Documento di Specifica di Progettazione "Biblioteca Universitaria" (23)

Corso di Ingegneria del Software
Anno Accademico 2025/26

INTRODUZIONE

DESCRIZIONE DELLE INFORMAZIONI CONTENUTE NEL DOCUMENTO

Un documento di design di un progetto software è un elemento cruciale nel processo di sviluppo di quest'ultimo, che serve a guidare il team di sviluppo attraverso la progettazione e l'implementazione del sistema software. Questo documento include:

1. **Architettura del sistema**: questa sezione descrive l'architettura di alto livello del sistema, comprese le relazioni tra i vari componenti di quest'ultimo.
2. **Detailed design**: questa sezione fornisce un'elaborazione dettagliata di ogni componente, inclusi i sequence diagrams, activity diagrams ed eventuali altri diagrammi di supporto.
3. **Interfaccia**: questa sezione descrive l'interfaccia tra il sistema in sviluppo e l'utente, incluso il modo in cui l'utente interagisce con essa.

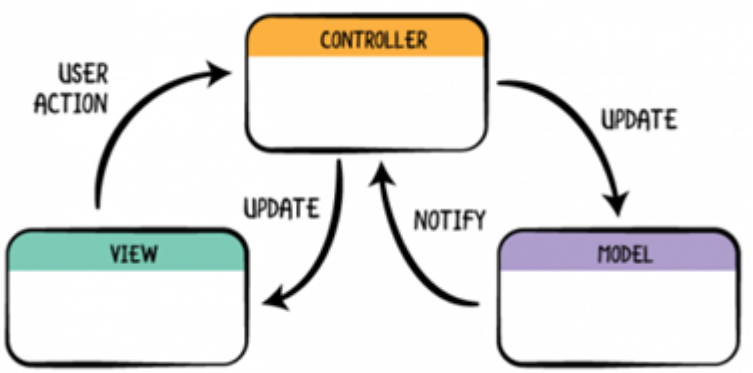
SCOPO DEL DOCUMENTO

Lo scopo principale di un documento di design è fornire una visione d'insieme dettagliata del sistema in sviluppo, in modo che tutti possano comprendere come il sistema funzionerà e come sarà sviluppato. Questo documento è un riferimento essenziale per il team di sviluppo durante tutto il processo, in quanto definisce una base comune da cui attingere nelle successive fasi di implementazione e testing.

ARCHITECTURAL DESIGN

PATTERN MVC

Questo progetto sarà implementato come struttura modulare, l'architettura utilizzata e quella MVC. Per questo progetto è stato scelto il design orientato agli oggetti, quest'ultimo favorisce eventuali modifiche future, garantendo la manutenibilità del software.



• **Model**: incapsula i dati e fornisce i metodi che permettono di interagire con essi e manipolarli. Non conosce nulla dell'interfaccia utente.
• **View:** responsabile della presentazione dei dati all'utente e gestisce i componenti dell'interfaccia.
• **Controller**: agisce come un intermediario tra il modello e la vista:– Comunica con il modello per richiedere i dati o effettuare modifiche in base alle azioni richieste dall'utente.– Aggiorna la vista per riflettere sull'interfaccia tutti i cambiamenti nel modello avvenuti a seguito di un'azione dell'utente.

1. Architettura del sistema

L'applicazione "Biblioteca Universitaria" è organizzata secondo un'architettura a livelli di tipo MVC (Model–View–Controller), con un package aggiuntivo dedicato alla persistenza dei dati. L'obiettivo è separare in modo netto la rappresentazione e gestione dei dati (Model), l'interfaccia grafica verso l'utente (View), la logica di coordinamento tra interfaccia e dati (Controller) e la gestione dei file e dell'archivio persistente (Persistence).

1.1 Package principali

Di seguito si descrivono i quattro package principali, le loro responsabilità e il loro ruolo all'interno del sistema.

1.1.1 biblioteca.model

Il package model rappresenta il cuore logico dell'applicazione, cioè il modello dei dati e le regole di business.

Al suo interno sono presenti le classi di dominio, che modellano i concetti principali del problema (Libro, Utente, Prestito ...), e una classe di facciata Biblioteca, che gestisce in modo centralizzato l'insieme di libri, utenti e prestiti. In Biblioteca sono implementate le operazioni di alto livello: inserimento, modifica ed eliminazione di libri e utenti, registrazione e restituzione dei prestiti, inserimento e controllo della blacklist e applicazione dei vincoli sui prestiti e sulla disponibilità delle copie.

Questo package non conosce l'interfaccia grafica né le modalità di salvataggio su file: espone soltanto classi e metodi che rappresentano il dominio della biblioteca e le relative regole.

1.1.2 biblioteca.view

Il package view contiene tutte le classi dedicate all'interfaccia grafica (GUI).

Le sue responsabilità principali sono: mostrare i dati all'utente tramite finestre, pannelli, tabelle e campi di input; raccogliere l'input dell'utente (click su pulsanti, testo inserito, selezioni nelle tabelle); inoltrare le azioni dell'utente ai rispettivi controller.

Le classi della view non implementano la logica di business: non verificano vincoli sui prestiti, non aggiornano direttamente le strutture dati del modello e non accedono ai file. Tutte queste responsabilità sono delegate ai package controller e model.

1.1.3 biblioteca.controller

Il package controller contiene le classi che fungono da ponte tra interfaccia grafica e modello.

Ogni controller è responsabile di una specifica area funzionale (ad esempio LibriController, UtentiController, PrestitiController, AuthController). I controller ricevono le richieste dalle view, leggono i dati inseriti dall'utente nei componenti grafici, invocano i metodi opportuni del modello (in particolare quelli della classe Biblioteca) e, quando necessario, richiedono al package persistence di salvare su file le modifiche effettuate. Infine, aggiornano le view con i risultati delle operazioni.

1.1.4 biblioteca.persistence

Il package persistence è dedicato alla gestione della persistenza dei dati, cioè al salvataggio e al caricamento dell'archivio su file.

Al suo interno si trovano classi come, ad esempio, ArchivioFile, incaricate di caricare all'avvio lo stato della biblioteca e di salvare su file le modifiche apportate durante l'utilizzo dell'applicazione. Questo package conosce le classi del modello, perché deve poter leggere e scrivere oggetti di tipo Biblioteca, Libro, Utente, Prestito, ecc. Il resto dell'applicazione non si occupa del formato fisico dei file, ma usa esclusivamente i metodi esposti da questo package.

1.1.5 biblioteca.main

Il package biblioteca.main raccoglie le classi che rappresentano il punto di avvio dell'applicazione e svolge il ruolo di semplice interruttore di avvio dell'applicazione: riceve il controllo dalla JVM e lo passa subito al livello di controllo che gestisce il normale funzionamento del sistema "Biblioteca".

1.2 Dipendenze tra i moduli

Le dipendenze tra i package seguono la stratificazione MVC con persistenza e main.

Alla base c'è *biblioteca.model*, che contiene solo dati e regole di business.

Sopra di lui troviamo biblioteca.persistence, che dipende dal *model* perché deve leggere e scrivere oggetti Biblioteca, Libro, Utente, Prestito su file.

Il package biblioteca.controller dipende sia dal *model* sia dalla *persistence*: usa Biblioteca per applicare le regole di dominio e, quando serve, chiama ArchivioFile per caricare/salvare i dati. Ma anche dalla *view* per leggere l'input e aggiornare le schermate.

Il package biblioteca.view dipende dai *controller*: le finestre e i pannelli invocano i metodi dei controller in risposta agli eventi dell'utente, senza accedere direttamente né al model né ai file.

Infine biblioteca.main dipende dal *Controller*: il metodo main istanzia il controller principale e gli passa il controllo, così da inizializzare modello, persistenza e interfaccia grafica e avviare l'applicazione.

Figura 1 – Riepilogo dei package principali:

Package	Responsabilità principale	Dipende da
biblioteca.model	Dati e logica di business (entità di dominio + classe Biblioteca).	—
biblioteca.view	Interfaccia grafica (finestre, pannelli, tabelle, form).	controller
biblioteca.controller	Ponte tra view e model; coordina operazioni e salvataggi.	model, view, persistence
biblioteca.persistence	Gestione della persistenza su file dell'archivio della biblioteca.	model
biblioteca.main	—-	controller

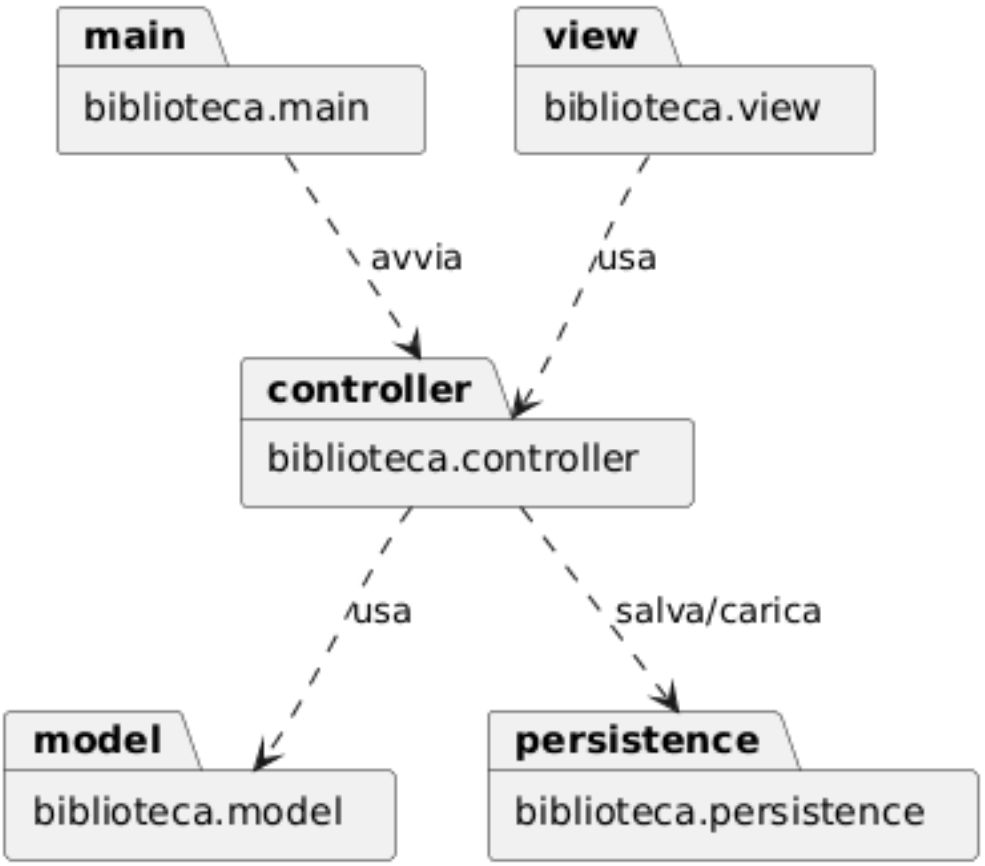
1.2 Pattern architetturali utilizzati

L'architettura del sistema adotta in particolare il pattern MVC (Model–View–Controller). Il Model corrisponde al package model, che contiene le entità di dominio e la classe Biblioteca con le regole di gestione; la View corrisponde al package view, che comprende tutte le finestre e i pannelli dell'interfaccia grafica; il Controller corrisponde al package controller, che riceve gli input dalla view, invoca il modello e aggiorna la view con i risultati.

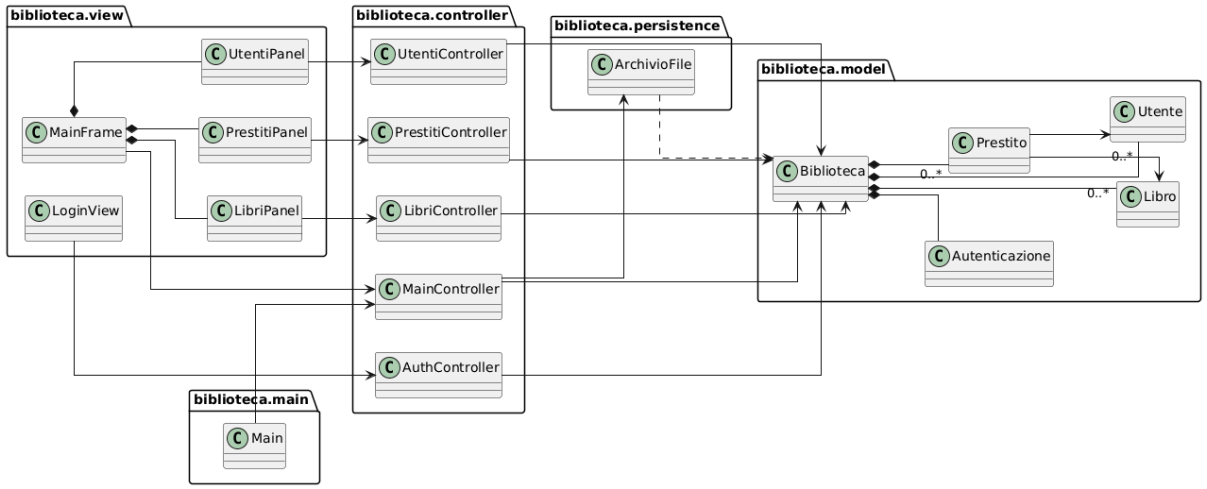
1.3 Diagramma dei Package

Figura 2 :

