati.

1.3.2 Vantaggi dell'Architettura MVC Adottata

1. **Separazione delle Responsabilità**: Ogni layer ha una responsabilita' ben definita, facilitando la comprensione e la manutenzione del codice.
2. **Riusabilità**: Il Model può essere utilizzato da diverse View o Controller senza modifiche.
3. **Testabilità**: La logica di business nel Model puo' essere testata indipendentemente dall'interfaccia grafica.
4. **Manutenibilità**: Modifiche all'interfaccia non impattano la logica di business e viceversa.

1.4 Pattern di Progettazione Utilizzati

1.4.1 Singleton Pattern

Applicazione nel Sistema La classe `Biblioteca` implementa il pattern Singleton per garantire che esista una sola istanza della biblioteca nell'intero sistema.

Motivazione

- **Unicità dei Dati**: Deve esistere un unico punto diaccesso centralizzato a libri, utenti e prestiti per garantire consistenza.
- **Controllo dell'Accesso**: Tutti i controller devono operare sulla stessa istanza per evitare inconsistenze.
- **Gestione Persistenza**: Il salvataggio e caricamento dati devono riferirsi a un'unica istanza.

1.4.2 Facade Pattern (Implicito)

Applicazione nel Sistema La classe `Biblioteca` funge anche da Facade, fornendo un'interfaccia semplificata per operzioni complesse che coinvolgono più componenti del sistema. Invece di far gestire ai controller la logica complessa delle interazioni tra `Utente`, `Libro` e `Prestito`, la `Biblioteca` si occupa di coordinare questi processi internamente.

Esempio Il metodo `registraPrestito()` coordina operazioni su Utente, Libro e Prestito:

- Verifica disponibilità libro
- Verifica limite prestiti utente
- Crea nuovo prestito
- Aggiorna copie disponibili
- Aggiunge prestito all'utente

Questa complessità è nascosta ai controller, che invocano un singolo metodo.

1.4.3 Observer Pattern (JavaFX)

Applicazione nel Sistema JavaFX utilizza internamente l'Observer pattern atraverso le `ObservableList`:

```
1 private ObservableList<Libro> listaLibri;
2 // La TableView osserva automaticamente le modifiche
3 tabellaLibri.setItems(listaLibri);
```

Quando la lista cambia, la tabella si aggiorna automaticamente senza intervento esplicito.

1.5 Dipendenze tra Moduli

1.5.1 Dipendenze del Package Main

- **main** → **model**: `MainApp` dipende da `Biblioteca` per caricamento/salvataggio dati
- **main** → **controller**: Carica il `MainController` come controller principale

1.5.2 Dipendenze del Package Controller

- **controller** → **model**: Tutti i controller dipendono dalle classi del model (`Biblioteca`, `Libro`, `Utente`, `Prestito`)
- **controller** → **util**: I controller utilizzano `AlertHelper` per i messaggi
- **controller** → **resources**: I controller sono collegati ai file FXML tramite annotazioni `@FXML`

1.5.3 Dipendenze del Package Model

- **Biblioteca** → **Libro, Utente, Prestito**: Gestisce collezioni di queste entità
- **Prestito** → **Libro, Utente**: Mantiene riferimenti a libro e utente prestati
- **Utente** → **Prestito**: Mantiene lista dei prestiti attivi

Nota Importante Le dipendenze sono unidirezionali e rispettano il principio di inversione delle dipendenze: i moduli di alto livello (Controller) dipendono da astrazioni (interfacce pubbliche del Model) e non da dettagli implementativi.

1.6 Scelte di Persistenza

Il sistema utilizza la serializzazione Java per la persistenza dei dati:

- **Vantaggi**: Semplicità di implementazione, serializzazione automatica dell'intero grafo di oggetti.
- **File**: `biblioteca_data.ser` generato automaticamente nella directory di esecuzione.
- **Gestione**: Caricamento all'avvio (`MainApp.start()`) e salvataggio alla chiusura (`MainApp.stop()`) e dopo ogni modifica.

2 Modello Statico

2.1 Diagramma delle Classi

Il diagramma delle classi completo del sistema è riportato nella Figura 2. Il diagramma mostra tutte le classi del sistema organizzate per package, con i relativi attributi, metodi e relazioni.

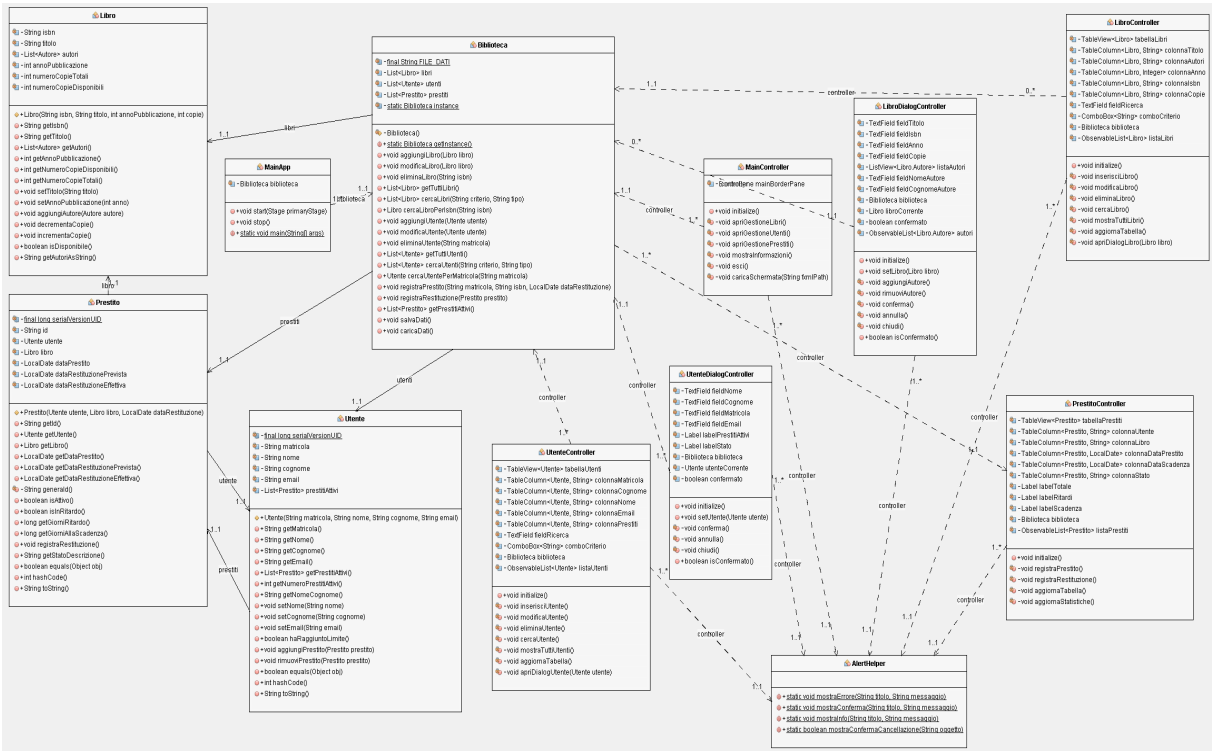

Figura 2: Diagramma delle classi

Il diagramma mostra:

- Package `model`: Libro, Libro.Autore, Utente, Prestito, Biblioteca
- Package `controller`: MainController, LibroController, UtenteController, PrestitoController, LibroDialogController, UtenteDialogController
- Package `main`: MainApp
- Package `util`: AlertHelper
- Relazioni di dipendenza, associazione e composizione tra le classi

Figura 3: Diagramma delle classi completo del sistema

2.2 Descrizione delle Classi

2.2.1 Package Model

Classe Libro

Responsabilità: Rappresenta un libro del catalogo della biblioteca con tutti i suoi attributi e gestisce la disponibilità delle copie.

Attributi principali:

- `isbn: String` - Codice identificativo univoco (max 20 caratteri)
- `titolo: String` - Titolo del libro (max 200 caratteri)
- `autori: List<Autore>` - Lista degli autori (minimo 1 obbligatorio)
- `annoPubblicazione: int` - Anno di pubblicazione (range 1000 - anno corrente)
- `numeroCopieTotali: int` - Numero totale di copie possedute
- `numeroCopieDisponibili: int` - Copie attualmente disponibili per il prestito

Metodi pubblici principali:

- `aggiungiAutore(Autore)`: Aggiunge un autore alla lista
- `isDisponibile(): boolean`: Verifica se almeno una copia è disponibile
- `incrementaCopie()`: Incrementa le copie disponibili (restituzione)
- `decrementaCopie()`: Decrementa le copie disponibili (prestito)
- `getAutoriAsString(): String`: Formatta la lista autori per visualizzazione

Relazioni:

- Composizione con `Libro.Autore` (1 a molti)
- Associazione con `Biblioteca` (molti a 1)
- Associazione con `Prestito` (1 a molti)

Classe Libro.Autore (Nested Class)

Responsabilità: Rappresenta un autore di un libro con nome e cognome.

Motivazione della classe annidata: L'autore è sempre associato a un libro e non ha senso di esistere indipendentemente nel contesto di questo sistema. La classe annidata esprime questa dipendenza concettuale.

Attributi:

- `nome: String` - Nome dell'autore (max 50 caratteri)
- `cognome: String` - Cognome dell'autore (max 50 caratteri)

Metodi:

- `toString(): String` - Restituisce "Cognome Nome"

Classe Utente

Responsabilità: Rappresenta un utente (studente universitario) della biblioteca e gestisce i suoi prestiti attivi.

Attributi principali:

- `matricola: String` - Identificativo univoco (max 10 caratteri, non modificabile)
- `nome: String` - Nome dell'utente (max 50 caratteri)
- `cognome: String` - Cognome dell'utente (max 50 caratteri)
- `email: String` - Email istituzionale (formato valido richiesto)
- `prestitiAttivi: List<Prestito>` - Lista dei prestiti attualmente attivi

Metodi pubblici principali:

- `haRaggiuntoLimite(): boolean` - Verifica se l'utente ha 3 prestiti attivi
- `aggiungiPrestito(Prestito)`: Aggiunge un prestito alla lista
- `rimuoviPrestito(Prestito)`: Rimuove un prestito dalla lista
- `getNumeroPrestitiAttivi(): int` - Conta i prestiti attivi
- `getNomeCognome(): String` - Restituisce "Cognome Nome"
- `getPrestitiAttivi(): List<Prestito>` - Restituisce copia difensiva della lista

Relazioni:

- Associazione con `Biblioteca` (molti a 1)
- Associazione con `Prestito` (1 a molti)

Classe Prestito

Responsabilità: Rappresenta un prestito di un libro a un utente e gestisce il ciclo di vita del prestito (attivo, in ritardo, chiuso).

Attributi principali:

- `id: String` - Identificativo univoco generato automaticamente
- `utente: Utente` - Riferimento all'utente che ha effettuato il prestito
- `libro: Libro` - Riferimento al libro prestato
- `dataPrestito: LocalDate` - Data di inizio prestito
- `dataRestituzioneRevista: LocalDate` - Data prevista di restituzione
- `dataRestituzioneEffettiva: LocalDate` - Data effettiva (null se attivo)

Metodi pubblici principali:

- `isAttivo(): boolean` - Verifica se il prestito è ancora aperto

- `isInRitardo(): boolean` - Verifica se è scaduto e non restituito
- `getGiorniRitardo(): long` - Calcola i giorni di ritardo
- `getGiorniAllaScadenza(): long` - Calcola i giorni mancanti
- `registraRestituzione()` - Imposta la data di restituzione effettiva
- `getStatoDescrizione(): String` - Restituisce descrizione testuale dello stato

Relazioni:

- Associazione con `Utente` (molti a 1)
- Associazione con `Libro` (molti a 1)
- Associazione con `Biblioteca` (molti a 1)

Logica di business: La classe implementa tutta la logica temporale dei prestiti, calcolando automaticamente ritardi e scadenze. Questo incapsula la complessità e facilita la manutenzione futura.