Diagramma dei package - Biblioteca Universitaria

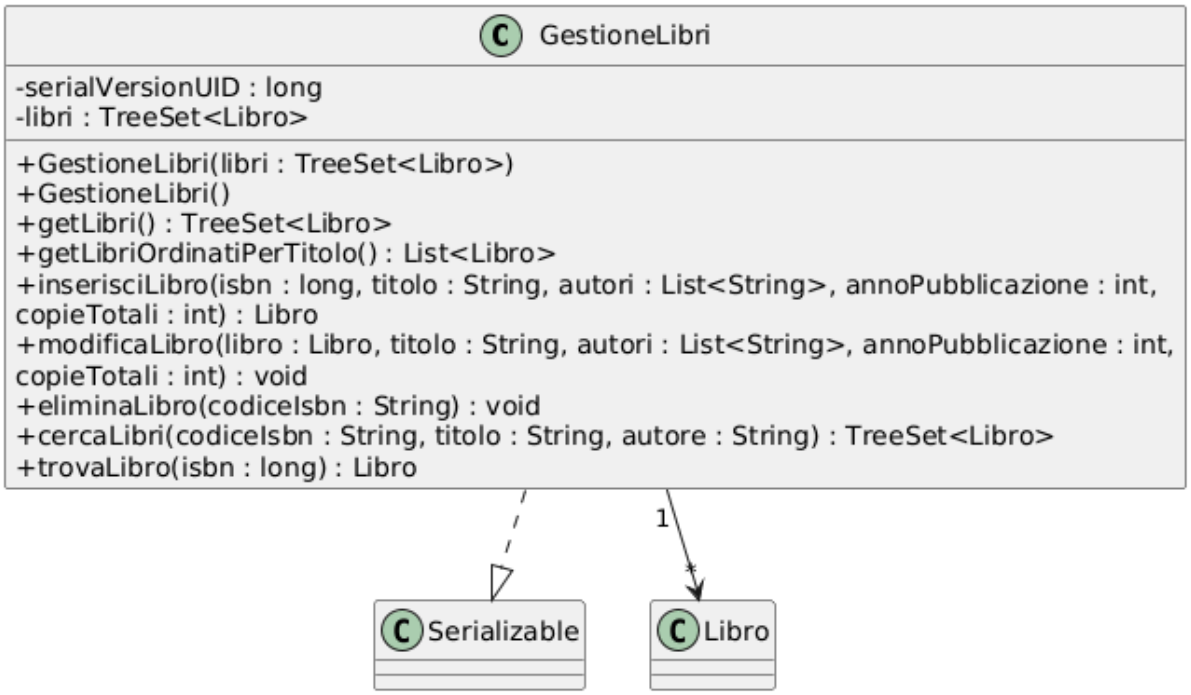


2. Modello statico



2.1 MODEL

➔ 2.1.1 GestioneLibri CLASS



Attributi:

-serialVersionUID: long:
attributo statico che rappresenta l'identificativo della classe serializzabile.

-libri : TreeSet<Libro>:
collezione che contiene i libri ordinati.

Metodi:

+GestioneLibri(libri:TreeSet<Libro>):
costruttore che permette di istanziare un oggetto di tipo GestioneLibri.

+GestioneLibri():
costruttore che permette di istanziare un oggetto di tipo GestioneLibri.

+getLibri():TreeSet<Libro>:
ritorna il treeset di libri ordinati per titolo.

+getLibriOrdinatiPerTitolo():List<Libro>:
ritorna la lista di libri ordinata per titolo.

+inserisciLibro(isbn:long, titolo: String, autori : List<String>, annoPubblicazione: int, copieTotali: int): Libro :

permette di inserire un nuovo libro specificando parametri come isbn,titolo,autori,annoPubblicazione,copieTotali.

+modificaLibro(libro:Libro, titolo: String, autori:List<String>, annoPubblicazione: int, copieTotali:int):void:

permette di modificare un libro accedendo a campi come titolo,autori,annoPubblicazione e copie totali.

+eliminaLibro(codiceIsbn: String):void:
permette di eliminare un libro dal sistema.

+cercaLibri(codiceIsbn: String, titolo:String, autore:String): TreeSet<Libro>:

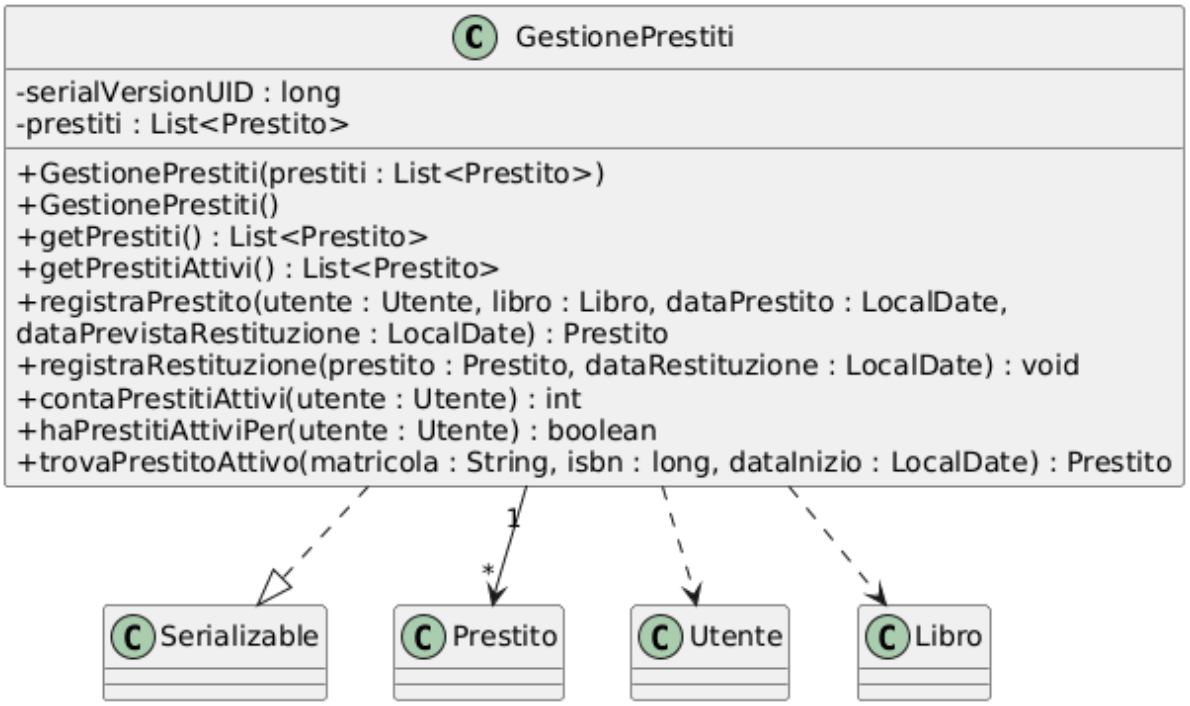
permette di cercare un libro nel sistema compilando i campi contenenti Isbn, titolo, autore.

+trovaLibro(isbn:long):Libro:
permette di trovare un libro specifico tramite codice isbn.

Relazioni:

La classe **GestioneLibri** implementa l'interfaccia **Serializable**, così l'intero insieme dei libri può essere salvato e ricaricato da file e ha una relazione di composizione con **Libro**: contiene internamente un **TreeSet<Libro>** e si occupa della loro gestione (inserimento, modifica, cancellazione, ricerca)

➔ 2.1.2 GestionePrestiti CLASS



Attributi:

-serialVersionUID : long :
attributo statico che rappresenta l'identificativo della classe serializzabile.

-prestiti : List<Prestito>:
collezione che contiene la lista di prestiti.

Metodi:

+GestionePrestiti(prestiti:List<Prestiti>):
costruttore che permette di istanziare un oggetto Gestione Prestiti.

+GestionePrestiti():
costruttore che permette di istanziare un oggetto Gestione Prestiti.

+getPrestiti():List<Prestito>:
ritorna la lista di prestiti.

+getPrestitiAttivi():List<Prestito>:

ritorna la lista di prestiti in corso.

+registraPrestito(utente: Utente, libro: Libro, dataPrestito:LocalDate, dataPrevistaRestituzione:LocalDate):Prestito:

permette di registrare un nuovo prestito.

+registraRestituzione(prestito:Prestito,dataRestituzione:LocalDate):void:

permette di registrare un'avvenuta restituzione.

+contaPrestitiAttivi(utente:Utente):int :

restituisce quanti prestiti in corso ha un utente.

+haPrestitiAttiviPer(utente:Utente):boolean:

restituisce true se l'utente ha prestiti attivi altrimenti false.

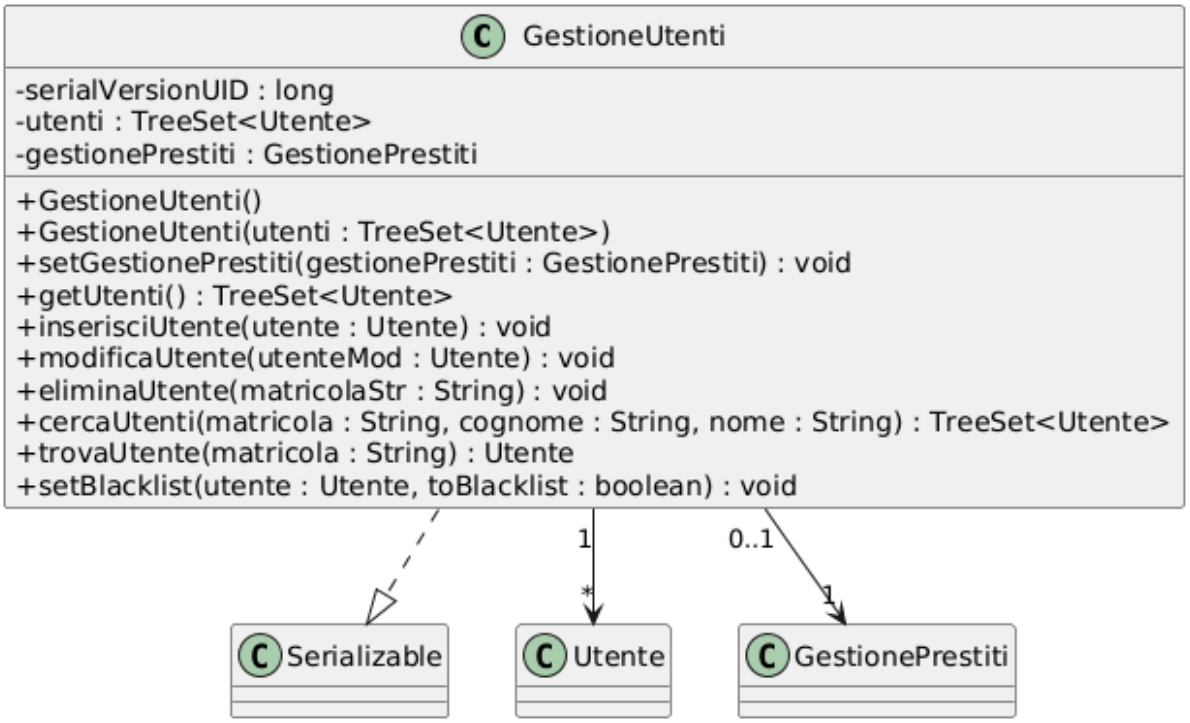
+trovaPrestitoAttivo(matricola:String,isbn:long,dataInizio:LocalDate):Prestito:

trova un prestito attivo specificando matricola,isbn, e data di inizio.

Relazioni:

La classe **GestionePrestiti** implementa **Serializable**, quindi i suoi oggetti possono essere salvati su file e ricaricati, mantenendo lo stato dei prestiti nel tempo. Al suo interno gestisce una lista di **Prestito**, quindi esiste una relazione diretta e stabile tra **GestionePrestiti** e **Prestito**. Le classi **Utente** e **Libro** non sono memorizzate come attributi, ma vengono usate nei metodi quando si registra, si cerca o si chiude un prestito, quindi con esse c'è solo una relazione di dipendenza.

➔ 2.1.3 GestioneUtenti CLASS



Attributi

-serialVersionUID:long:
attributo statico che rappresenta l'identificativo della classe serializzabile.

-utenti: TreeSet<Utente>:
contiene la collezione di utenti ordinata.

-gestionePrestiti:GestionePrestiti:
porta con sé le informazioni sui prestiti, utili per gestire gli utenti.

Metodi

+GestioneUtenti():
costruttore che permette di istanziare un ooggetto GestioneUtenti.

+GestioneUtenti(utenti:TreeSet<Utente>):

costruttore che permette di istanziare un oggetto GestioneUtenti che porta con se un treeSet di utenti.

+setGestionePrestiti(gestionePrestiti: GestionePrestiti):void:

assegna un prestito ad un utente.

+getUtenti():TreeSet<Utente>:

ritorna il set di utenti ordinato.

+InserisciUtente(utente:Utente):void:

permette di inserire e registrare un nuovo utente nel sistema.

+modificaUtente(utenteMod:Utente):void:

permette di modificare un utente presente nel sistema.

+eliminaUtente(matricolaStr:String):void:

permette di eliminare un utente del sistema.

+cercaUtenti(matricola: String, cognome : String, nome:String): TreeSet<Utente>:

permette di ricercare utenti nel sistema.

+trovaUtente(matricola:String):Utente:

permette di trovare un utente tramite la matricola.

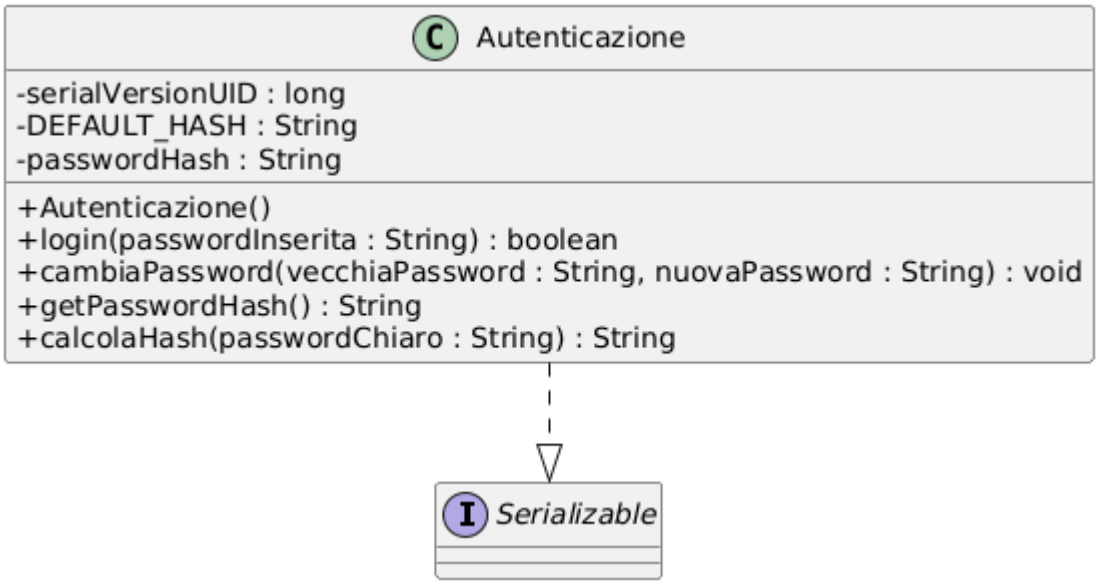
+setBlackList(utente: Utente, toBlacklist: boolean):void:

permette di aggiunere/rimuovere un utente alla blacklist.

Relazioni:

La classe **GestioneUtenti** implementa **Serializable**, quindi i suoi oggetti possono essere salvati e ripristinati insieme all'insieme ordinato di **Utente** che gestisce tramite un TreeSet, creando una relazione stabile uno-a-molti con la classe Utente. Inoltre mantiene un riferimento (transient) a **GestionePrestiti**, che utilizza per controllare l'eventuale presenza di prestiti attivi prima di eliminare un utente, instaurando così un'associazione diretta tra i due gestori.

➔ 2.1.4 Autenticazione CLASS



Attributi

-serialVersionUID:long:

attributo statico che rappresenta l'identificativo della classe serializzabile.

-passwordHash:String:

contiene la password codificata.

Metodi:

+Autenticazione():

costruttore che permette di istanziare un oggetto di tipo Autenticazione.

+login(passwordInserita:String):boolean:

ritorna il valore che deriva dal confronto tra la password codificata e la password inserita al momento del login, serve ad autenticare il bibliotecario ad accedere al sistema.

+cambiaPassword(vecchiaPassword:String, nuovaPassword:String):void:

permette di cambiare la password di accesso.

+getPasswordHash():String:

ritorna la stringa contenente la password codificata.

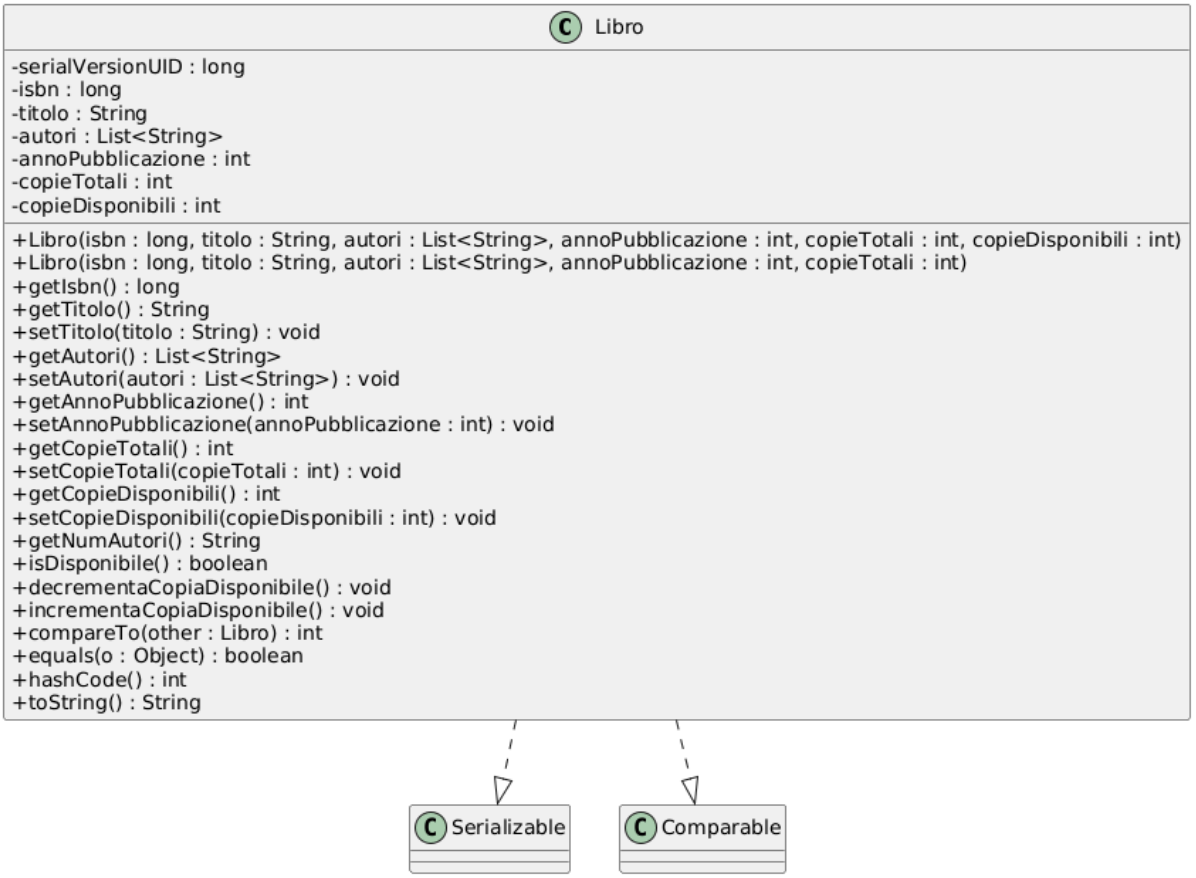
+calcolaHash(passwordChiaro:String):String:

metodo statico che permette di calcolare l'hash della password.

Relazioni:

La classe Autenticazione usa l'interfaccia Serializable per permettere il salvataggio della password su file. Questo consente al programma di ricordare la password anche dopo la chiusura dell'applicazione.

➔ 2.1.5 Libro CLASS



Attributi

-serialVersionUID: long :
Identificatore di versione della classe per la serializzazione, usato da Java per verificare la compatibilità degli oggetti Utente salvati su file.

+MAX_PRESTITI: int :
Numero massimo di prestiti contemporanei consentiti per ogni utente.

-matricola: String :
Codice identificativo univoco dell'utente all'interno del sistema (ad esempio numero di matricola).

-nome: String :
Nome proprio dell'utente.

-cognome: String :
Cognome dell'utente.

-email: String :
Indirizzo email associato all'utente, usato per contatti o notifiche.

-inBlacklist: boolean :
Indica se l'utente è in blacklist (true) e quindi non può effettuare nuovi prestiti, oppure no (false).

-prestitiAttivi: List<Prestito> :
Elenco dei prestiti ancora attivi associati all'utente (libri non ancora restituiti).

Metodi:

+Utente(matricola: String, nome: String, cognome: String, email: String) :
Costruttore che crea un nuovo utente impostando matricola, nome, cognome ed email.

+getMatricola() : String :
Restituisce la matricola dell'utente.

+getNome() : String :
Restituisce il nome dell'utente.

+setNome(nome: String) : void :
Modifica il nome dell'utente.

+getCognome() : String :
Restituisce il cognome dell'utente.

+setCognome(cognome: String) : void :
Modifica il cognome dell'utente.

+getEmail() : String :
Restituisce l'indirizzo email dell'utente.

+setEmail(email: String) : void :
Modifica l'indirizzo email associato all'utente.

+isInBlacklist() : boolean :
Indica se l'utente risulta attualmente inserito in blacklist.

+setInBlacklist(valore: boolean) : void :
Imposta o rimuove l'utente dalla blacklist in base al valore passato.

+getNumPrestitiAttivi() : int :
Restituisce il numero di prestiti attivi associati all'utente.

+getPrestitiAttivi() : List<Prestito> :
Restituisce la lista dei prestiti attivi dell'utente.

+canNuovoPrestito() : boolean :
Verifica se l'utente può effettuare un nuovo prestito (non in blacklist e con numero di prestiti attivi inferiore a MAX_PRESTITI).

+aggiungiPrestito(prestito: Prestito) : void :
Aggiunge un prestito alla lista dei prestiti attivi dell'utente.

+rimuoviPrestito(prestito: Prestito) : void :
Rimuove un prestito dalla lista dei prestiti attivi (ad esempio dopo la restituzione del libro).

+compareTo(o: Utente) : int :
Confronta questo utente con un altro per l'ordinamento, in base al criterio definito (ad esempio matricola o cognome/nome).

+equals(o: Object) : boolean :
Verifica se l'oggetto passato rappresenta lo stesso utente (stessa identità logica).

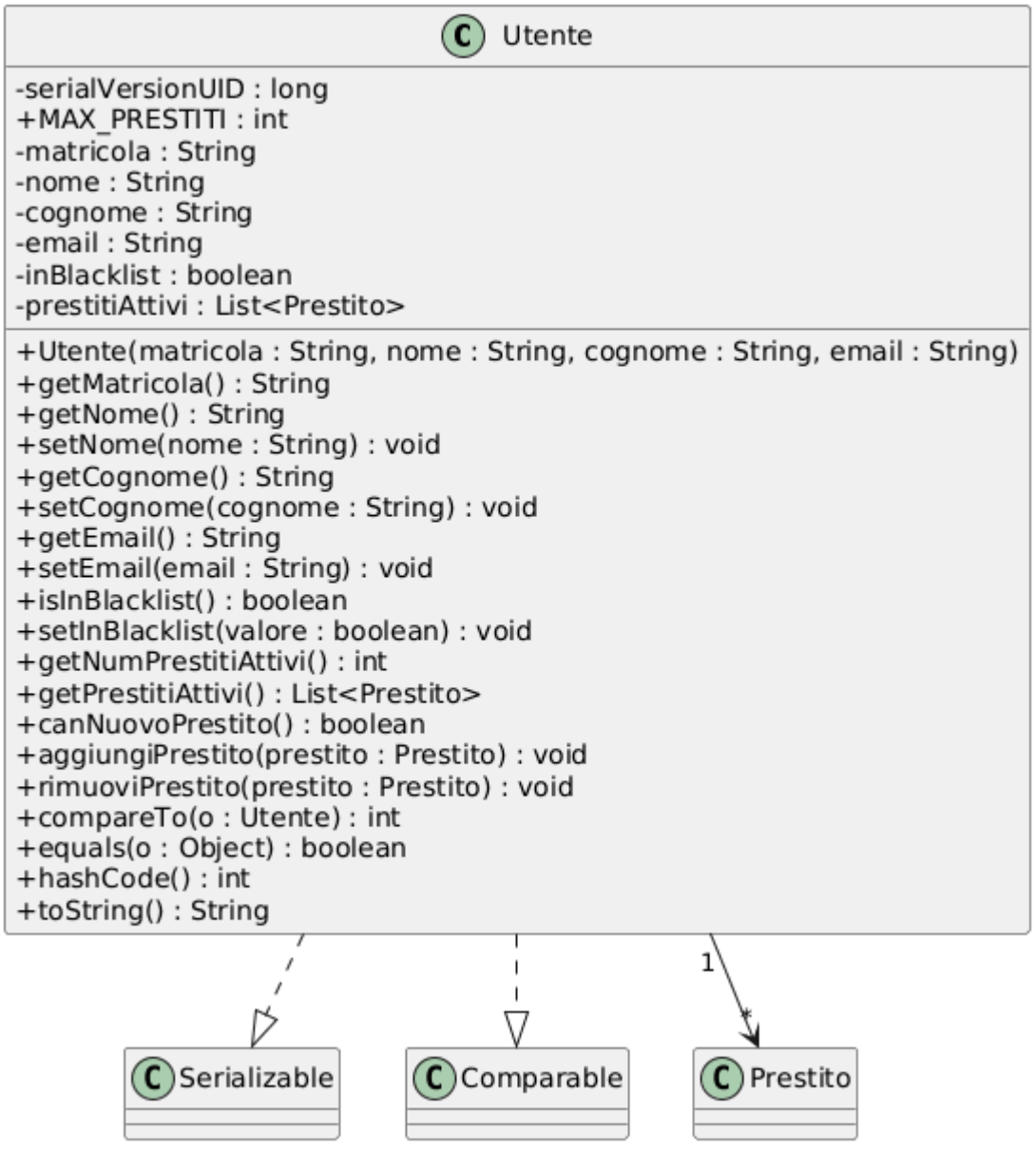
+hashCode() : int :
Restituisce il codice hash dell'utente, coerente con il metodo equals.

+toString() : String :
Restituisce una rappresentazione testuale dell'utente (ad esempio combinazione di matricola, nome e cognome).

Relazioni:

La classe **Libro** implementa **Serializable**, quindi può essere salvata e ricaricata da file insieme al resto dell'archivio della biblioteca, e implementa anche **Comparable**, così i libri possono essere ordinati in base al loro ISBN

➔ 2.1.6 Utente CLASS



Attributi

-serialVersionUID: long :
Identificatore di versione della classe per la serializzazione, usato da Java per verificare la compatibilità degli oggetti Utente salvati su file.