Classe Biblioteca

Responsabilità: Classe centrale che coordina tutte le operazioni del sistema, gestisce le collezioni di libri, utenti e prestiti, e fornisce i servizi di alto livello ai controller.
Pattern implementati:

- **Singleton**: Garantisce una sola istanza
- **Facade**: Nasconde la complessità delle operazioni multi-entità

Attributi principali:

- `instance: static Biblioteca` - Istanza singleton
- `libri: List<Libro>` - Collezione di tutti i libri
- `utenti: List<Utente>` - Collezione di tutti gli utenti
- `prestiti: List<Prestito>` - Collezione di tutti i prestiti

Metodi pubblici principali - Gestione Libri:

- `aggiungiLibro(Libro)`: Inserisce un nuovo libro con validazioni
- `modificaLibro(Libro)`: Modifica un libro esistente
- `eliminaLibro(String isbn)`: Elimina un libro (verifica prestiti attivi)
- `getTuttiLibri(): List<Libro>` - Restituisce lista ordinata per titolo
- `cercaLibri(String, String): List<Libro>` - Ricerca per titolo/autore/ISBN
- `cercaLibroPerIsbn(String): Libro` - Ricerca esatta per ISBN

Metodi pubblici principali - Gestione Utenti:

- `aggiungiUtente(Utente)`: Inserisce un nuovo utente con validazioni
- `modificaUtente(Utente)`: Modifica un utente esistente
- `eliminaUtente(String matricola)`: Elimina un utente (verifica prestiti attivi)
- `getTuttiUtenti(): List<Utente>` - Restituisce lista ordinata per cognome/nome
- `cercaUtenti(String, String): List<Utente>` - Ricerca per cognome/matricola
- `cercaUtentePerMatricola(String): Utente` - Ricerca esatta

Metodi pubblici principali - Gestione Prestiti:

- `registraPrestito(String, String, LocalDate)` - Registra nuovo prestito con validazioni complete
- `registraRestituzione(Prestito)` - Registra restituzione e aggiorna stato
- `getPrestitiAttivi(): List<Prestito>` - Restituisce prestiti attivi ordinati per scadenza

Metodi di persistenza:

- `salvaDati()`: Serializza l'intera biblioteca su file
- `caricaDati()`: Deserializza i dati all'avvio

Note sulla progettazione:

1. Tutti i metodi di modifica invocano automaticamente `salvaDati()` per garantire persistenza immediata
2. Le validazioni sono centralizzate in questa classe per garantire consistenza
3. I metodi di ricerca restituiscono sempre nuove liste per prevenire modifiche esterne
4. L'ordinamento è gestito a questo livello per separare la logica di presentazione dal model

2.2.2 Package Controller

Classe MainController

Responsabilità: Gestisce la navigazione principale dell'applicazione, coordinando il caricamento delle diverse schermate nel layout principale.
Attributi:

- `mainBorderPane: BorderPane` - Contenitore principale dell'interfaccia (annotato @FXML)

Metodi pubblici:

- `initialize()`: Carica la schermata iniziale (gestione libri)

- `apriGestioneLibri()`: Carica la view dei libri nel centro
- `apriGestioneUtenti()`: Carica la view degli utenti nel centro
- `apriGestionePrestiti()`: Carica la view dei prestiti nel centro
- `mostraInformazioni()`: Mostra dialog con info applicazione
- `esci()`: Chiude l'applicazione
- `caricaSchermata(String)`: Metodo privato per caricare FXML dinamicamente

Classe LibroController

Responsabilità: Gestisce tutte le operazioni relative ai libri nell'interfaccia principale (visualizzazione, ricerca, inserimento, modifica, eliminazione).

Attributi principali:

- `biblioteca: Biblioteca` - Riferimento al model
- `tabellaLibri: TableView<Libro>` - Tabella principale (annotato @FXML)
- `listaLibri: ObservableList<Libro>` - Lista osservabile per binding
- `fieldRicerca: TextField` - Campo ricerca (annotato @FXML)
- `comboCriterio: ComboBox<String>` - Criterio ricerca (annotato @FXML)
- Varie `TableColumn` per le colonne della tabella

Metodi pubblici principali:

- `initialize()`: Configura tabella, combo e carica dati iniziali
- `inserisciLibro()`: Apre dialog per nuovo libro
- `modificaLibro()`: Apre dialog per modifica libro selezionato
- `eliminaLibro()`: Elimina libro con conferma
- `cercaLibro()`: Esegue ricerca secondo criterio selezionato
- `mostraTuttiLibri()`: Reset ricerca, mostra tutti i libri
- `aggiornaTabella()`: Ricarica dati da Biblioteca e aggiorna view
- `apriDialogLibro(Libro)`: Metodo privato per gestire dialog

Note implementative:

- La tabella evidenzia visivamente i libri non disponibili (sfondo rosso)
- L'ordinamento è sempre mantenuto (alfabetico per titolo)
- Tutte le operazioni richiedono conferma esplicita

Classe LibroDialogController

Responsabilità: Gestisce il dialog per inserimento e modifica di un libro, inclusa la gestione della lista autori.

Attributi principali:

- `biblioteca: Biblioteca` - Riferimento al model
- `libroCorrente: Libro` - Libro in modifica (null se nuovo)
- `confermato: boolean` - Flag per comunicare l'esito al chiamante
- `autori: ObservableList<Libro.Autore>` - Lista autori in editing
- Vari `TextField` annotati @FXML per i campi del form
- `listaAutori: ListView<Libro.Autore>` - Lista visuale autori

Metodi pubblici principali:

- `initialize()`: Inizializza lista autori osservabile
- `setLibro(Libro)`: Precompila i campi se in modifica
- `aggiungiAutore()`: Aggiunge autore alla lista temporanea
- `rimuoviAutore()`: Rimuove autore selezionato
- `conferma()`: Valida e salva il libro
- `annulla()`: Chiude senza salvare
- `isConfermato(): boolean` - Comunica l'esito al chiamante

Classe UtenteController

Responsabilità: Gestisce tutte le operazioni relative agli utenti (visualizzazione, ricerca, inserimento, modifica, eliminazione).

Struttura: Analoga a `LibroController`, adattata per l'entità Utente.

Attributi: Simili a LibroController (tabella, lista osservabile, campi ricerca)

Metodi: Stessa struttura di LibroController per consistenza

Classe UtenteDialogController

Responsabilità: Gestisce il dialog per inserimento e modifica utenti.

Particolarità rispetto a LibroDialogController:

- Campo matricola disabilitato in modifica (requisito IF-2.2)
- Mostra info prestiti attivi solo in modalità modifica
- Mostra stato utente (può/non può effettuare prestiti)

Classe PrestitoController

Responsabilità: Gestisce la registrazione di nuovi prestiti, la restituzione e la visualizzazione dei prestiti attivi con statistiche.

Attributi principali:

- `tabellaPrestiti: TableView<Prestito>`
- `listaPrestiti: ObservableList<Prestito>`
- `labelTotale, labelRitardi, labelScadenza: Label` - Statistiche

Metodi pubblici principali:

- `initialize()`: Configura tabella con colorazione righe
- `registraPrestito()`: Apre dialog inline per nuovo prestito
- `registraRestituzione()`: Registra restituzione con conferma
- `aggiornaTabella()`: Ricarica prestiti ordinati per scadenza
- `aggiornaStatistiche()`: Calcola e mostra totali, ritardi, scadenze

Particolarità:

- Il dialog per nuovo prestito è inline (non separato) con ComboBox per utenti e libri
- Colorazione automatica righe: rosso (ritardo), giallo (scadenza 2gg), verde (attivo)
- Validazioni multiple: disponibilità libro, limite prestiti utente, data futura

2.2.3 Package Util

Classe AlertHelper

Responsabilità: Classe di utilità che centralizza la creazione e visualizzazione di messaggi all'utente.

Pattern: Utility class con metodi statici (non istanziabile)

Metodi pubblici statici:

- `mostraErrore(String, String)`: Alert di errore
- `mostraConferma(String, String)`: Alert di conferma operazione
- `mostraInfo(String, String)`: Alert informativo
- `mostraConfermaCancellazione(String): boolean` - Richiede conferma eliminazione

Vantaggi:

1. Evita duplicazione codice nei controller
2. Garantisce consistenza visiva dei messaggi
3. Facilita future modifiche (es. internazionalizzazione)
4. Applica il principio DRY (Don't Repeat Yourself)

2.2.4 Package Main

Classe MainApp

Responsabilità: Punto di ingresso dell'applicazione JavaFX, gestisce il ciclo di vita e il caricamento/salvataggio dati.

Attributi:

- `biblioteca: Biblioteca` - Riferimento al singleton

Metodi:

- `main(String[])`: Entry point applicazione
- `start(Stage)`: Inizializza GUI e carica dati
- `stop()`: Salva dati alla chiusura

Responsabilità lifecycle:

1. Ottiene istanza Biblioteca singleton
2. Carica dati persistenti all'avvio
3. Carica FXML principale e CSS
4. Configura Stage (dimensioni minime, titolo)
5. Salva dati automaticamente alla chiusura

2.3 Analisi di Coesione

La tabella seguente riassume il livello di coesione di ciascuna classe del sistema, con relativa giustificazione.

Classe	**Livello Coesione**	**Giustificazione**
MainApp	Funzionale	Gestisce un singolo compito: il ciclo di vita dell'applicazione JavaFX. Tutti i metodi contribuiscono a questo obiettivo (start, stop, caricamento dati).
Libro	Funzionale	Rappresenta un'unica entità (libro) con operazioni strettamente correlate: gestione copie disponibili, verifica disponibilità, formattazione autori.
Libro.Autore	Funzionale	Rappresenta un'unica entità semplice (autore) con attributi e metodi direttamente correlati (nome, cognome, toString).
Utente	Funzionale	Gestisce un'unica entità (utente) e le operazioni sui suoi prestiti attivi. Tutti i metodi operano sul concetto di "utente della biblioteca".

Classe	**Livello Coesione**	**Giustificazione**
Prestito	Funzionale	Gestisce il ciclo di vita di un prestito con metodi strettamente correlati: verifica stato, calcolo ritardi, registrazione restituzione.
Biblioteca	Funzionale	Funge da coordinatore centrale del sistema biblioteca. Nonostante gestisca tre entità (libri, utenti, prestiti), mantiene coesione funzionale perché tutte le operazioni sono parte del dominio "gestione biblioteca universitaria".
MainController	Procedurale	Gestisce la navigazione secondo una procedura: riceve input utente → carica schermata corrispondente. I metodi seguono una sequenza logica di navigazione.
LibroController	Comunicazionale	I metodi operano sulla stessa struttura dati (lista libri) ma svolgono compiti diversi: inserimento, modifica, eliminazione, ricerca, visualizzazione.
LibroDialogController	Funzionale	Gestisce un singolo compito: dialog per inserimento/modifica libro. Tutti i metodi e attributi contribuiscono a questo obiettivo specifico.
UtenteController	Comunicazionale	Analoga a LibroController: metodi operano sulla lista utenti con funzioni diverse ma correlate.
UtenteDialogController	Funzionale	Gestisce un singolo compito: dialog per inserimento/modifica utente.
PrestitoController	Comunicazionale	I metodi operano sulla lista prestiti con funzioni diverse: registrazione, restituzione, visualizzazione, calcolo statistiche.
AlertHelper	Funzionale	Centralizza la creazione di alert. Tutti i metodi statici operano sullo stesso concetto: visualizzazione messaggi all'utente.