+MAX_PRESTITI: int :
Numero massimo di prestiti contemporanei consentiti per ogni utente.

-matricola: String :
Codice identificativo univoco dello studente/utente della biblioteca.

-nome: String :
Nome dell'utente.

-cognome: String :
Cognome dell'utente.

-email: String :
Indirizzo email associato all'utente, usato per contatti o notifiche.

-inBlacklist: boolean :
Indica se l'utente è inserito nella blacklist (ad esempio per ritardi o violazioni del regolamento).

-prestitiAttivi: List<Prestito> :
Elenco dei prestiti attualmente attivi associati a questo utente.

Metodi:

+Utente(matricola: String, nome: String, cognome: String, email: String) :
Costruttore che crea un nuovo utente impostando matricola, nome, cognome ed email.

+getMatricola() : String :
Restituisce la matricola dell'utente.

+getNome() : String :
Restituisce il nome dell'utente.

+setNome(nome: String) : void :
Aggiorna il nome dell'utente.

+getCognome() : String :

Restituisce il cognome dell'utente.

+setCognome(cognome: String) : void :
Aggiorna il cognome dell'utente.

+getEmail() : String :
Restituisce l'indirizzo email dell'utente.

+setEmail(email: String) : void :
Aggiorna l'indirizzo email dell'utente.

+isInBlacklist() : boolean :
Indica se l'utente risulta attualmente in blacklist.

+setInBlacklist(valore: boolean) : void :
Imposta o rimuove l'utente dalla blacklist in base al valore passato.

+getNumPrestitiAttivi() : int :
Restituisce il numero di prestiti ancora attivi per questo utente.

+getPrestitiAttivi() : List<Prestito> :
Restituisce la lista dei prestiti attivi associati all'utente.

+canNuovoPrestito() : boolean :
Verifica se l'utente può effettuare un nuovo prestito, controllando che non superi MAX_PRESTITI e che non sia in blacklist.

+aggiungiPrestito(prestito: Prestito) : void :
Aggiunge un nuovo prestito all'elenco dei prestiti attivi dell'utente.

+rimuoviPrestito(prestito: Prestito) : void :
Rimuove il prestito indicato dalla lista dei prestiti attivi (tipicamente dopo la restituzione del libro).

+compareTo(o: Utente) : int :
Confronta questo utente con un altro per definire un ordine (ad esempio in base alla matricola o al cognome).

+equals(o: Object) : boolean :
Determina se questo utente è uguale all'oggetto passato, in base ai campi identificativi (ad esempio la matricola).

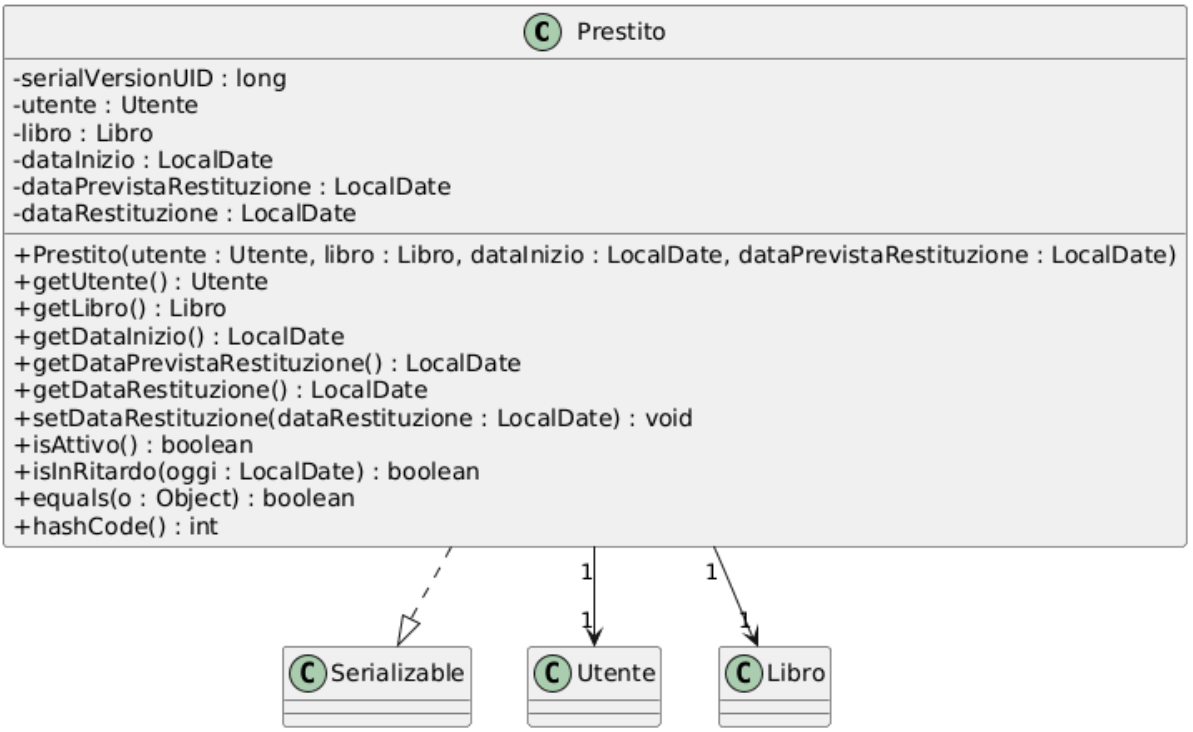
+hashCode() : int :
Restituisce il codice hash associato a questo utente, coerente con il criterio adottato in equals.

+toString() : String :
Restituisce una rappresentazione testuale dell'utente, utile per stampe e debug.

Relazioni:

La classe Utente implementa **Serializable**, quindi può essere salvata e ripristinata insieme ai dati della biblioteca, e implementa anche **Comparable**, così gli utenti possono essere ordinati per matricola. Ogni utente mantiene al suo interno la lista dei **Prestito** attivi che lo riguardano, creando una relazione uno-a-molti verso la classe Prestito.

➔ 2.1.7 Prestito CLASS



Attributi

-serialVersionUID: long :
Identificatore di versione della classe per la serializzazione, usato da Java per controllare la compatibilità degli oggetti Prestito salvati su file.

-utente: Utente :
Riferimento all'utente che ha effettuato il prestito del libro.

-libro: Libro :
Riferimento al libro che è stato prestato all'utente.

-dataInizio: LocalDate :
Data in cui il prestito è stato registrato e il libro è stato consegnato all'utente.

-dataPrevistaRestituzione: LocalDate :
Data entro la quale l'utente dovrebbe restituire il libro, secondo le regole della biblioteca.

-dataRestituzione: LocalDate :
Data in cui il libro è stato effettivamente restituito; può essere nulla se il prestito è ancora attivo.

Metodi:

+Prestito(utente: Utente, libro: Libro, dataInizio: LocalDate, dataPrevistaRestituzione: LocalDate) :
Costruttore che crea un nuovo prestito, collegando utente e libro e impostando data di inizio e data prevista di restituzione.

+getUtente() : Utente :
Restituisce l'utente a cui è associato il prestito.

+getLibro() : Libro :
Restituisce il libro che è stato prestato.

+getDataInizio() : LocalDate :
Restituisce la data di inizio del prestito.

+getDataPrevistaRestituzione() : LocalDate :
Restituisce la data entro cui il libro dovrebbe essere restituito.

+getDataRestituzione() : LocalDate :
Restituisce la data in cui il libro è stato effettivamente restituito, se presente.

+setDataRestituzione(dataRestituzione: LocalDate) : void :
Imposta la data di restituzione del prestito quando il libro viene riportato in biblioteca.

+isAttivo() : boolean :
Indica se il prestito è ancora attivo, cioè se il libro non è stato ancora restituito.

+isInRitardo(oggi: LocalDate) : boolean :
Verifica se, alla data passata come parametro, il prestito risulta in ritardo rispetto alla data prevista di restituzione.

+equals(o: Object) : boolean :
Confronta questo prestito con un altro oggetto per stabilire se rappresentano lo stesso prestito (stesso utente, libro e date).

+hashCode() : int :
Restituisce il codice hash associato a questo prestito, coerente con il criterio usato nel metodo equals.

-libro: Libro {final}:
Indica il libro oggetto del prestito, ovvero il volume che è stato consegnato all'utente.

-dataInizio: LocalDate {final}:
Memorizza la data di inizio del prestito, cioè il giorno in cui il bibliotecario registra l'uscita del libro.

-dataPrevistaRestituzione: LocalDate {final}:
Indica la data entro la quale l'utente dovrebbe restituire il libro, calcolata in base alla durata massima del prestito.

Metodi:

+Prestito(Utente utente, Libro libro, LocalDate dataInizio, LocalDate +dataPrevistaRestituzione):
Costruttore della classe. Inizializza un nuovo prestito associando l'utente, il libro, la data di inizio e la data prevista di restituzione. Imposta inizialmente dataRestituzione a null.

+getUtente(): Utente :

Restituisce l'utente a cui è intestato il prestito.

+getLibro(): Libro :
Restituisce il libro oggetto del prestito.

+getDataInizio(): LocalData :
Restituisce la data di inizio del prestito.

+getDataPrevistaRestituzione(): LocalData :
Restituisce la data prevista di restituzione del libro.

+isAttivo(): boolean :
Restituisce true se il prestito è ancora attivo (cioè dataRestituzione è null), false altrimenti.

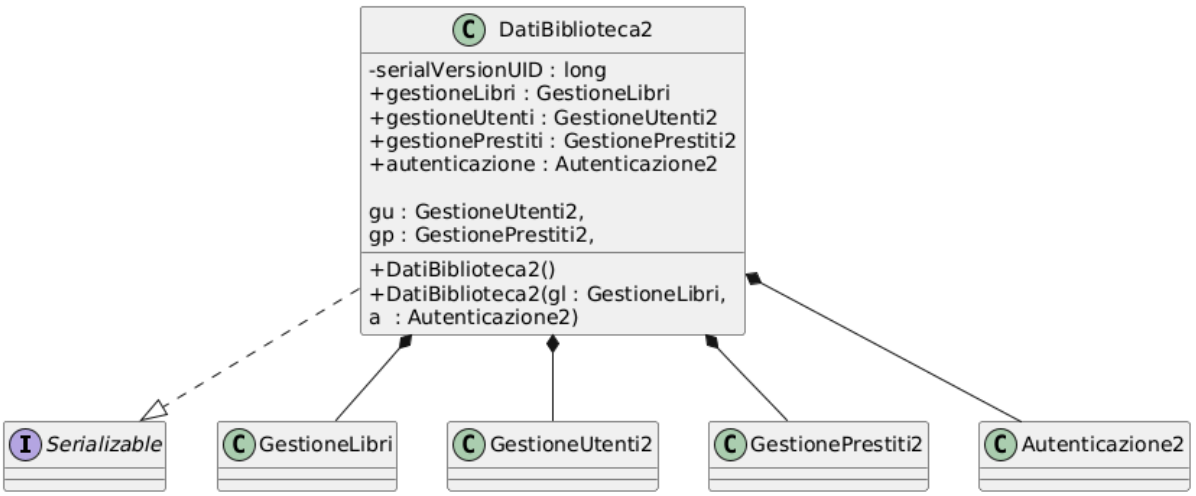
+isInRitardo(LocalDate oggi): boolean :
Verifica se il prestito è in ritardo rispetto alla data prevista. Restituisce true se il prestito è ancora attivo e la data corrente (o quella passata come parametro) è successiva a dataPrevistaRestituzione.

+toString(): String :
Metodo Override che ci permette di modificare l'output.

Relazioni:

La classe **Prestito** implementa **Serializable**, quindi i suoi oggetti possono essere salvati insieme allo stato della biblioteca. Ogni prestito collega in modo diretto un singolo **Utente** e un singolo **Libro**, con una relazione uno a uno verso entrambe le classi, e memorizza le varie date che permettono di sapere se il prestito è ancora attivo o se è in ritardo rispetto alla data prevista di restituzione.

➔ 2.1.8 DatiBiblioteca CLASS



Attributi:

-serialVersionUID: long :
Identificatore di versione della classe per la serializzazione, usato da Java per verificare la compatibilità degli oggetti salvati su file.

+gestioneLibri: GestioneLibri :
Riferimento al gestore dei libri, che contiene l'insieme dei volumi e le operazioni di gestione ad essi relative.

+gestioneUtenti: GestioneUtenti :
Riferimento al gestore degli utenti, responsabile della memorizzazione e manipolazione dell'elenco degli utenti.

+gestionePrestiti: GestionePrestiti :
Riferimento al gestore dei prestiti, che mantiene la lista dei prestiti e le relative operazioni (registrazione, restituzione, ecc.).

+autenticazione: Autenticazione :
Riferimento all'oggetto che gestisce le informazioni di autenticazione del bibliotecario (password, hash, ecc.).

Metodi:

+DatiBiblioteca() :
Costruttore di default che crea un contenitore vuoto per i dati della biblioteca.