Tabella 1: Analisi della coesione delle classi

Considerazioni:

- La maggior parte delle classi presenta **coesione funzionale**, il livello più alto
- I controller di gestione (Libro, Utente, Prestito) hanno **coesione comunicazionale**
- Nessuna classe presenta coesione di basso livello (logica, temporale, coincidentale)

2.4 Analisi di Accoppiamento

La tabella seguente analizza l'accoppiamento tra le principali coppie di classi del sistema.

Classi	**Tipo Accoppiamento**	**Giustificazione**
MainApp, Biblioteca	Dati	MainApp interagisce con Biblioteca solo tramite metodi pubblici (getInstance, caricaDati, salvaDati). Non accede a dettagli implementativi.
Biblioteca, Libro	Dati	Biblioteca gestisce liste di Libro accedendo solo all'interfaccia pubblica (getter, isDisponibile, incrementa/decrementaCopie).
Biblioteca, Utente	Dati	Biblioteca usa solo metodi pubblici di Utente (getter, haRaggiuntoLimite, aggiungi/rimuoviPrestito).
Biblioteca, Prestito	Dati	Biblioteca crea e gestisce Prestito tramite costruttore e metodi pubblici (isAttivo, registraRestituzione, getter).
Prestito, Libro	Dati	Prestito mantiene riferimento a Libro ma accede solo a getter pubblici (getTitolo, getIsbn). Non modifica lo stato del libro.
Prestito, Utente	Dati	Prestito mantiene riferimento a Utente e usa solo metodi pubblici (getNomeCognome, getMatricola).
Utente, Prestito	Dati	Utente mantiene lista di Prestito e restituisce copia difensiva (getPrestitiAttivi). Gestisce la lista tramite add/remove standard.
LibroController, Biblioteca	Dati	Il controller usa solo l'interfaccia pubblica di Biblioteca (getInstance, getTuttiLibri, cercaLibri, eliminaLibro). Passa solo parametri necessari.
LibroController, LibroDialogController	Controllo	LibroController controlla il flusso del dialog: lo apre con setLibro(), attende con showAndWait(), verifica esito con isConfermato().
LibroController, AlertHelper	Dati	Usa solo metodi statici pubblici di AlertHelper passando stringhe semplici. Nessuna dipendenza da stato interno.
UtenteController, Biblioteca	Dati	Interazione analoga a LibroController: solo interfaccia pubblica, parametri essenziali.
UtenteController, UtenteDialogController	Controllo	Pattern identico a LibroController-LibroDialogController per consistenza.

Classi Interessate	**Tipo Accoppiamento**	**Giustificazione**
UtenteController AlertHelper	Dati	Uso identico a LibroController per consistenza.
PrestitoController, Biblioteca	Dati	Usa metodi pubblici: getPrestitiAttivi, registraPrestito, registraRestituzione, getTuttiUtenti, getTuttiLibri.
PrestitoController, Utente, Libro	Dati	Accede solo a riferimenti oggetto e metodi pubblici (haRaggiuntoLimite, isDisponibile, toString per visualizzazione).
PrestitoController, AlertHelper	Dati	Uso consistente con altri controller.
LibroDialogController, Biblioteca	Dati	Usa solo aggiungiLibro e modificaLibro passando oggetti Libro completi.
UtenteDialogController, Biblioteca	Dati	Usa solo aggiungiUtente e modificaUtente passando oggetti Utente completi.

Tabella 2: Analisi dell'accoppiamento tra classi

Considerazioni:

- Tutto l'accoppiamento è di tipo **Dati** o **Controllo**, i livelli più bassi e desiderabili
- **Nessun accoppiamento** di tipo Content, Common o Stamp
- I controller sono accoppiati solo con Biblioteca e AlertHelper, mantenendo basso il numero totale di dipendenze
- L'uso dell'interfaccia pubblica e il passaggio di parametri essenziali garantiscono basso accoppiamento

2.5 Scelte Progettuali e Principi di Buona Progettazione

2.5.1 Principi SOLID

Single Responsibility Principle (SRP) Ogni classe del sistema ha una singola responsabilità ben definita:

- **Libro**: Gestisce solo i dati e la logica di un libro
- **Utente**: Gestisce solo i dati e la logica di un utente
- **Prestito**: Gestisce solo la logica temporale di un prestito
- **Biblioteca**: Coordina le entità e gestisce persistenza
- **AlertHelper**: Gestisce solo la visualizzazione messaggi
- Ogni Controller gestisce solo le interazioni per una specifica sezione dell'interfaccia

Beneficio: Modifiche a una responsabilità impattano una sola classe. Ad esempio, cambiare il formato di visualizzazione degli autori richiede modifiche solo in `Libro.getAutoriAsString()`.

Open/Closed Principle (OCP) Il sistema applica questo principio attraverso:

1. **Incapsulamento**: Tutti gli attributi sono privati, accessibili solo tramite metodi pubblici
2. **Interfacce stabili**: I metodi pubblici delle classi Model forniscono un contratto stabile
3. **Estensibilità**: È possibile estendere le funzionalità senza modificare il codice esistente

Esempio: Per aggiungere un nuovo tipo di ricerca libri, si potrebbe estendere `Biblioteca.cercaLibri()` aggiungendo un nuovo criterio senza modificare i criteri esistenti.

Interface Segregation Principle (ISP) Anche se non sono definite interfacce esplicite, il principio è applicato implicitamente:

- Ogni classe espone solo i metodi necessari ai suoi client
- `AlertHelper` fornisce metodi specifici per ogni tipo di messaggio invece di un unico metodo generico
- I controller dialogano con `Biblioteca` solo attraverso i metodi di cui hanno bisogno

Dependency Inversion Principle (DIP) Il principio è applicato nella separazione Model-View-Controller:

- I Controller (alto livello) dipendono dall'interfaccia pubblica di Biblioteca (astrazione)
- I Controller non dipendono da dettagli implementativi (come ArrayList vs LinkedList)
- L'uso di `List` invece di implementazioni concrete nelle signature dei metodi

2.5.2 Altri Principi Applicati

KISS (Keep It Simple, Stupid) Il sistema mantiene la semplicità attraverso:

- Classi con poche responsabilità e metodi brevi
- Algoritmi diretti (es. ricerca lineare nelle liste)
- Strutture dati semplici (ArrayList, non strutture complesse)
- Logica chiara e facilmente comprensibile

Esempio: La ricerca libri usa un semplice `stream().filter()` invece di strutture dati complesse o indici.

DRY (Don't Repeat Yourself) Evitata la duplicazione attraverso:

- **AlertHelper**: Centralizza la logica di creazione alert invece di ripeterla in ogni controller
- **Metodi comuni**: `aggiornaTabella()` presente in ogni controller ma implementato localmente (non duplicato)
- **Validazioni**: Centralizzate in `Biblioteca` invece di sparse nei controller

YAGNI (You Aren't Gonna Need It) Il sistema implementa solo le funzionalità richieste dai requisiti:

- Nessuna generalizzazione prematura (es. non è presente un sistema di ruoli utente)
- Persistenza semplice tramite serializzazione (non database)
- Interfaccia italiana (no internazionalizzazione non richiesta)

Separazione delle Responsabilità (Separation of Concerns) Applicata attraverso:

- **MVC**: Separazione netta tra Model, View e Controller
- **Package**: Organizzazione logica del codice
- **Persistenza**: Gestita solo in Biblioteca, separata dalla logica di business

Principio della Minima Sorpresa Il codice è strutturato in modo intuitivo:

- Nomi di classi e metodi descrittivi (`registraPrestito`, `isInRitardo`)
- Convenzioni Java standard (camelCase, getter/setter)
- Comportamenti coerenti tra classi simili (tutti i controller CRUD hanno la stessa struttura)

Composition over Inheritance Il sistema privilegia la composizione:

- `Biblioteca` compone `Libro`, `Utente`, `Prestito` (non eredita)
- `Prestito` compone `Utente` e `Libro` (associazione)
- `Libro` compone `Autore` (classe annidata)

Unica ereditarietà: I controller estendono implicitamente le classi JavaFX (necessario per il framework).

2.5.3 Pattern di Design Applicati (Riepilogo)

1. **Singleton** (Biblioteca): Garantisce istanza unica del sistema
2. **Facade** (Biblioteca): Semplifica operazioni complesse multi-entità
3. **MVC** (Architettura): Separa Model, View, Controller
4. **Observer** (JavaFX ObservableList): Notifica automatica cambiamenti

2.5.4 Gestione degli Errori e Robustezza

Validazioni Centralizzate Tutte le validazioni critiche sono centralizzate in `Biblioteca`:

- Verifica unicità ISBN e matricola
- Controllo limiti (es. 3 prestiti max, anno valido)
- Verifica integrità referenziale (no eliminazione con prestiti attivi)
- Validazione formati (email, date)

Gestione Eccezioni

- Tutti i metodi che possono fallire lanciano `Exception` con messaggi chiari
- I controller catturano le eccezioni e mostrano messaggi user-friendly tramite AlertHelper
- Nessun crash dell'applicazione per errori utente

Persistenza Sicura

- Salvataggio automatico dopo ogni modifica
- Try-catch per gestire errori I/O
- Messaggi di errore in caso di fallimento (non perdita silenziosa)

2.5.5 Qualità del Design: Riepilogo

Attributo	**Come è garantito**
Coesione	Alta coesione funzionale nella maggior parte delle classi
Accoppiamento	Basso accoppiamento (solo Dati e Controllo)
Manutenibilità	Separazione responsabilità, codice chiaro, nomi descrittivi
Estensibilità	Interfacce stabili, principio Open-Closed, incapsulamento
Riusabilità	Classi Model riusabili, AlertHelper utility
Testabilità	Model separato da UI, logica centralizzata
Robustezza	Validazioni, gestione eccezioni, persistenza sicura

Tabella 3: Attributi di qualità del design

3 Modello Dinamico

3.1 Introduzione

Il modello dinamico descrive il comportamento del sistema durante l'esecuzione, mostrando le interazioni tra gli oggetti per realizzare i casi d'uso principali. Vengono utilizzati i **diagrammi di sequenza** per rappresentare il flusso di messaggi tra le istanze delle classi.