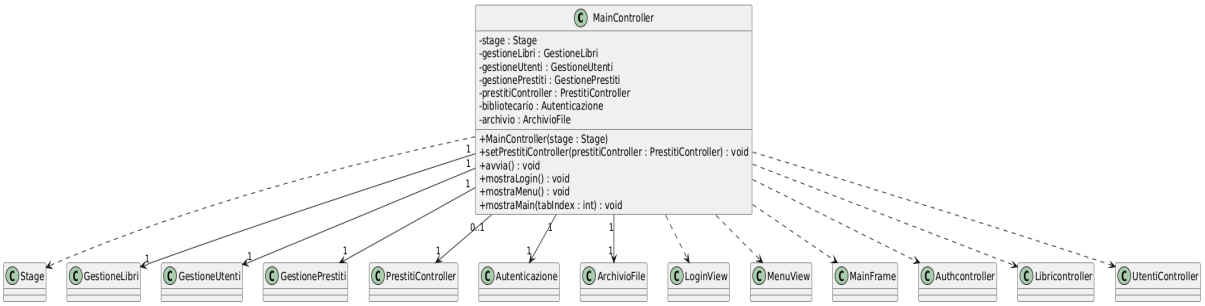
+DatiBiblioteca(gl : GestioneLibri, gu : GestioneUtenti, gp : GestionePrestiti, a : Autenticazione) :
Costruttore che inizializza l'oggetto impostando direttamente i riferimenti a gestore libri, gestore utenti, gestore prestiti e autenticazione.

Relazioni:

La classe **DatiBiblioteca** implementa **Serializable** e funge da contenitore unico per i moduli principali del sistema: **GestioneLibri**, **GestioneUtenti**, **GestionePrestiti** e **Autenticazione**.

2.2 CONTROLLER

➔ 2.2.1 MainController CLASS



Attributi:

-stage: Stage :
Finestra principale JavaFX dell'applicazione, sulla quale vengono mostrate di volta in volta login, menu e finestra principale con le tab.

-gestioneLibri: GestioneLibri :
Gestore del modello dei libri, contiene la collezione dei volumi e le operazioni di gestione (inserimento, modifica, eliminazione, ricerca).

-gestioneUtenti: GestioneUtenti :
Gestore del modello degli utenti, usato per mantenere e manipolare l'elenco degli utenti registrati.

-gestionePrestiti: GestionePrestiti :
Gestore del modello dei prestiti, mantiene la lista dei prestiti e le relative operazioni di registrazione e restituzione.

-prestitiController: PrestitiController :
Riferimento al controller della sezione prestiti, utile per collegare correttamente le tab e aggiornare la vista quando necessario.

-bibliotecario: Autenticazione :
Oggetto che gestisce le informazioni di autenticazione (password) del bibliotecario che utilizza il sistema.

-archivio: ArchivioFile :

Componente che si occupa di salvare e caricare su file lo stato di libri, utenti, prestiti e autenticazione.

Metodi:

+MainController(stage : Stage) :
Costruttore che inizializza il controller principale associandolo alla finestra JavaFX e creando i vari gestori del modello e la persistenza.

+setPrestitiController(prestitiController : PrestitiController): void:
Imposta il riferimento al controller dei prestiti, collegandolo al controller principale dopo la sua creazione.

+avvia() : void :
Punto di ingresso del flusso grafico: inizializza ciò che serve e mostra la prima schermata (tipicamente il login).

+mostraLogin() : void :
Configura la scena dello stage per visualizzare la schermata di login.

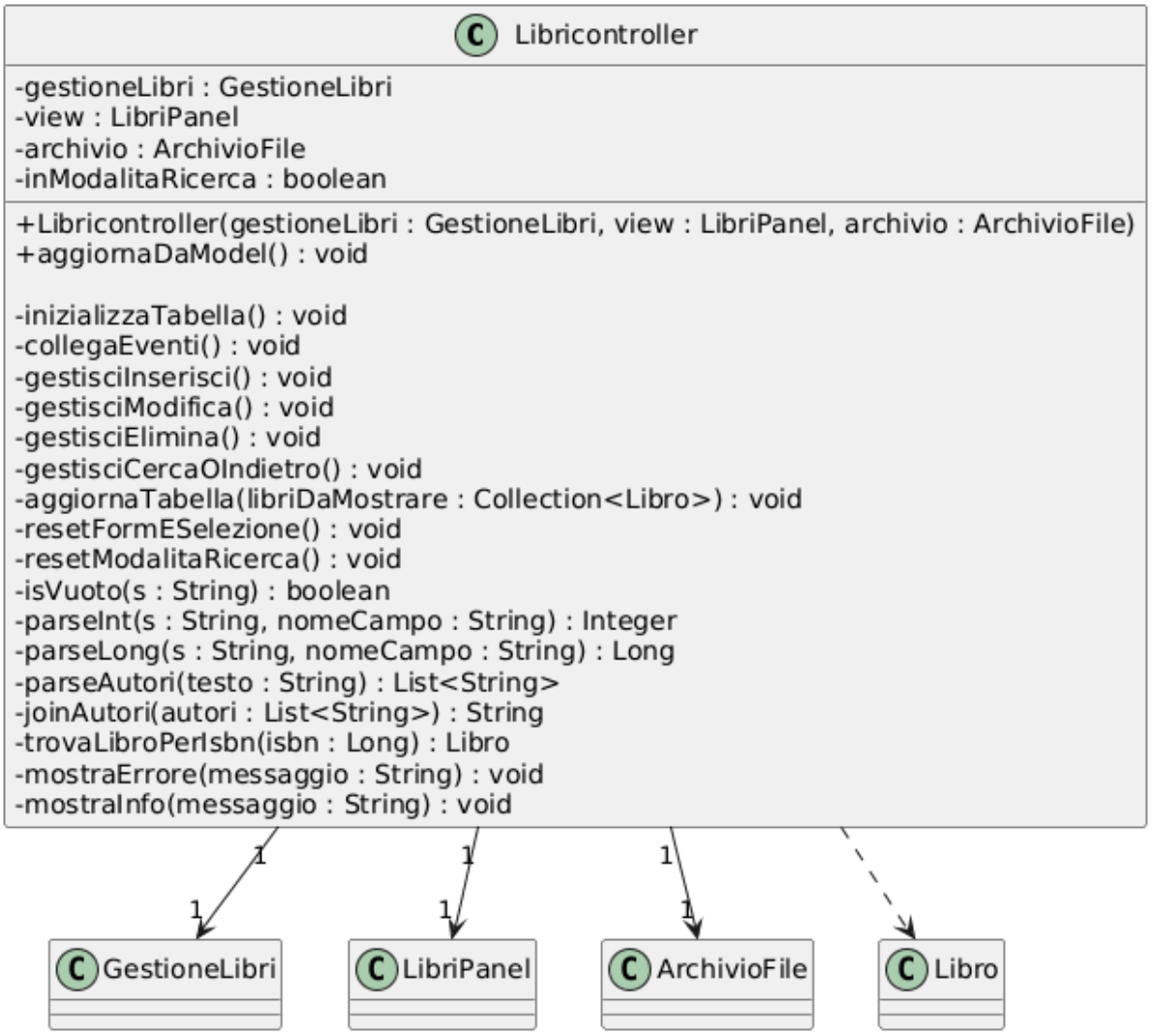
+mostraMenu() : void :
Sostituisce il contenuto della finestra con il menu principale, accessibile dopo un login corretto.

+mostraMain(tabIndex : int) : void :
Mostra la finestra principale con le tab (Libri, Utenti, Prestiti), aprendo inizialmente la tab indicata dall'indice passato come parametro.

Relazioni:

La classe **MainController** è il controller principale dell'applicazione: mantiene riferimenti stabili ai gestori del modello (**GestioneLibri, GestioneUtenti, GestionePrestiti**), al componente di autenticazione (**Autenticazione**) e all'oggetto di persistenza (**ArchivioFile**), con cui collabora per gestire lo stato della biblioteca. Ha inoltre un riferimento opzionale a **PrestitiController**, usato per coordinare la sezione dei prestiti. Attraverso i suoi metodi crea e utilizza le viste (**LoginView**, **MenuView**, **MainFrame**) e i controller specifici (**Authcontroller**, **Libricontroller**, **UtentiController**, **PrestitiController**), rispetto ai quali ha una relazione di semplice dipendenza, perché li istanzia e li usa per gestire il flusso tra le schermate.

➔ 2.2.2 LibriController CLASS



Attributi:

-gestioneLibri: GestioneLibri :

Gestore dei libri, contiene la collezione dei volumi e le operazioni di inserimento, modifica, eliminazione e ricerca.

-view: LibriPanel :

Pannello grafico dedicato alla gestione dei libri, con campi di input, tabella dei libri e pulsanti di azione.

-archivio: ArchivioFile :

Componente incaricato di salvare e caricare su file lo stato dei libri (e, in generale, dei dati dell'applicazione).

-inModalitaRicerca: boolean :

Indica se la tabella sta mostrando i risultati di una ricerca (true) oppure l'elenco completo dei libri (false).

Metodi:

+Libricontroller(gestioneLibri: GestioneLibri, view: LibriPanel, archivio: ArchivioFile) :

Costruttore che inizializza il controller dei libri collegando model, vista e sistema di persistenza.

-inizializzaTabella() : void :

Configura la tabella dei libri nella vista e la popola inizialmente con l'elenco dei libri presenti nel sistema.

-collegaEventi() : void :

Associa ai pulsanti del pannello libri le azioni corrispondenti (inserisci, modifica, elimina, cerca).

-gestisciInserisci() : void :

Legge i dati inseriti nei campi, li valida e, se corretti, crea un nuovo libro nel model aggiornando anche la tabella.

-gestisciModifica() : void :

Applica le modifiche ai dati del libro selezionato nella tabella, sincronizzando model e interfaccia grafica.

-gestisciElimina() : void :

Elimina dal model il libro selezionato (se consentito) e aggiorna la tabella dei libri.

-gestisciCercaOIndietro() : void :

Se non è in modalità ricerca, esegue una ricerca filtrando i libri in base ai campi compilati; se è già in ricerca, ripristina la visualizzazione completa.

-aggiornaTabella(libriDaMostrare: Collection<Libro>) : void :

Aggiorna il contenuto della tabella visualizzando l'insieme di libri passato come parametro.

-resetFormESelezione() : void :

Svuota i campi di input e deseleziona eventuali righe selezionate nella tabella dei libri.

-resetModalitaRicerca() : void :

Disattiva la modalità ricerca e riporta la tabella all'elenco completo dei libri.

-isVuoto(s: String) : boolean :

Verifica se la stringa passata è nulla, vuota o composta solo da spazi, per controllare la validità degli input.

-parseInt(s: String, nomeCampo: String) : Integer :

Prova a convertire la stringa in un numero intero (per esempio anno o copie totali), gestendo eventuali errori legati al campo indicato.

-parseLong(s: String, nomeCampo: String) : Long :

Converte la stringa in un valore long, tipicamente usato per il codice ISBN, controllando la correttezza del formato.

-parseAutori(testo: String) : List<String> :

Interpreta il testo inserito nel campo autori e lo suddivide in una lista di singoli autori.**+aggiornaDaModel() : void :**

-joinAutori(autori: List<String>) : String :

Ricompone la lista di autori in una singola stringa da mostrare nella tabella o nei campi di input.

-trovaLibroPerIsbn(isbn: Long) : Libro :

Cerca nel model il libro corrispondente all'ISBN indicato e lo restituisce, se presente.

-mostraErrore(messaggio: String) : void :

Mostra un messaggio di errore all'utente, ad esempio in seguito a un input non valido o a un'operazione non consentita.

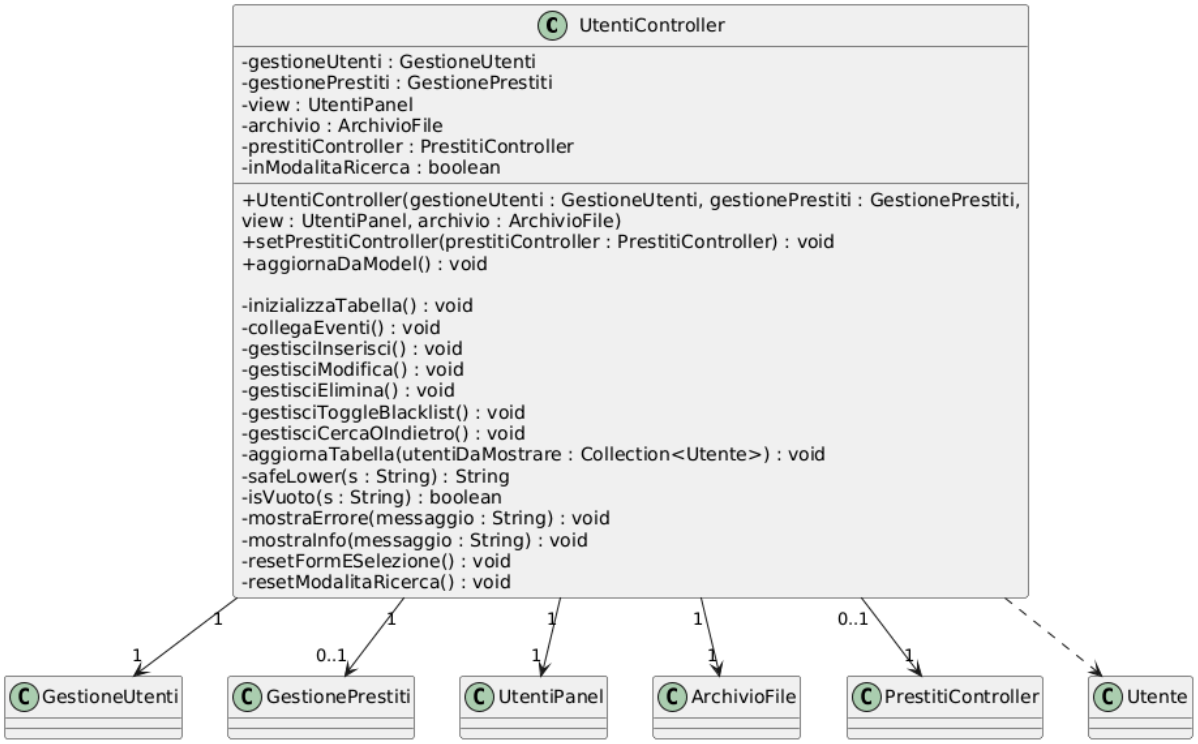
-mostraInfo(messaggio: String) : void :

Mostra un messaggio informativo che conferma il buon esito di un'operazione (inserimento, modifica, ecc.).

Relazioni:

La classe **Libricontroller** collega la parte grafica della sezione "Gestione Libri" (**LibriPanel**) con il modello **GestioneLibri** e con la persistenza gestita da **ArchivioFile**; ; usa inoltre la classe **Libro** come tipo di dato principale per rappresentare i singoli volumi mostrati e manipolati dall'utente.

➔ 2.2.3 UtentiController CLASS



Attributi:

-gestioneUtenti: GestioneUtenti :
Gestore degli utenti, contiene l'insieme degli utenti e le operazioni di inserimento, modifica, eliminazione e ricerca.

-gestionePrestiti: GestionePrestiti :
Oggetto che gestisce i prestiti, usato per controllare se un utente ha prestiti attivi (ad esempio prima di eliminarlo).

-view: UtentiPanel :
Pannello grafico dedicato alla gestione degli utenti, con campi di input, tabella e pulsanti.

-archivio: ArchivioFile :
Componente responsabile del salvataggio e caricamento su file dei dati relativi agli utenti (e, in generale, dell'intero sistema).

-prestitiController: PrestitiController :
Riferimento al controller dei prestiti, utilizzato per aggiornare la vista dei prestiti quando cambia lo stato degli utenti (es. blacklist).

-inModalitaRicerca: boolean :
Indica se la tabella sta mostrando il risultato di una ricerca (true) oppure l'elenco completo degli utenti (false).

Metodi:

+UtentiController(gestioneUtenti: GestioneUtenti, gestionePrestiti: GestionePrestiti, view: UtentiPanel, archivio: ArchivioFile) :
Costruttore che inizializza il controller collegando il model degli utenti, il model dei prestiti, la vista utenti e il sistema di persistenza su file.

+setPrestitiController(prestitiController: PrestitiController) :void :
Imposta il riferimento al controller dei prestiti, così da poter notificare aggiornamenti (ad esempio quando cambia la blacklist).

-inizializzaTabella() : void :
Configura la tabella degli utenti nella vista e la popola inizialmente con l'elenco degli utenti presenti nel sistema.

-collegaEventi() : void :
Associa ai pulsanti del pannello utenti le azioni corrispondenti (inserisci, modifica, elimina, blacklist, cerca).

-gestisciInserisci() : void :
Legge i dati inseriti nei campi, valida l'input e, se corretto, crea un nuovo utente nel model e aggiorna la tabella.

-gestisciModifica() : void :
Applica le modifiche ai dati dell'utente selezionato in tabella, aggiornando sia il model che la vista.

-gestisciElimina() : void :
Rimuove l'utente selezionato, dopo aver verificato che non abbia prestiti attivi, e aggiorna la tabella.

-gestisciToggleBlacklist() : void :
Aggiunge o rimuove l'utente selezionato dalla blacklist e notifica il controller dei prestiti per aggiornare i relativi dati.

-gestisciCercaOIndietro() : void :
Se non è in modalità ricerca, esegue una ricerca filtrando gli utenti in base ai campi compilati; se è già in ricerca, ripristina l'elenco completo.

-aggiornaTabella(utentiDaMostrare: Collection<Utente>) : void :
Aggiorna il contenuto della tabella mostrando l'insieme di utenti passato come parametro.

-safeLower(s: String) : String :
Restituisce la stringa in minuscolo, gestendo il caso in cui il valore sia nullo (utile per confronti case-insensitive).

-isVuoto(s: String) : boolean :
Verifica se una stringa è nulla, vuota o composta solo da spazi, per controllare la validità degli input.

-mostraErrore(messaggio: String) : void :
Mostra un messaggio di errore all'utente, tipicamente tramite una finestra di dialogo.

-mostraInfo(messaggio: String) : void :
Mostra un messaggio informativo (operazione avvenuta con successo, conferme, ecc.).

-resetFormESelezione() : void :
Svuota i campi di input e deseleziona eventuali righe selezionate nella tabella.

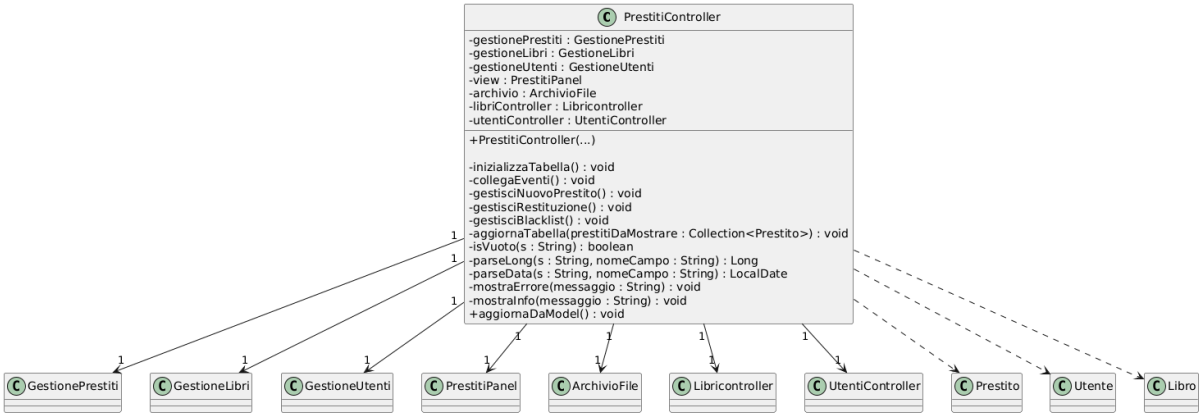
-resetModalitaRicerca() : void :
Disattiva la modalità ricerca e riporta la tabella alla visualizzazione dell'elenco completo.

+aggiornaDaModel() : void :
Ricarica l'elenco utenti dal model e aggiorna la tabella, per riflettere eventuali cambiamenti esterni.

Relazioni:

La classe **UtentiController** collega la sezione grafica "Gestione Utenti" (**UtentiPanel**) con i gestori del modello **GestioneUtenti** e **GestionePrestiti** e con la persistenza gestita da **ArchivioFile**, coordinando inserimento, modifica, eliminazione, ricerca e gestione della blacklist. Mantiene inoltre un riferimento opzionale a **PrestitiController** per aggiornare la tabella dei prestiti quando cambia lo stato di blacklist di un utente, e utilizza la classe **Utente** come tipo di dato principale per popolare e ordinare la tabella degli utenti.

➔ 2.2.4 PrestitiController CLASS



Attributi

-gestionePrestiti: GestionePrestiti :
Riferimento all'oggetto che gestisce la logica dei prestiti (registrazione, restituzione, elenco dei prestiti).

-gestioneLibri: GestioneLibri :
Gestore dei libri, usato per controllare disponibilità copie e aggiornare lo stato dei volumi quando vengono prestati o restituiti.

-gestioneUtenti: GestioneUtenti :
Gestore degli utenti, utilizzato per verificare matricole, stato di blacklist e numero di prestiti attivi.

-view: PrestitiPanel :
Pannello grafico dedicato alla gestione dei prestiti, contenente campi di input e tabella dei prestiti.

-archivio: ArchivioFile :
Oggetto incaricato di salvare e caricare su file lo stato di libri, utenti e prestiti.

-libriController: Libricontroller :
Controller della sezione libri, richiamato quando è necessario aggiornare la vista dei libri (ad esempio dopo un prestito o una restituzione).

-utentiController: UtentiController :
Controller della sezione utenti, usato per aggiornare le informazioni sugli utenti (es. blacklist) in seguito alle operazioni sui prestiti.

Metodi

+PrestitiController(gestionePrestiti: GestionePrestiti, view: PrestitiPanel, archivio: ArchivioFile, gestioneLibri: GestioneLibri, gestioneUtenti: GestioneUtenti, libriController: Libricontroller, utentiController: UtentiController) :
Costruttore che inizializza il controller dei prestiti collegando model, vista, archivio e gli altri controller coinvolti.

-inizializzaTabella() : void :
Configura la tabella dei prestiti nella vista, popolandola inizialmente con i prestiti attivi.

-collegaEventi() : void :
Associa ai pulsanti del pannello dei prestiti le azioni corrispondenti (nuovo prestito, restituzione, gestione blacklist).

-gestisciNuovoPrestito() : void :
Legge i dati inseriti nella vista, verifica matricola e disponibilità del libro e, se tutto è corretto, registra un nuovo prestito.

-gestisciRestituzione() : void :
Gestisce la restituzione del prestito selezionato, aggiornando data di restituzione, copie disponibili e tabelle collegate.

-gestisciBlacklist() : void :
Inserisce o rimuove l'utente legato al prestito selezionato dalla blacklist, aggiornando anche la vista degli utenti.

-aggiornaTabella(prestitiDaMostrare: Collection<Prestito>) : void :
Aggiorna il contenuto della tabella dei prestiti mostrando l'insieme di prestiti passato come parametro.

-isVuoto(s: String) : boolean :
Verifica se una stringa è nulla o vuota e quindi non valida come input.

-parseLong(s: String, nomeCampo: String) : Long :
Converte la stringa in un valore long (ad esempio per l'ISBN), gestendo eventuali errori di formattazione legati al campo indicato.

-parseData(s: String, nomeCampo: String) : LocalDate :
Converte la stringa in una data, controllando che il formato sia valido per il campo specificato (es. data prevista restituzione).

-mostraErrore(messaggio: String) : void :
Mostra nella GUI un messaggio di errore, tipicamente tramite una finestra di dialogo.

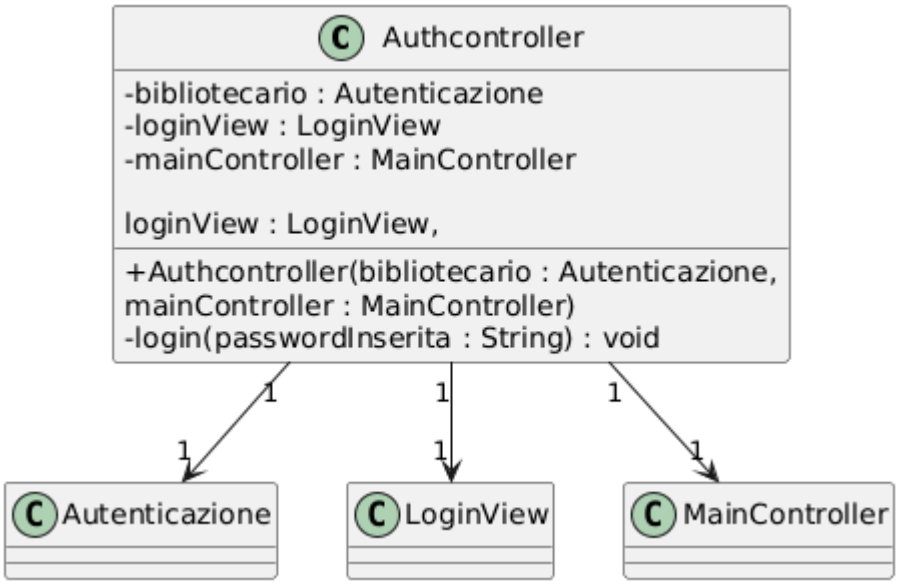
-mostraInfo(messaggio: String) : void :
Mostra nella GUI un messaggio informativo relativo all'esito positivo di un'operazione.

+aggiornaDaModel() : void :
Ricarica i dati dei prestiti dal model e aggiorna la tabella, ad esempio dopo modifiche esterne.

Relazioni:

La classe **PrestitiController** collega la sezione grafica dei prestiti (**PrestitiPanel**) con i gestori del modello (**GestionePrestiti, GestioneLibri, GestioneUtenti**) e con la persistenza (**ArchivioFile**), gestendo registrazione, restituzione e blacklist dei prestiti. Inoltre mantiene riferimenti ai controller di libri e utenti, così da poter aggiornare in modo coordinato le rispettive tabelle quando lo stato dei prestiti, dei libri o degli utenti cambia.

➔ 2.2.5 AuthController CLASS



Attributi

-bibliotecario: Autenticazione:

Modello che contiene i dati del bibliotecario e le informazioni necessarie per l'autenticazione.

-loginView: LoginView :

Schermata di login usata per inserire credenziali e visualizzare eventuali errori.

-mainController: MainController :

Controller principale notificato in caso di login avvenuto con successo.

Metodi

+AuthController(Autenticazione Bibliotecario , LoginView loginView, MainController mainController):

Inizializza il controller di autenticazione collegandolo al modello, alla view di login e al controller principale.