3.2 Diagrammi di Sequenza

3.2.1 UC-1/UC-6: Inserimento Libro/Utente

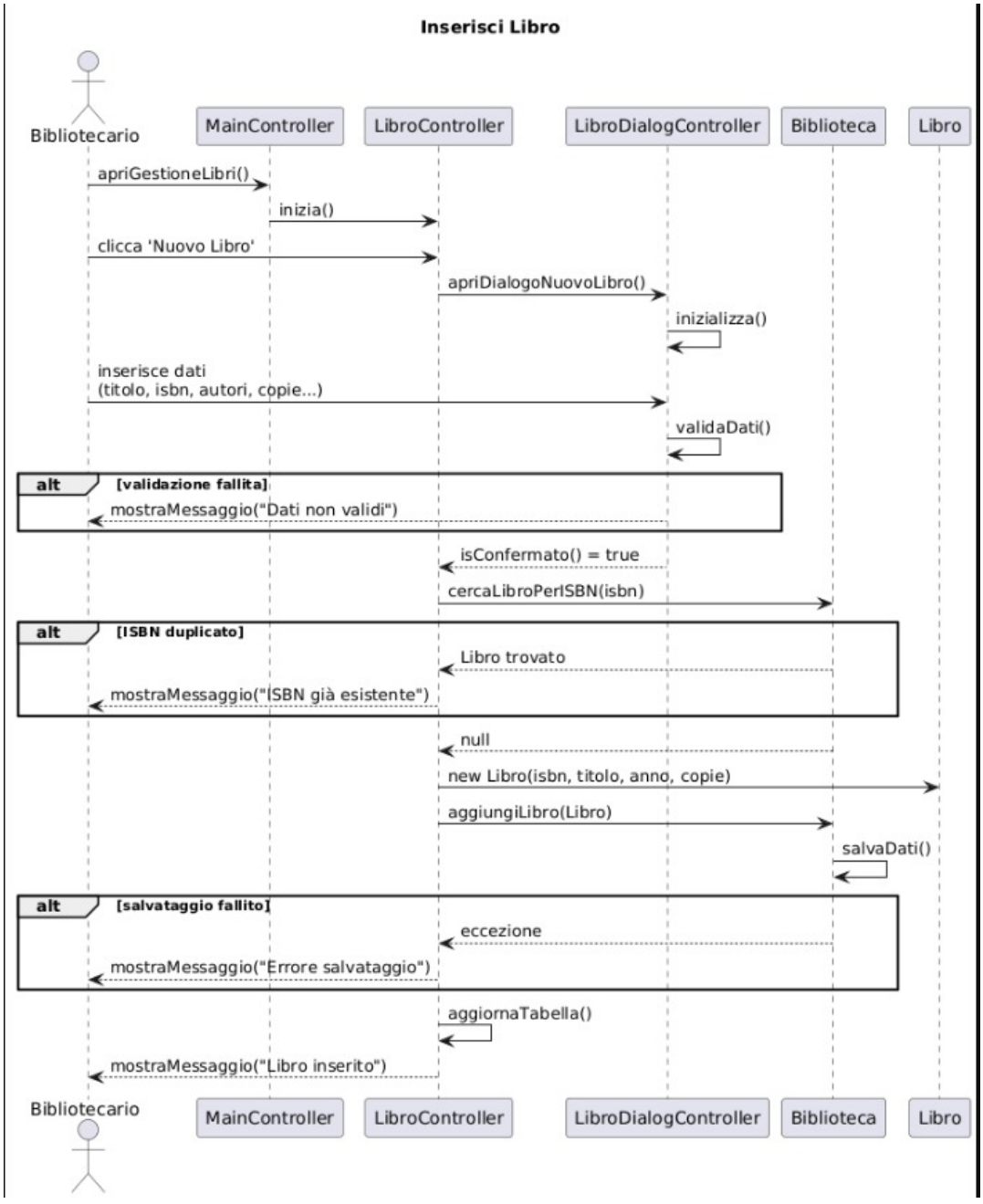

Figura 4: Diagramma di sequenza: Inserimento Libro/Utente

Nota: Il diagramma rappresenta il caso d'uso UC-1 (Inserimento Libro). Il processo per UC-6 (Inserimento Utente) è del tutto analogo, con le seguenti differenze:

- Controller utilizzato: `UtenteController` invece di `LibroController`
- Dialog utilizzato: `UtenteDialogController` invece di `LibroDialogController`
- Entità creata: `Utente` invece di `Libro`
- Validazioni specifiche: matricola univoca ed email valida invece di ISBN univoco

Descrizione del flusso:

1. Il Bibliotecario seleziona "Inserisci Libro" (o "Inserisci Utente") dal controller principale
2. Il `LibroController` (o `UtenteController`) invoca `apriDialogLibro(null)`
3. Viene caricato il dialog tramite FXMLLoader
4. Il `LibroDialogController` viene inizializzato con campi vuoti
5. L'utente compila i campi (titolo, ISBN, autori, anno, copie)
6. L'utente clicca "Conferma"
7. Il DialogController valida i dati:
   - Verifica campi obbligatori non vuoti
   - Controlla formati (anno numerico, range valido)
   - Verifica almeno un autore presente
8. Se validazione fallisce: mostra messaggio errore e torna al punto 5
9. Se validazione OK: crea nuovo oggetto `Libro`
10. Invoca `biblioteca.aggiungiLibro(nuovoLibro)`
11. `Biblioteca` verifica unicità ISBN:
    - Se ISBN duplicato: lancia eccezione
    - Se OK: aggiunge libro alla lista
12. `Biblioteca` invoca `salvaDati()` per persistenza
13. Se salvataggio fallisce: mostra messaggio errore
14. Se tutto OK: il dialog si chiude restituendo `confermato = true`
15. Il controller principale invoca `aggiornaTabella()`
16. La tabella viene aggiornata con il nuovo libro
17. Mostra messaggio di conferma all'utente

Flussi alternativi:

- **Validazione fallita**: Il sistema mostra un alert specifico e l'utente può correggere i dati
- **ISBN duplicato**: Viene mostrato errore "ISBN già esistente"
- **Errore salvataggio**: Viene mostrato errore e il libro non viene aggiunto
- **Annulla**: L'utente può annullare in qualsiasi momento, il dialog si chiude senza salvare

Note di design:

- La validazione è distribuita: controlli formali nel DialogController, controlli di business in Biblioteca
- Il pattern di comunicazione controller-dialog è consistente per tutti i casi d'uso di inserimento
- Il salvataggio è automatico e trasparente all'utente

3.2.2 UC-2/UC-7: Modifica Libro/Utente

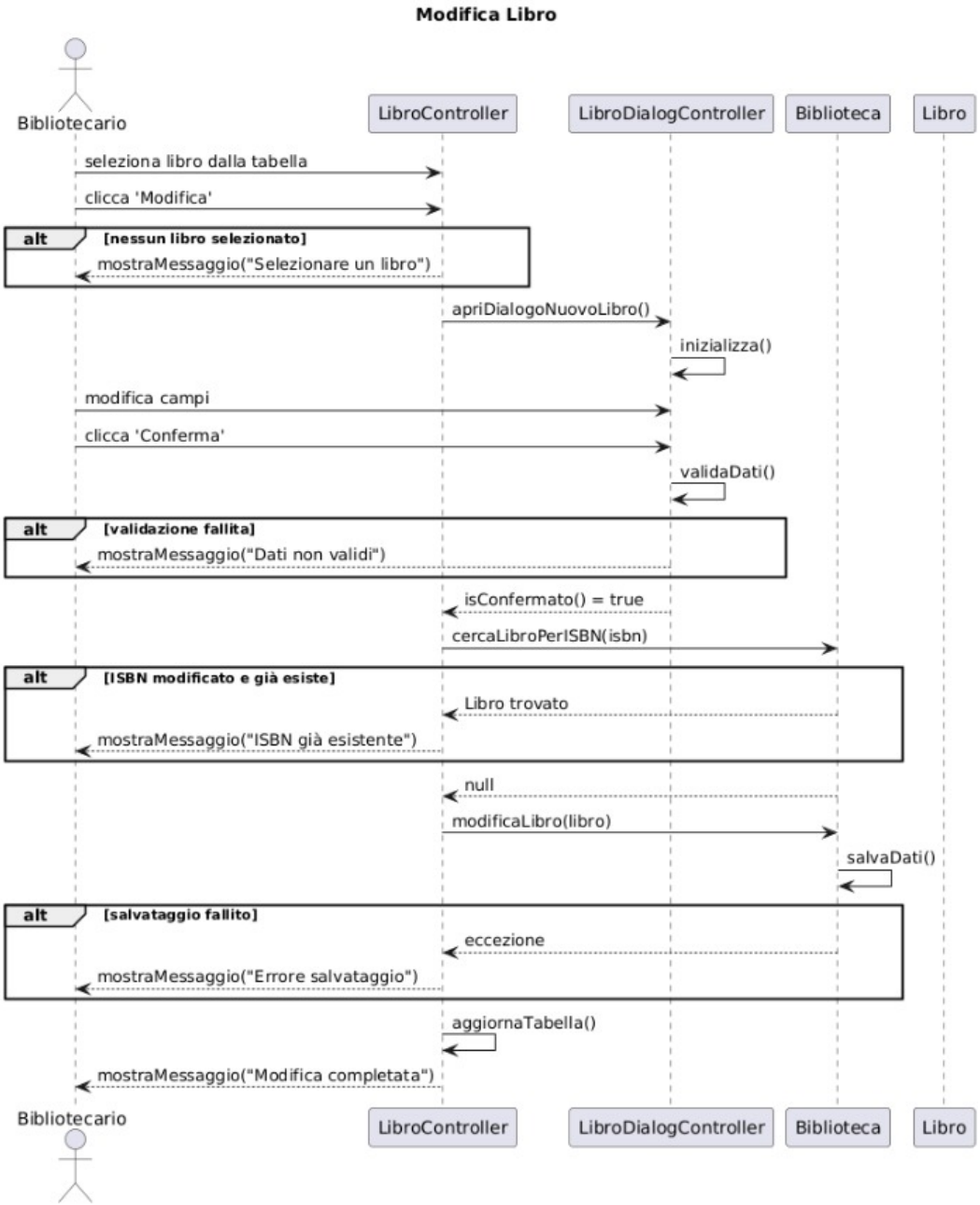

Figura 5: Diagramma di sequenza: Modifica Libro/Utente

Nota: Il diagramma illustra UC-2 (Modifica Libro). Il processo per UC-7 (Modifica Utente) è del tutto analogo, con `UtenteController/Dialog` al posto di `LibroController/Dialog`.
Particolarità:

- Per Libro: ISBN non modificabile
- Per Utente: Matricola non modificabile (requisito IF-2.2)

Descrizione del flusso:

1. Il Bibliotecario seleziona un libro/utente dalla tabella
2. Clicca "Modifica"
3. Il controller verifica che un elemento sia selezionato:
    - Se nessuna selezione: mostra errore e termina
4. Invoca `apriDialogLibro(libroSelezionato)`
5. Il dialog viene inizializzato con `setLibro(libro)`
6. I campi vengono precompilati con i dati esistenti:
    - ISBN disabilitato (non modificabile per i libri)
    - Matricola disabilitata (non modificabile per gli utenti)
7. L'utente modifica uno o più campi
8. Clicca "Conferma"
9. Il DialogController valida i nuovi dati
10. Se validazione OK: invoca `biblioteca.modificaLibro(libroCorrente)`
11. `Biblioteca` cerca il libro nella lista per ISBN
12. Se trovato: sostituisce l'oggetto con quello modificato
13. Invoca `salvaDati()`
14. Il dialog si chiude con `confermato = true`
15. Il controller aggiorna la tabella
16. Mostra messaggio di conferma

Particolarità:

- L'ISBN (per libri) e la matricola (per utenti) sono immutabili come da requisiti
- Il dialog distingue modalità inserimento/modifica tramite il parametro `libroCorrente`
- La validazione è identica a quella dell'inserimento