+login(String password): void :

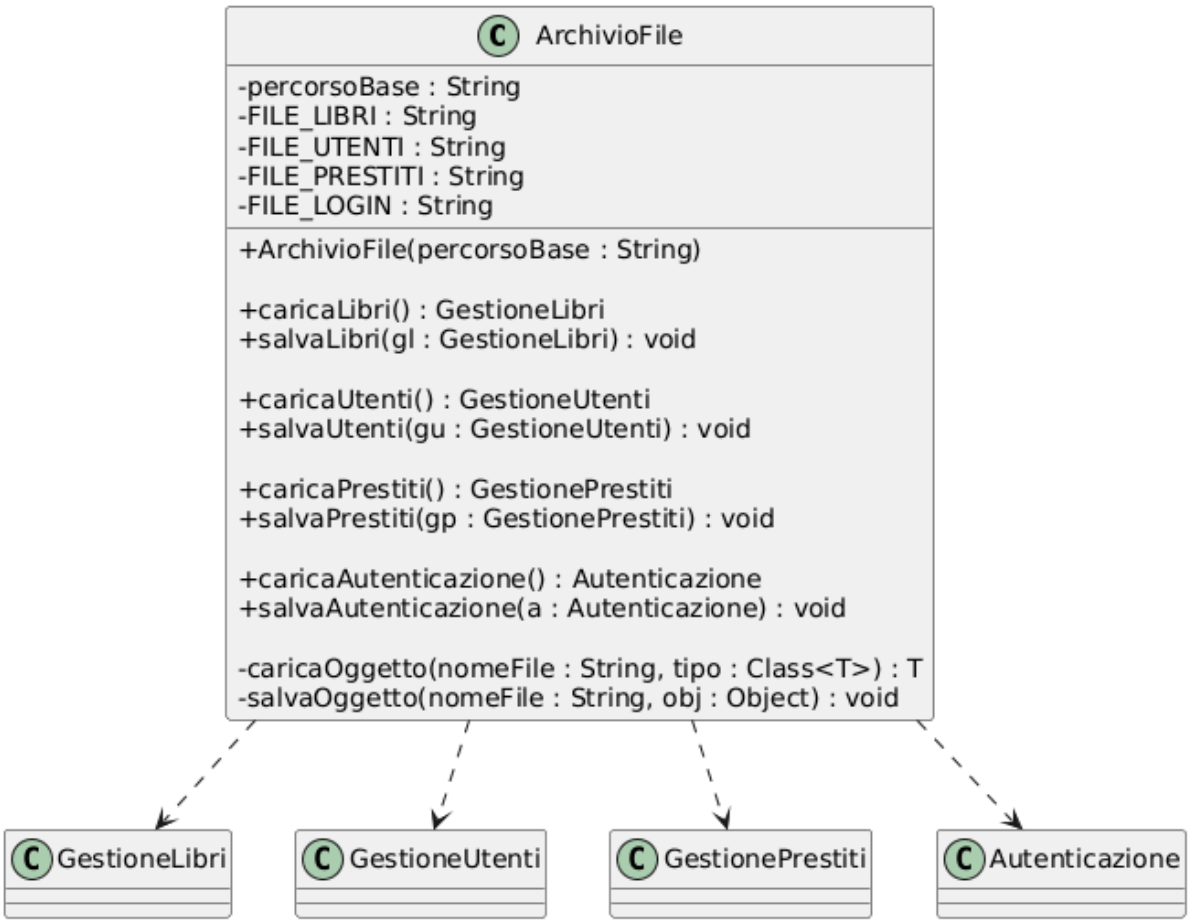
Verifica le credenziali del bibliotecario; in caso di successo chiede al MainController di mostrare la finestra principale.

Relazioni:

La classe **Authcontroller** collega la vista di login (**LoginView**) con il componente di autenticazione (**Autenticazione**) e con il controller principale (**MainController**): usa Autenticazione per verificare la password inserita e, in caso di successo, chiede a **MainController** di passare dal login al menu principale, aggiornando allo stesso tempo i messaggi mostrati nella LoginView.

2.3 PERSISTENZA DATI

➔ 2.3.1 ArchiovioFile CLASS.



Attributi

-percorsoBase: String :
Percorso di base sul file system in cui vengono salvati e letti i file dell'archivio (cartella dei dati).

-FILE_LIBRI: String :
Nome del file che contiene i dati serializzati relativi ai libri.

-FILE_UTENTI: String :
Nome del file che contiene i dati serializzati relativi agli utenti.

-FILE_PRESTITI: String :
Nome del file che contiene i dati serializzati relativi ai prestiti.

-FILE_LOGIN: String :
Nome del file che contiene i dati serializzati relativi alle informazioni di autenticazione del bibliotecario.

Metodi

+ArchivioFile(percorsoBase: String) :
Costruttore che inizializza l'oggetto impostando il percorso di base dove si trovano (o verranno creati) i file di salvataggio.

+caricaLibri() : GestioneLibri :
Legge dal file dei libri e restituisce un oggetto GestioneLibri ricostruito tramite deserializzazione.

+salvaLibri(gl: GestioneLibri) : void :
Salva su file, in forma serializzata, l'oggetto GestioneLibri passato come parametro.

+caricaUtenti() : GestioneUtenti :
Legge dal file degli utenti e restituisce un oggetto GestioneUtenti ricostruito.

+salvaUtenti(gu: GestioneUtenti) : void :
Salva su file l'oggetto GestioneUtenti, sovrascrivendo i dati precedenti.

+caricaPrestiti() : GestionePrestiti :
Legge dal file dei prestiti e restituisce un oggetto GestionePrestiti con i prestiti precedentemente salvati.

+salvaPrestiti(gp: GestionePrestiti) : void :
Salva su file l'oggetto GestionePrestiti, aggiornando lo stato dei prestiti memorizzati.

+caricaAutenticazione() : Autenticazione :
Legge dal file di login e restituisce un oggetto Autenticazione con le informazioni salvate sulla password.

+salvaAutenticazione(a: Autenticazione) : void :
Salva su file l'oggetto Autenticazione, memorizzando l'hash della password corrente.

-caricaOggetto<T>(nomeFile: String, tipo: Class<T>) : T :
Metodo di utilità generico che apre il file indicato, deserializza un oggetto e lo restituisce come istanza del tipo specificato.

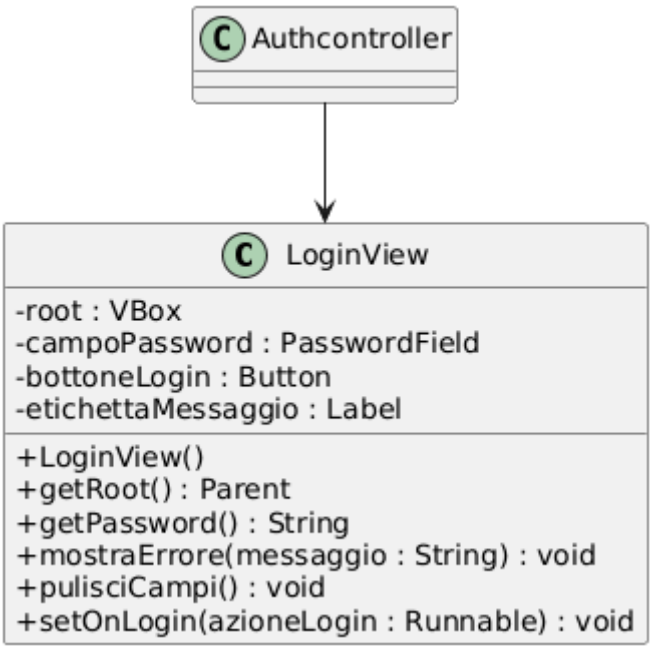
-salvaOggetto(nomeFile: String, obj: Object) : void :
Metodo di utilità generico che serializza l'oggetto passato e lo scrive nel file indicato dal nome ricevuto in input.

Relazioni:

La classe **ArchivioFile** si occupa della persistenza dei dati dell'applicazione: salva e carica su file gli oggetti di **GestioneLibri**, **GestioneUtenti**, **GestionePrestiti** e **Autenticazione**, usando un percorso base e quattro file distinti (per libri, utenti, prestiti e login).

2.4 VIEW

➔ 2.4.1 LoginView CLASS



Attributi

-root: VBox:

Contenitore principale verticale della schermata di login, che il Main utilizza per mostrare tutta la view.

-campoPassword: PasswordField :

Campo di testo per l'inserimento della password.

-bottoneLogin: Button :

Pulsante per inviare la richiesta di login.

-etichettaMessaggio: Label :

Etichetta utilizzata per mostrare messaggi di errore o informazione.

Metodi

+LoginView():

Costruttore che inizializza i componenti grafici della schermata di login.

+getRoot(): Parent :

Restituisce il contenitore grafico principale della schermata di login da mostrare nella finestra.

+getPassword(): String :

Restituisce la password inserita.

+mostraErrore(String messaggio): void :

Visualizza un messaggio di errore nell'etichetta dedicata.

+pulisciCampi(): void :

Svuota i campi di username e password.

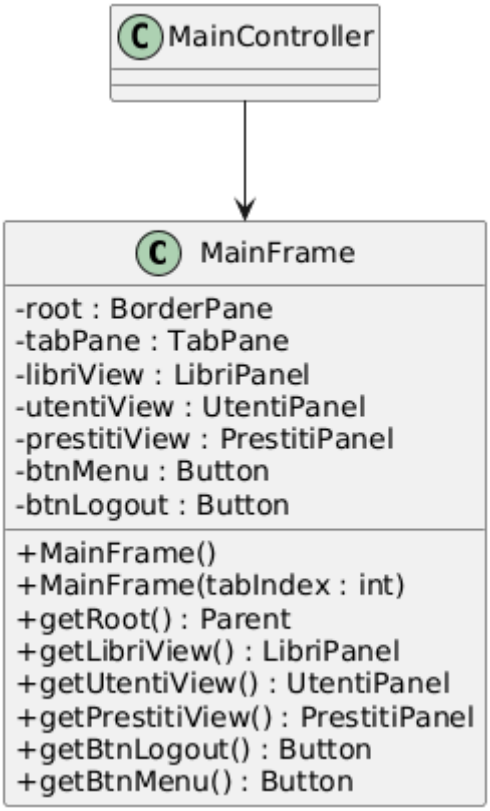
+setOnLogin(Runnable azioneLogin) void :

Imposta il codice da eseguire quando l'utente preme il pulsante "Login"

Relazioni:

La classe **Authcontroller** è il controller dell'autenticazione e mantiene un riferimento a **LoginView**, la schermata di login. Il controller usa la view per leggere la password inserita, mostrare eventuali messaggi di errore e definire cosa succede quando l'utente preme il bottone "Login"; per questo tra Authcontroller e LoginView c'è una relazione di associazione in cui il controller gestisce il comportamento della view.

➔ 2.4.2 MainFrame CLASS



Attributi:

-root: BorderPane :
Contenitore principale della finestra; ospita l'intera interfaccia grafica con barra superiore e area centrale con le tab.

-tabPane: TabPane :
Componente che contiene le schede (tab) della finestra principale: Libri, Utenti e Prestiti.

-libriView: LibriPanel :
Pannello associato alla tab che gestisce la visualizzazione e l'interazione con i libri.

-utentiView: UtentiPanel :
Pannello associato alla tab che gestisce la visualizzazione e l'interazione con gli utenti.

-prestitiView: PrestitiPanel :
Pannello associato alla tab che gestisce la visualizzazione e l'interazione con i prestiti.

-btnMenu: Button :
Pulsante nella barra superiore che permette di tornare al menu principale dell'applicazione.

-btnLogout: Button :
Pulsante nella barra superiore che consente al bibliotecario di effettuare il logout.

Metodi:

+MainFrame() :
Costruttore che crea la finestra principale inizializzando struttura grafica e tab senza specificare una tab iniziale.

+MainFrame(tabIndex : int) :
Costruttore che crea la finestra principale e apre inizialmente la tab indicata dall'indice passato (0 = Libri, 1 = Utenti, 2 = Prestiti).

+getRoot() : Parent :
Restituisce il nodo radice della finestra principale, da usare come contenuto della scena JavaFX.

+getLibriView() : LibriPanel :
Restituisce il pannello dedicato alla gestione dei libri, associato alla relativa tab.

+getUtentiView() : UtentiPanel :
Restituisce il pannello dedicato alla gestione degli utenti.

+getPrestitiView() : PrestitiPanel :
Restituisce il pannello dedicato alla gestione dei prestiti.

+getBtnLogout() : Button :
Restituisce il pulsante di logout, così che il controller possa agganciare l'azione di uscita.