3.2.3 UC-3/UC-8: Eliminazione Libro/Utente

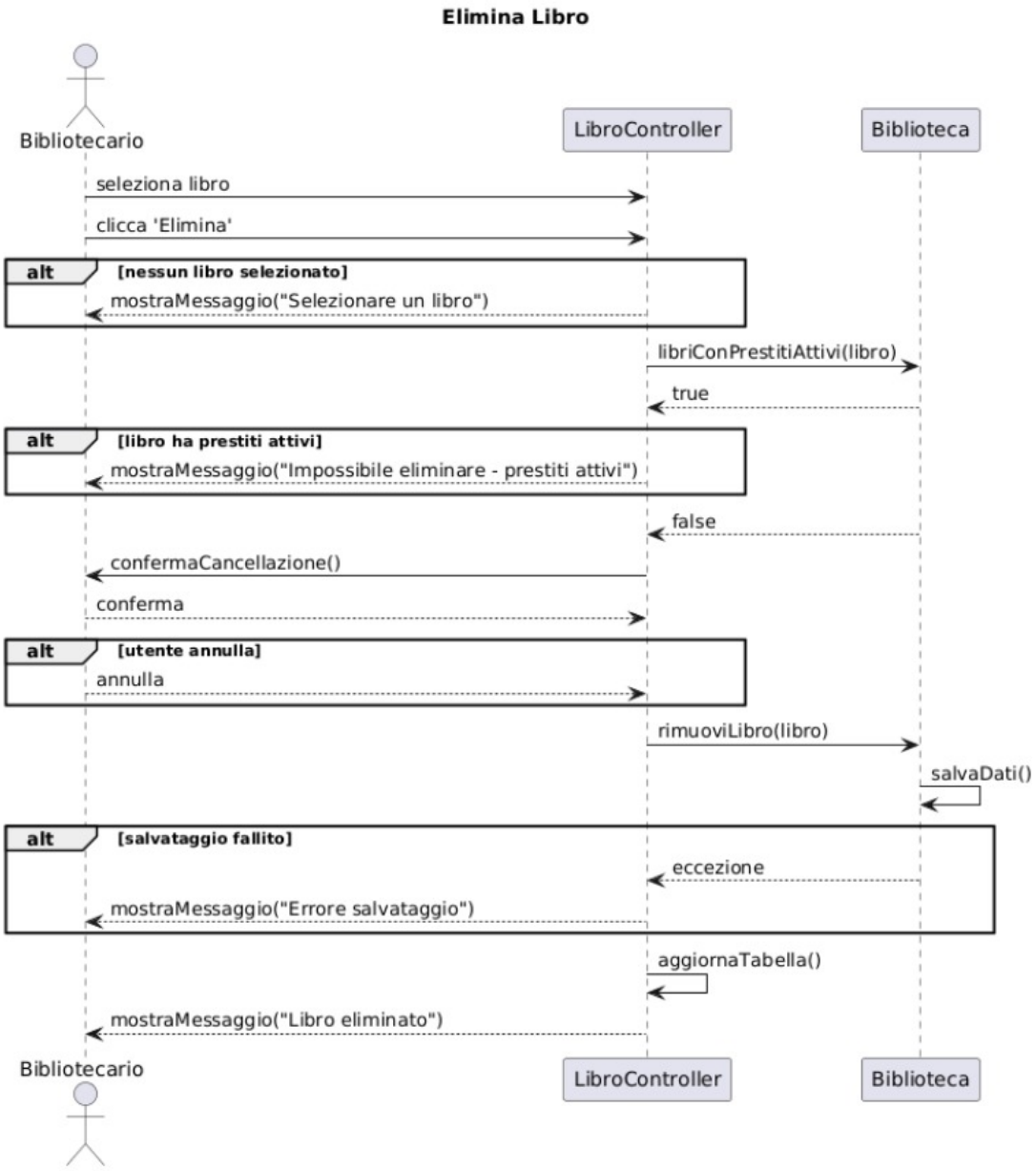

Figura 6: Diagramma di sequenza: Eliminazione Libro/Utente

Nota: Il diagramma rappresenta UC-3 (Eliminazione Libro). Il flusso per UC-8 (Eliminazione Utente) è identico, sostituendo `LibroController` con `UtenteController`.

Vincolo comune (Requisito FC-2): Non è possibile eliminare libri o utenti con prestiti attivi, garantendo l'integrità referenziale del sistema.

Descrizione del flusso:

1. Il Bibliotecario seleziona un libro/utente dalla tabella
2. Clicca "Elimina"

3. Il controller verifica che un elemento sia selezionato:
    - Se nessuna selezione: mostra errore e termina
4. Invoca `AlertHelper.mostraConfermaCancellazione(oggetto)`
5. Viene mostrato dialog di conferma con dettagli dell'elemento
6. L'utente può confermare o annullare:
    - Se annulla: l'operazione termina
7. Se conferma: il controller invoca `biblioteca.eliminaLibro(isbn)`
8. `Biblioteca` verifica la presenza di prestiti attivi:
    - Itera sui prestiti cercando prestiti attivi per quel libro
    - Se trova prestiti attivi: lancia eccezione "Impossibile eliminare: prestiti attivi"
9. Se nessun prestito attivo: rimuove il libro dalla lista
10. Invoca `salvaDati()`
11. Se salvataggio fallisce: mostra messaggio errore
12. Se OK: il controller aggiorna la tabella
13. Mostra messaggio di conferma eliminazione

Vincoli di integrità:

- **Requisito FC-2**: Non è possibile eliminare libri o utenti con prestiti attivi
- Questo garantisce l'integrità referenziale del sistema
- L'utente riceve feedback chiaro sul motivo del rifiuto

3.2.4 UC-11: Registrazione Prestito

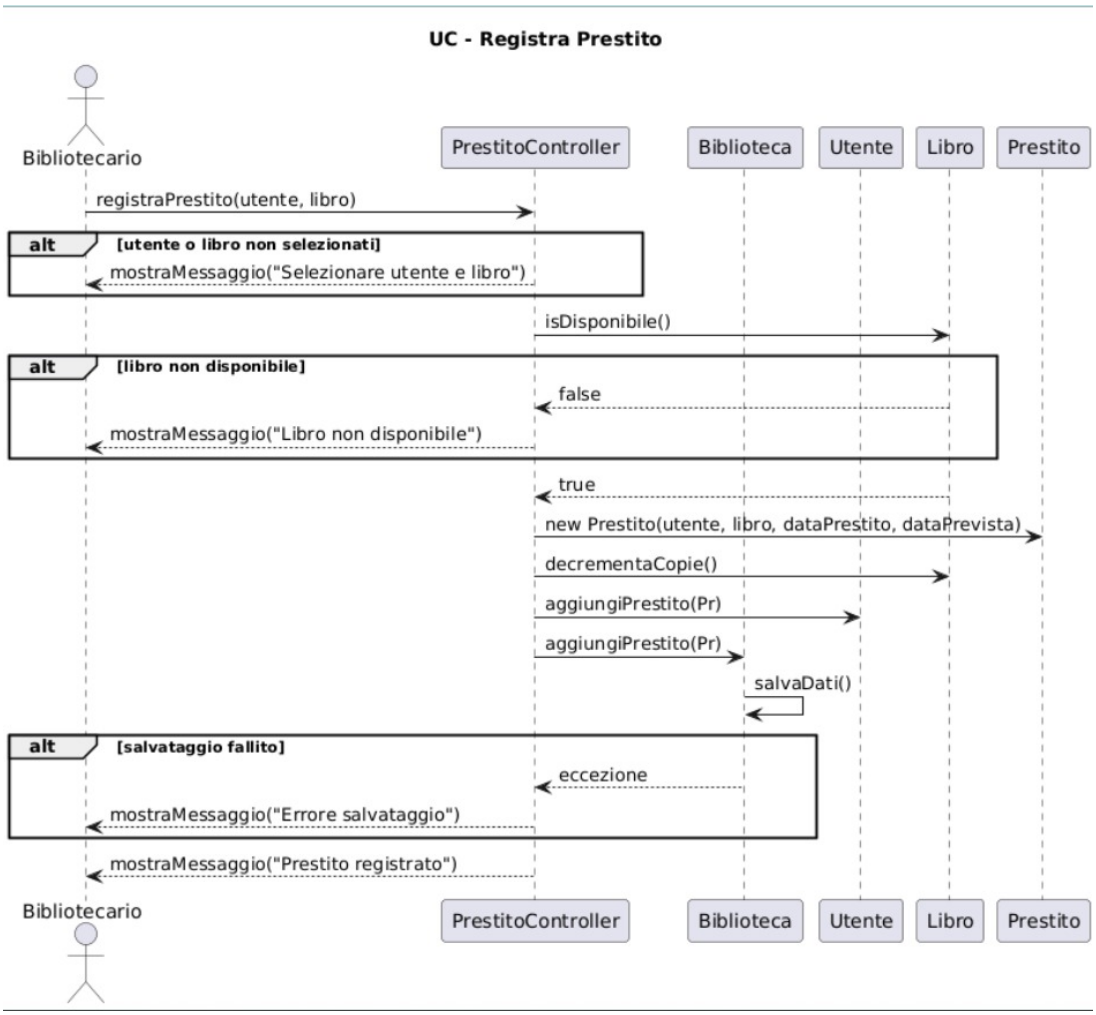

Figura 7: Diagramma di sequenza: Registrazione Prestito

Descrizione del flusso:

1. Il Bibliotecario clicca "Nuovo Prestito" in `PrestitoController`
2. Viene creato un dialog inline con:
   - ComboBox per selezione utente (popolata con `biblioteca.getTuttiUtenti()`)
   - ComboBox per selezione libro (popolata con `biblioteca.getTuttiLibri()`)
   - DatePicker per data restituzione (default: oggi + 14 giorni)
3. L'utente seleziona utente e libro
4. Durante la selezione, vengono mostrate info in tempo reale:
   - Per utente: "Prestiti: N/3" (verde se OK, rosso se limite raggiunto)
   - Per libro: "Copie disponibili: N/M" (verde se disponibile, rosso se 0)
5. L'utente imposta la data di restituzione

6. Clicca "Conferma"
7. Il controller verifica che tutti i campi siano compilati
8. Invoca `biblioteca.registraPrestito(matricola, isbn, data)`
9. `Biblioteca` esegue le validazioni (UC-14, UC-15):
    - **UC-14 - Verifica Disponibilità**: Invoca `libro.isDisponibile()`
    - Se false: lancia eccezione "Nessuna copia disponibile"
    - **UC-15 - Verifica Limite**: Invoca `utente.haRaggiuntoLimite()`
    - Se true: lancia eccezione "Limite prestiti raggiunto (3/3)"
    - Verifica data futura: Se `data.isBefore(LocalDate.now())` lancia eccezione
10. Se tutte le validazioni passano:
    - Crea nuovo oggetto `Prestito(utente, libro, data)`
    - Invoca `libro.decrementaCopie()` (UC-16)
    - Invoca `utente.aggiungiPrestito(prestito)`
    - Aggiunge prestito alla lista `prestiti`
11. Invoca `salvaDati()`
12. Il dialog si chiude
13. Il controller aggiorna la tabella prestiti
14. Mostra messaggio "Prestito registrato con successo"

Flussi alternativi:

- **Libro non disponibile**: Mostra errore specifico, prestito non creato
- **Limite prestiti**: Mostra errore "Utente ha già 3 prestiti attivi"
- **Data passata**: Mostra errore "La data deve essere futura"
- **Campi incompleti**: Mostra errore "Compila tutti i campi"

Coordinazione multi-entità: Questo caso d'uso dimostra come `Biblioteca` coordini operazioni su tre entità diverse (Utente, Libro, Prestito) mantenendo la consistenza del sistema.

3.2.5 UC-12: Registrazione Restituzione

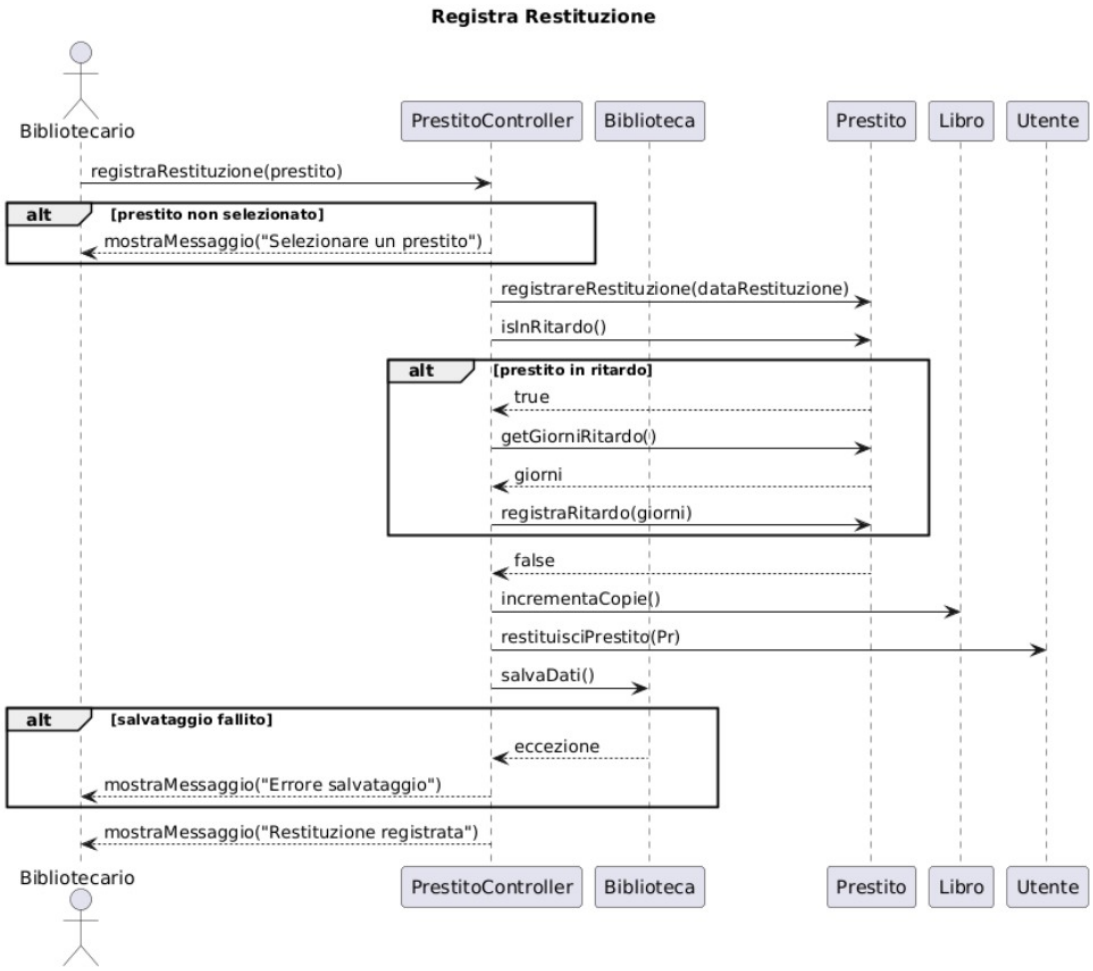

Figura 8: Diagramma di sequenza: Registrazione Restituzione

Descrizione del flusso:

1. Il Bibliotecario seleziona un prestito dalla tabella prestiti attivi
2. Clicca "Registra Restituzione"
3. Il controller verifica che un prestito sia selezionato
4. Invoca `prestito.isInRitardo()` per determinare lo stato
5. Mostra dialog di conferma con dettagli prestito:
   - Utente: Cognome Nome (matricola)
   - Libro: Titolo
   - Data prestito e scadenza
   - Stato: se in ritardo, mostra "ATTENZIONE: Prestito in ritardo di N giorni"
6. L'utente conferma la restituzione
7. Il controller invoca `biblioteca.registraRestituzione(prestito)`

8. `Biblioteca` verifica `prestito.isAttivo()`:
    - Se già chiuso: lancia eccezione "Prestito già chiuso"
9. Se attivo:
    - Invoca `prestito.registraRestituzione()` che imposta `dataRestituzioneEffettiva = LocalDate.now()`
    - Invoca `libro.incrementaCopie()` (UC-16)
    - Invoca `utente.rimuoviPrestito(prestito)`
10. Invoca `salvaDati()`
11. Il controller aggiorna la tabella (il prestito scompare dalla lista attivi)
12. Aggiorna le statistiche (totale, ritardi, scadenze)
13. Mostra messaggio "Restituzione registrata con successo"

Gestione ritardi:

- Il sistema calcola automaticamente i giorni di ritardo tramite `prestito.getGiorniRitardo()`
- L'informazione è mostrata all'utente per eventuale applicazione sanzioni (fuori scope del sistema)
- I prestiti in ritardo sono evidenziati visivamente nella tabella (sfondo rosso)

3.3 Considerazioni sul Modello Dinamico

3.3.1 Pattern di Interazione Comuni

Tutti i casi d'uso seguono pattern simili:

1. Interazione utente con controller
2. Validazioni locali nel controller/dialog
3. Invocazione metodi Biblioteca
4. Validazioni di business in Biblioteca
5. Aggiornamento stato entità
6. Persistenza automatica
7. Aggiornamento vista
8. Feedback utente

Questa consistenza facilita:

- Comprensione del codice
- Manutenzione
- Aggiunta nuove funzionalità
- Testing

4 Design dell'Interfaccia Utente

4.1 Introduzione

L'interfaccia utente è stata progettata seguendo i principi di usabilità e le linee guida dei requisiti (UI-1.1 a UI-2.1). I wireframe rappresentano il design concettuale delle schermate principali prima dell'implementazione effettiva.

4.2 Struttura Generale

Tutte le schermate condividono una struttura comune basata su un layout `BorderPane`:

- **Top**: MenuBar con File, Gestione, Aiuto
- **Center**: Area contenuto dinamico (caricata tramite navigazione)
- **Bottom**: Status bar con informazioni di stato

Questa struttura garantisce:

- Coerenza visiva tra le schermate
- Navigazione sempre accessibile
- Feedback continuo sullo stato del sistema

4.3 Wireframe delle Schermate Principali

4.3.1 WF-01: Gestione Libri

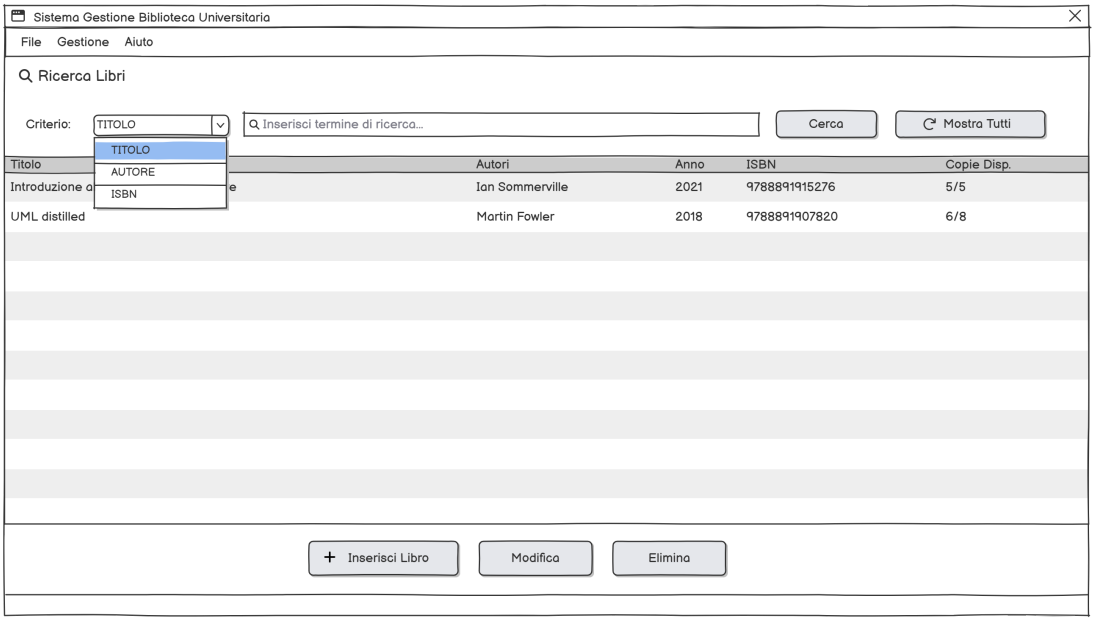

Figura 9: Wireframe: Gestione Libri

Requisiti implementati: BF-1, IF-1.1 a IF-1.5, UC-1 a UC-5

Elementi principali:

1. **Area Ricerca** (in alto):
    - Label: "Ricerca Libri"
    - ComboBox criterio: TITOLO — AUTORE — ISBN
    - TextField per inserimento termine
    - Bottoni: "Cerca" e "Mostra Tutti"
2. **Tabella Risultati** (centro):
    - Colonne: Titolo — Autori — Anno — ISBN — Copie Disp.
    - Ordinamento alfabetico per titolo
    - Evidenziazione libri non disponibili (sfondo rosso chiaro)
3. **Barra Azioni** (in basso):
    - "Inserisci Libro" (apre dialog)
    - "Modifica" (richiede selezione)
    - "Elimina" (richiede selezione e conferma)

Interazioni principali:

- Click "Inserisci" → Apre WF-06 (Dialog Libro)
- Selezione libro + "Modifica" → Apre WF-06 precompilato
- Selezione libro + "Elimina" → Mostra WF-04 (Conferma)
- Ricerca per criterio → Filtra tabella
- "Mostra Tutti" → Reset filtri

Design rationale:

- Ricerca sempre visibile per accesso rapido
- Criteri limitati a quelli specificati nei requisiti
- Bottoni disabilitati quando azione non disponibile
- Colore rosso comunica immediatamente "non disponibile"

4.3.2 WF-02: Gestione Utenti

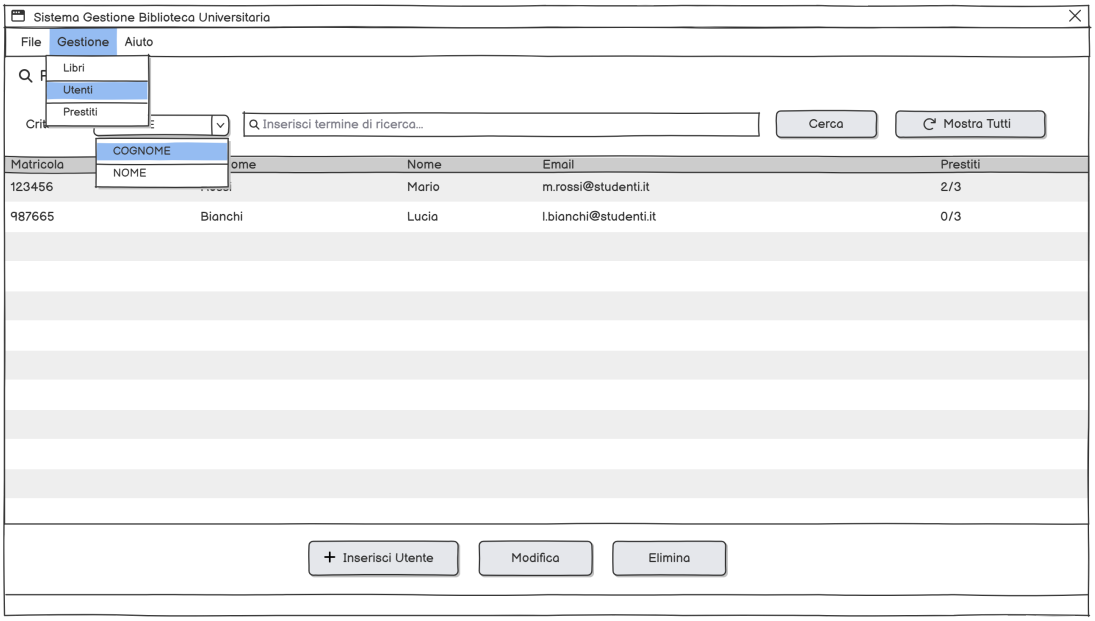

Figura 10: Wireframe: Gestione Utenti

Requisiti implementati: BF-2, IF-2.1 a IF-2.5, UC-6 a UC-10

Elementi principali:

1. **Area Ricerca**:
   - ComboBox criterio: COGNOME — MATRICOLA
   - TextField ricerca
   - Bottoni "Cerca" e "Mostra Tutti"
2. **Tabella Utenti**:
   - Colonne: Matricola — Cognome — Nome — Email — Prestiti
   - Ordinamento per cognome, poi nome
   - Colonna "Prestiti" mostra formato "N/3"
   - Evidenziazione utenti con limite raggiunto (sfondo giallo)
3. **Barra Azioni**:
   - "Inserisci Utente"
   - "Modifica"
   - "Elimina"

Particolarità:

- La colonna "Prestiti" fornisce feedback visivo immediato sullo stato utente
- Giallo per limite raggiunto comunica "attenzione" ma non blocco
- Matricola non ricercabile per nome (solo ricerca esatta)

4.3.3 WF-03: Gestione Prestiti

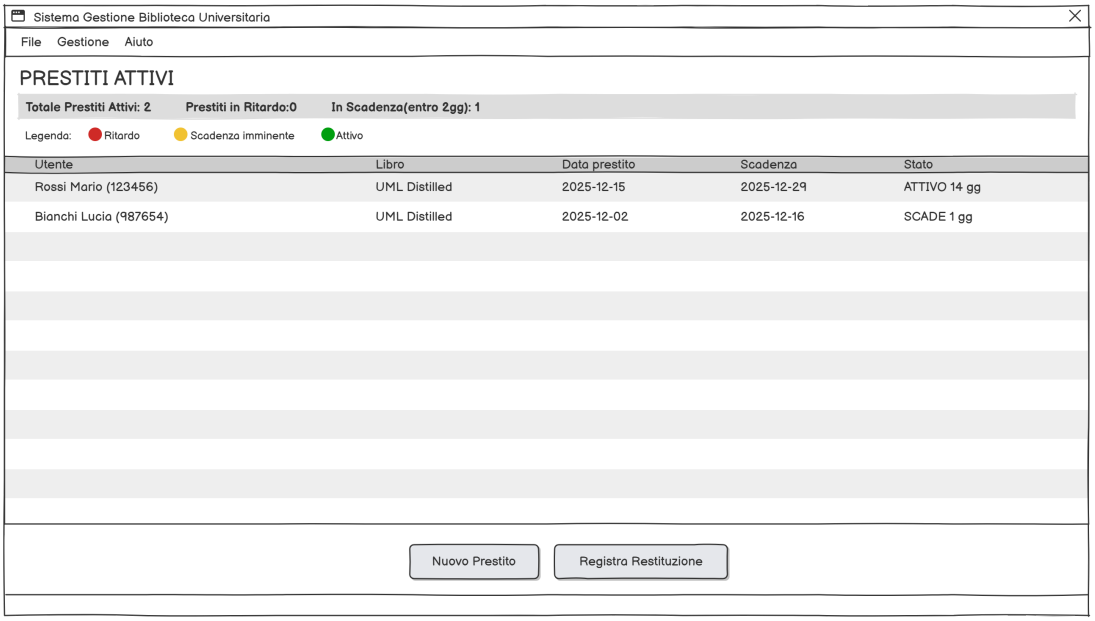

Figura 11: Wireframe: Gestione Prestiti

Requisiti implementati: BF-3, BF-4, BF-5, IF-3.1 a IF-3.3, UC-11 a UC-13
Elementi principali:

1. **Header Informativo**:
   - Titolo "PRESTITI ATTIVI"
   - Statistiche in tempo reale:
     - "Totale Prestiti Attivi: N"
     - "Prestiti in Ritardo: N" (rosso se ¿ 0)
     - "In Scadenza (entro 2gg): N"
   - Legenda colori: Ritardo — Scadenza imminente — Attivo
2. **Tabella Prestiti Attivi**:
   - Colonne: Utente — Libro — Data Prestito — Scadenza — Stato
   - Ordinamento per data scadenza (urgenti prima)
   - Colorazione righe:
     - Rosso: Prestito in ritardo (requisito UI-1.5)
     - Giallo: Scadenza entro 2 giorni
     - Verde: Prestito regolare
   - Colonna "Stato" mostra descrizione testuale:
     - "RITARDO N gg"
     - "SCADE N gg"
     - "ATTIVO N gg"

3. **Barra Azioni**:
    - "Nuovo Prestito" (apre WF-07)
    - "Registra Restituzione" (richiede selezione)
    - "Aggiorna" (refresh dati)

Design rationale:

- **Statistiche in evidenza**: Il bibliotecario vede immediatamente la situazione critica
- **Ordinamento per urgenza**: I prestiti che scadono prima sono in cima
- **Codifica colore universale**: Rosso = pericolo, Giallo = attenzione, Verde = OK
- **Doppia informazione**: Colore riga + testo stato per accessibilità
- **Legenda sempre visibile**: Evita ambiguità nell'interpretazione

Particolarità implementative:

- La colorazione è dinamica: si aggiorna automaticamente quando un prestito va in ritardo
- Le statistiche si ricalcolano ogni volta che la tabella viene aggiornata
- Il calcolo dei giorni è sempre relativo alla data corrente

4.4 Wireframe dei Dialog

4.4.1 WF-04: Alert e Dialog di Sistema

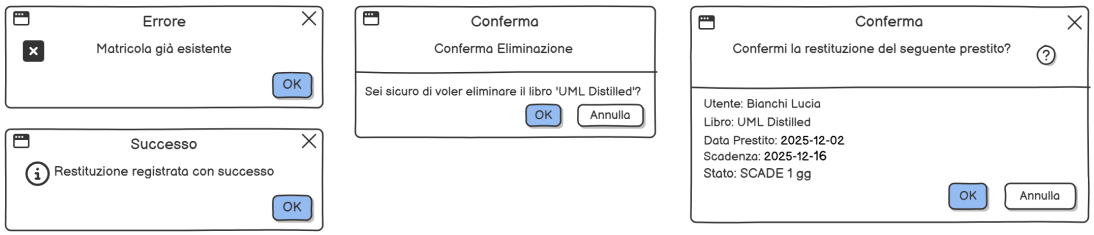

Figura 12: Esempi di Dialog di Errore, Conferma, Successo

Requisito implementato: UI-1.6, FC-1

Tipologie di messaggi:

1. **Errori di Validazione Selezione**
    - **Tipo**: Error Alert
    - **Titolo**: "Errore"
    - **Messaggio**: "Seleziona un [elemento] da [operazione]"

- **Esempi**:
  - "Seleziona un libro da modificare"
  - "Seleziona un prestito da restituire"
- **Azioni**: [OK]

2. Errori di Validazione Campi

- **Campi obbligatori**:
  - "Il titolo è obbligatorio"
  - "L'ISBN è obbligatorio"
  - "Inserisci almeno un autore"
- **Unicità**:
  - "ISBN già esistente" (requisito DF-1.1)
  - "Matricola già esistente" (requisito DF-1.2)
- **Range e formato**:
  - "Anno non valido (1000 - 2025)"
  - "Email non valida"
  - "Titolo troppo lungo (max 200 caratteri)"

3. Conferme di Cancellazione

- **Tipo**: Confirmation Alert
- **Titolo**: "Conferma Eliminazione"
- **Messaggio**: "Sei sicuro di voler eliminare [oggetto]?"
- **Dettagli**: Nome/titolo dell'elemento
- **Azioni**: [OK] [Annulla]

4. Errori di Integrità

- "Impossibile eliminare: prestiti attivi" (requisito FC-2)
- "Limite prestiti raggiunto (3/3)"
- "Nessuna copia disponibile"

5. Conferme di Successo

- **Tipo**: Information Alert
- **Titolo**: "Successo"
- **Esempi**:
  - "Libro inserito con successo"
  - "Prestito registrato con successo"
  - "Restituzione registrata con successo"
- **Azioni**: [OK]

Principi applicati:

- **Chiarezza**: Messaggi in italiano semplice (UI-2.1)
- **Specificità**: Indicazione precisa del problema
- **Azione suggerita**: L'utente sa cosa fare per risolvere
- **Consistenza**: Stesso formato per errori simili

4.4.2 WF-05: Dialog Inserisci/Modifica Utente

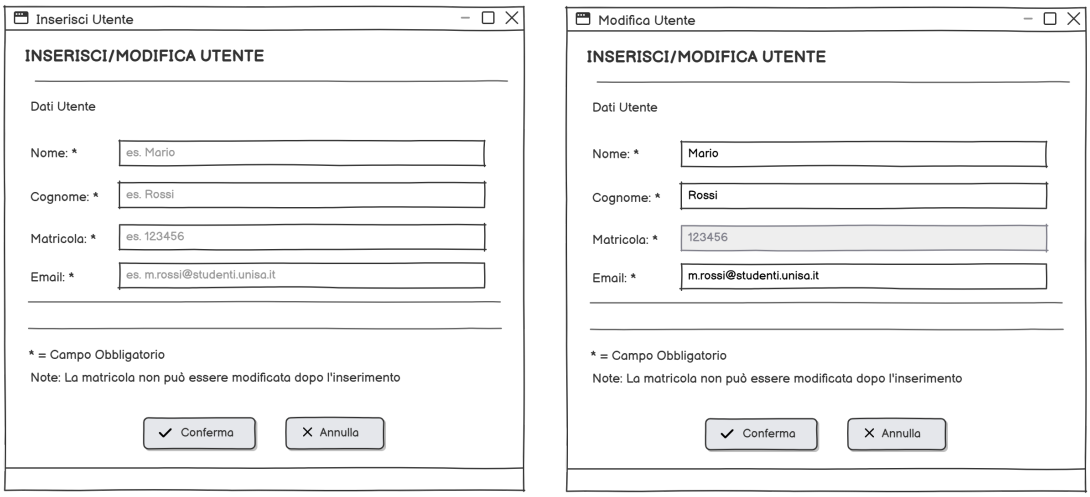

Figura 13: Wireframe: Dialog Inserisci/Modifica Utente

Requisiti implementati: IF-2.1, IF-2.2, DF-1.2, UC-6, UC-7
Sezione "Dati Utente":

- **Nome***: TextField, placeholder "es. Mario", max 50 caratteri
- **Cognome***: TextField, placeholder "es. Rossi", max 50 caratteri
- **Matricola***: TextField, placeholder "es. 123456", max 10 caratteri
  - Disabilitata in modalità modifica (requisito IF-2.2)

- **Email***: TextField, placeholder "es. m.rossi@studenti.unisa.it"

Sezione Informazioni (solo in modifica):

- Label "Prestiti Attivi: N"
- Label "Stato:" con indicatore:
  - "Può effettuare prestiti" (verde, se N ¡ 3)
  - "Limite prestiti raggiunto" (giallo, se N = 3)

Note informative:

- "* = Campo obbligatorio"
- "Nota: La matricola non può essere modificata dopo l'inserimento"

Azioni:

- "Conferma" (salva e chiude)
- "Annulla" (chiude senza salvare)

Differenze modalità:

Elemento	**Inserimento**	**Modifica**
Titolo dialog	"Inserisci Utente"	"Modifica Utente"
Campi	Vuoti	Precompilati
Matricola	Modificabile	Disabilitata
Info prestiti	Nascosta	Visibile

Tabella 4: Differenze tra modalità inserimento e modifica

4.4.3 WF-06: Dialog Inserisci/Modifica Libro

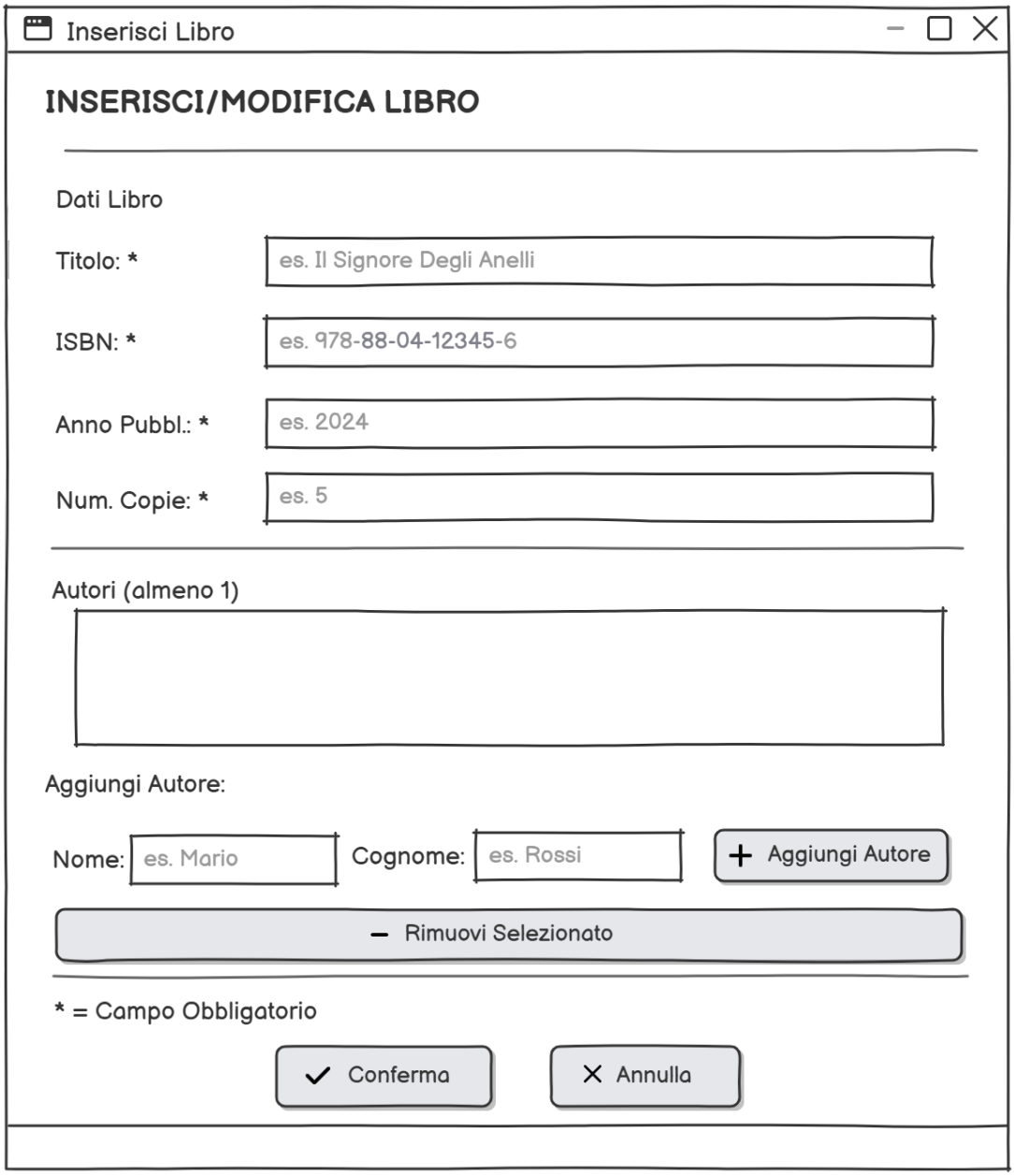

Figura 14: Wireframe: Dialog Inserisci/Modifica Libro

Requisiti implementati: IF-1.1, IF-1.2, DF-1.1, UC-1, UC-2
Sezione "Dati Libro":

- **Titolo***: TextField, max 200 caratteri
- **ISBN***: TextField, max 20 caratteri (disabilitato in modifica)
- **Anno Pubbl.***: TextField numerico, range 1000-2025
- **Num. Copie***: TextField numerico, ¿= 0

Sezione "Autori (almeno 1)":

- **ListView autori**: Mostra autori già inseriti
  - Formato visualizzazione: "Cognome Nome"
  - Selezione per rimozione
- **Aggiungi Autore**:
  - Campo "Nome": TextField, max 50 caratteri
  - Campo "Cognome": TextField, max 50 caratteri
  - Bottone "Aggiungi Autore"
- **Bottone "Rimuovi Selezionato"**: Rimuove autore dalla lista

Workflow gestione autori:

1. Inserire nome e cognome autore
2. Cliccare "Aggiungi Autore" → Appare in ListView
3. Ripetere per aggiungere più autori
4. Per rimuovere: selezionare dalla lista e cliccare "Rimuovi"
5. Validazione: almeno 1 autore obbligatorio alla conferma

Design rationale:

- **Lista dinamica**: Permette gestione flessibile numero autori
- **Feedback visivo**: L'utente vede immediatamente autori aggiunti
- **Rimozione facile**: Selezione + click invece di ricordare posizione
- **Validazione posticipata**: Non blocca durante inserimento, valida alla conferma

4.4.4 WF-07: Dialog Registra Nuovo Prestito

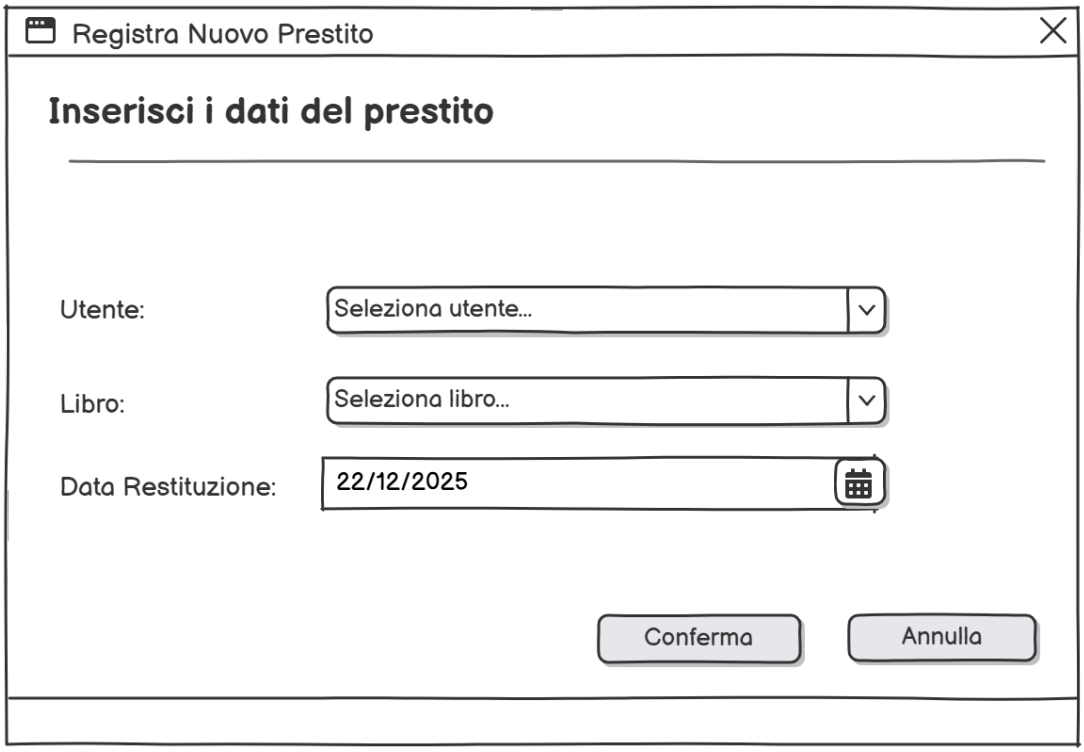

Figura 15: Wireframe: Dialog Registra Nuovo Prestito

Requisiti implementati: IF-3.1, BF-3, UC-11, UC-14, UC-15
Campi del form:

1. **Utente**:
   - ComboBox con lista utenti
   - Formato: "Cognome Nome (Matricola)"
   - Placeholder: "Seleziona utente..."
   - Info dinamica sotto il campo:
     - Se utente OK: "Prestiti: N/3" (verde)
     - Se limite: "Limite prestiti raggiunto (3/3)" (rosso)
2. **Libro**:
   - ComboBox con lista libri
   - Formato: "Titolo"
   - Placeholder: "Seleziona libro..."
   - Info dinamica:
     - Se disponibile: "Copie disponibili: N/M"
     - Se non disponibile: "Nessuna copia disponibile"
3. **Data Restituzione**:

- DatePicker
- Default: Data odierna + 14 giorni
- Formato: gg/mm/aaaa
- Vincolo: Solo date future

Validazioni automatiche (UC-14, UC-15):

1. **Disponibilità libro**: `libro.isDisponibile()`
   - Se false: Errore "Nessuna copia disponibile"
2. **Limite prestiti**: `utente.haRaggiuntoLimite()`
   - Se true: Errore "Limite prestiti raggiunto (3/3)"
3. **Data futura**: `data.isAfter(LocalDate.now())`
   - Se false: Errore "La data deve essere futura"
4. **Campi completi**: Tutti i campi devono essere compilati

Azioni:

- "Conferma": Valida e registra prestito
- "Annulla": Chiude senza salvare

Feedback in tempo reale:

- Durante la selezione di utente/libro, vengono mostrate immediatamente le info
- L'utente sa *prima* di confermare se il prestito è possibile
- Riduce frustrazioni: nessun tentativo "a vuoto"

4.5 Linee Guida di Usabilità Applicate

4.5.1 Coerenza (Consistency)

- **Layout**: Tutte le schermate CRUD seguono lo stesso schema (ricerca in alto, tabella al centro, azioni in basso)
- **Terminologia**: Stesso linguaggio per operazioni simili ("Inserisci", "Modifica", "Elimina")
- **Colori**: Rosso = errore/pericolo, Verde = successo/OK, Giallo = attenzione
- **Icone**: Uso consistente (+ = aggiungi, V = conferma, X = annulla)

4.5.2 Feedback Visivo

- **Evidenziazione stato**: Righe colorate nelle tabelle (libri non disponibili, utenti al limite, prestiti in ritardo)
- **Messaggi immediati**: Alert per conferme ed errori
- **Info contestuali**: Statistiche in tempo reale nella gestione prestiti
- **Disabilitazione bottoni**: Bottoni grigi quando azione non disponibile

4.5.3 Prevenzione Errori

- **Campi obbligatori**: Marcati con asterisco
- **Placeholder**: Esempi di formato atteso
- **ComboBox**: Vincola scelte a opzioni valide
- **Conferme**: Richieste per azioni distruttive (eliminazione)
- **Campi disabilitati**: ISBN e matricola non modificabili dopo creazione
- **Validazione in tempo reale**: Info disponibilità mostrate prima di confermare prestito

4.5.4 Riconoscimento piuttosto che Memorizzazione

- **Menu sempre visibile**: Non serve ricordare come navigare
- **Legenda colori**: Spiegazione sempre presente nella gestione prestiti
- **Note inline**: Spiegazioni direttamente nei dialog (es. "matricola non modificabile")
- **Statistiche visibili**: Stato sistema sempre evidente

4.5.5 Gestione Errori Graceful

- **Messaggi chiari**: Linguaggio semplice, non tecnico
- **Spiegazione causa**: "ISBN già esistente" invece di "Errore inserimento"
- **Suggerimento soluzione**: Cosa fare per risolvere
- **Non distruttivi**: Gli errori non causano perdita dati o chiusura dialog

4.6 Accessibilità

4.6.1 Contrasto e Colori

- Colori scelti con contrasto sufficiente per leggibilità
- Informazioni non veicolate *solo* tramite colore (anche testo descrittivo)
- Esempio: Prestiti in ritardo hanno sia sfondo rosso che testo "RITARDO N gg"

4.6.2 Dimensioni e Spaziatura

- Bottoni sufficientemente grandi per click facile
- Spaziatura adeguata tra elementi
- Font leggibile (14px base)

4.6.3 Navigazione da Tastiera

- Tab order logico nei form
- Supporto tasti scorciatoia standard (Enter = conferma, Esc = annulla)
- Selezione tabella tramite frecce

4.7 Responsive Design (Considerazioni)

Anche se l'applicazione è desktop, è stata considerata la ridimensionabilità:

- **Dimensioni minime**: 1000x700 pixel (requisito esplicito)
- **Layout adattivo**: BorderPane e HBox/VBox si adattano al ridimensionamento
- **Scroll automatico**: Tabelle con scroll se contenuto eccede area visibile
- **Colonne proporzionali**: Larghezze colonne in percentuale quando possibile