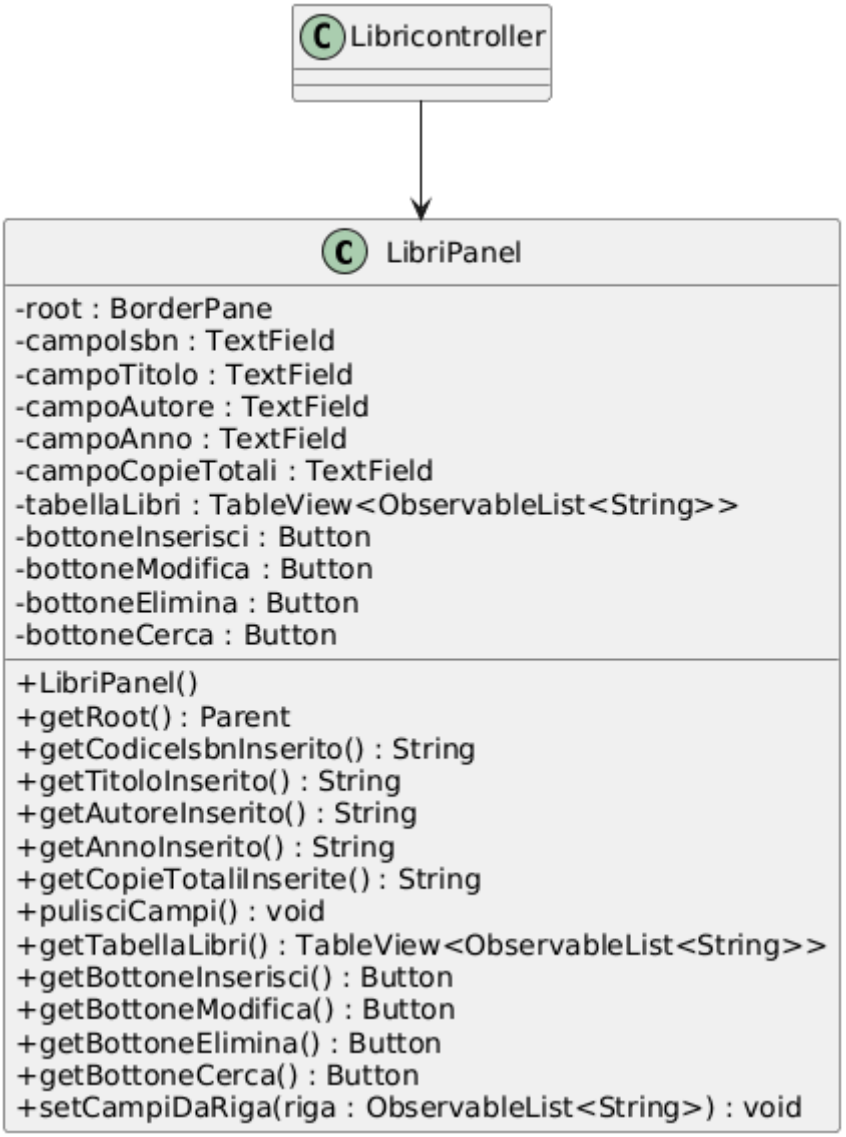
+getBtnMenu() : Button :
Restituisce il pulsante che permette di tornare al menu principale, per collegare la relativa azione nel controller

---

Relazioni:

La classe **MainController** è il controller principale dell'applicazione e mantiene un riferimento a **MainFrame**, che è la finestra operativa con le tab Libri, Utenti e Prestiti. Il controller crea il MainFrame, lo mostra nella Stage e usa i suoi getter per collegare le varie view ai rispettivi controller e per gestire i pulsanti di navigazione; tra MainController e MainFrame c'è quindi una relazione di associazione in cui il controller gestisce la finestra principale.

➔ 2.4.3 LibriPanel CLASS



Attributi:

-tabellaLibri: Table :

Tabella che visualizza l'elenco dei libri presenti in biblioteca.

-campoCodiceIsbn: TextField :

Campo per l'inserimento o il filtro del codice ISBN.

-campoTitolo: TextField :

Campo per l'inserimento o il filtro del titolo del libro.

-campoAutore: TextField :

Campo per l'inserimento o il filtro dell'autore.

-campoAnno: TextField :

Campo per l'inserimento dell'anno di pubblicazione.

-campoCopieTotali: TextField :

Campo per l'inserimento del numero di copie totali.

-bottoneInserisci: Button :

Pulsante per inserire un nuovo libro.

-bottoneModifica: Button :

Pulsante per modificare un libro esistente selezionato.

-bottoneElimina: Button :

Pulsante per eliminare un libro selezionato.

-bottoneCerca: Button :

Pulsante per effettuare la ricerca nel catalogo.

-controller: LibriController :

Controller che gestisce le azioni relative ai libri.

Metodi

+LibriPanel():

Costruttore che inizializza i componenti grafici del pannello libri.

+**getRoot() : Parent** :

Restituisce il nodo radice del pannello libri, da utilizzare come contenuto nella scena JavaFX.

+getCodiceIsbnInserito(): int :

Restituisce il codice ISBN inserito nel relativo campo.

+getTitoloInserito(): String :

Restituisce il titolo inserito.

+getAutoreInserito(): String :

Restituisce l'autore inserito.

+getAnnoInserito(): int :

Restituisce l'anno di pubblicazione inserito (convertito da testo a intero).

+getCopieTotaliInserite(): int :

Restituisce il numero di copie totali inserito.

+**pulisciCampi() : void** :

Cancella il contenuto di tutti i campi di input del pannello, riportandoli vuoti.

+getLibroSelezionato(): Libro :

Restituisce il libro attualmente selezionato nella tabella, se presente.

+**getTabellaLibri() : TableView<ObservableList<String>>** :
Restituisce la tabella che visualizza l'elenco dei libri presenti in biblioteca.

+**getBottoneInserisci() : Button** :
Restituisce il pulsante usato per confermare l'inserimento di un nuovo libro.

+**getBottoneModifica() : Button** :
Restituisce il pulsante che permette di confermare la modifica dei dati del libro selezionato.

+**getBottoneElimina() : Button** :
Restituisce il pulsante che consente di eliminare il libro selezionato dalla tabella.

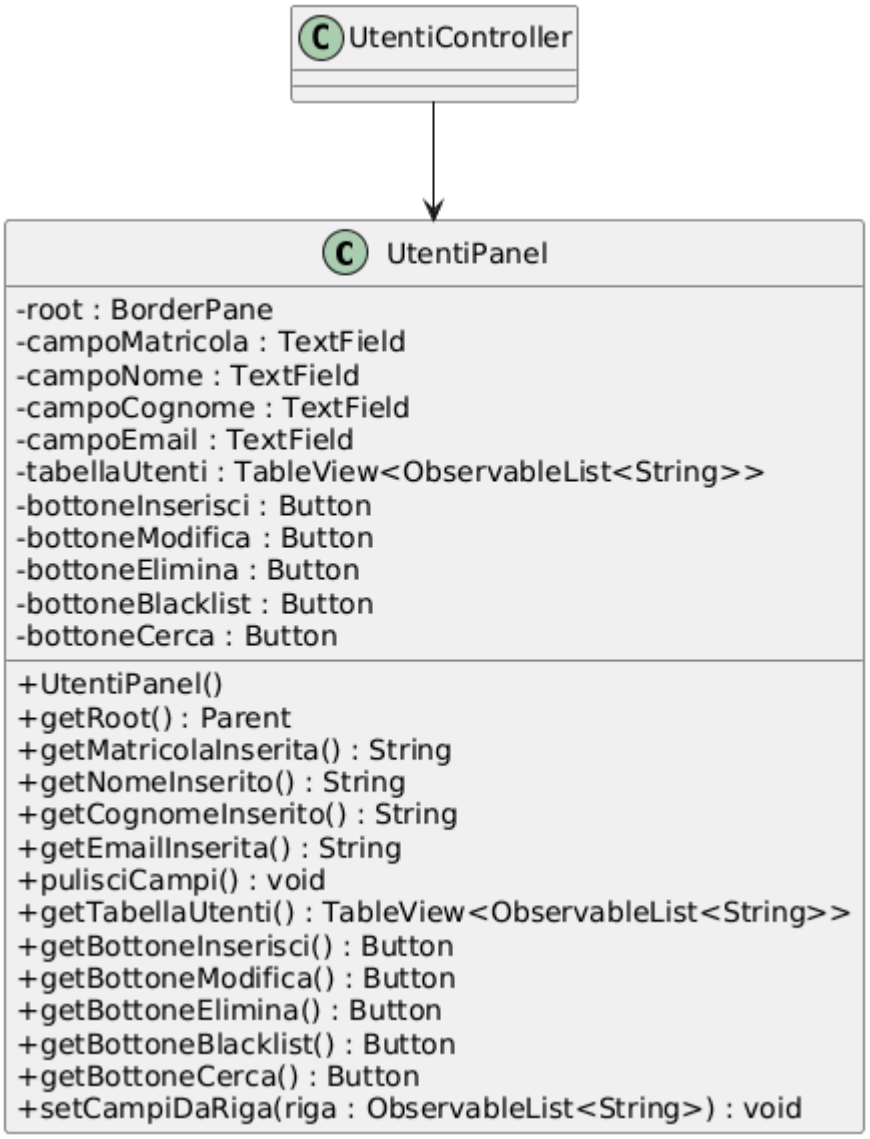
+**getBottoneCerca() : Button** :
Restituisce il pulsante utilizzato per avviare la ricerca dei libri in base ai dati inseriti nei campi.

+**setCampiDaRiga(riga : ObservableList<String>) : void** :
Imposta automaticamente i campi di input del pannello con i valori presenti nella riga selezionata della tabella libri.

Relazioni:

La classe **Libricontroller** è il controller della sezione libri e mantiene un riferimento a **LibriPanel**, che rappresenta la parte grafica. Il controller usa LibriPanel per leggere i dati inseriti dall'utente nei campi, aggiornare la tabella dei libri e collegare i pulsanti alle operazioni sul modello; la relazione è quindi di associazione 1–1, in cui il controller "gestisce" la view senza che la view conosca il controller.

➔ 2.4.4 UtentiPanel CLASS



Attributi

-tabellaUtenti: Table :
Tabella che visualizza l'elenco degli utenti registrati.

-campoMatricola: TextField :
Campo per l'inserimento o il filtro della matricola.

-campoNome: TextField :
Campo per l'inserimento del nome.

-campoCognome: TextField :
Campo per l'inserimento del cognome.

-campoEmail: TextField :
Campo per l'inserimento dell'email istituzionale.

-bottoneInserisci: Button :
Pulsante per inserire un nuovo utente.

-bottoneModifica: Button :
Pulsante per modificare l'utente selezionato.

-bottoneElimina: Button :
Pulsante per eliminare l'utente selezionato.

-bottoneCerca: Button :
Pulsante per cercare utenti in base al filtro.

-bottoneBlacklist: Button :
Pulsante per aggiungere/rimuovere un utente dalla blacklist.

-controller: UtentiController :
Controller che gestisce le azioni relative agli utenti.

Metodi

+UtentiPanel():
Costruttore che inizializza i componenti grafici del pannello utenti.

+getRoot() : Parent :

Restituisce il nodo radice del pannello utenti, da utilizzare come contenuto nella scena JavaFX.

+getMatricolaInserita(): int :
Restituisce la matricola inserita nel relativo campo.

+getNomeInserito(): String :
Restituisce il nome inserito.

+getCognomeInserito(): String :
Restituisce il cognome inserito.

+getEmailInserita(): String :
Restituisce l'email istituzionale inserita.

+getUtenteSelezionato(): Utente :
Restituisce l'utente selezionato nella tabella, se presente.

+pulisciCampi(): void :
Svuota i campi di input del pannello.

+getTabellaUtenti() : TableView<ObservableList<String>> :
Restituisce la tabella che visualizza l'elenco degli utenti presenti nel sistema.

+getBottoneInserisci() : Button :
Restituisce il pulsante usato per confermare l'inserimento di un nuovo utente.

+getBottoneModifica() : Button :
Restituisce il pulsante che permette di confermare la modifica dei dati dell'utente selezionato.

+getBottoneElimina() : Button :
Restituisce il pulsante che consente di eliminare l'utente selezionato dalla tabella.

+getBottoneBlacklist() : Button :
Restituisce il pulsante che permette di aggiungere o rimuovere l'utente selezionato dalla blacklist.

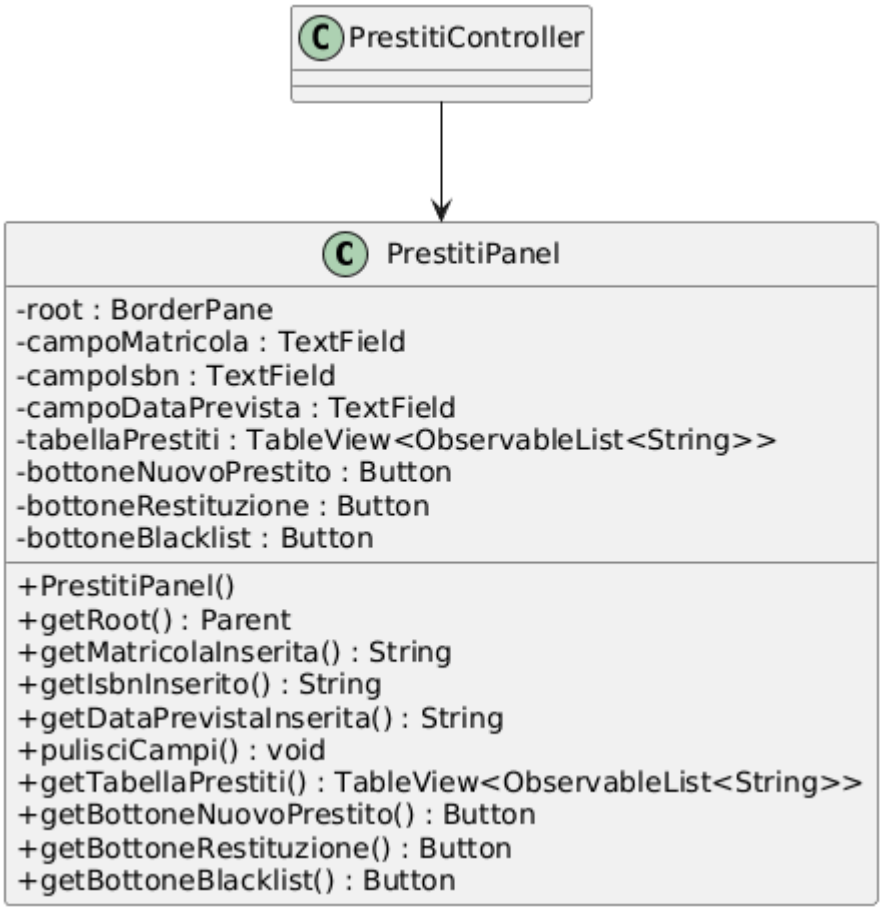
+getBottoneCerca() : Button :
Restituisce il pulsante utilizzato per avviare la ricerca degli utenti in base ai dati inseriti nei campi.

+setCampiDaRiga(riga : ObservableList<String>) : void :
Imposta automaticamente i campi di input del pannello con i valori presenti nella riga selezionata della tabella utenti.

Relazioni:

La classe **UtentiController** è il controller della sezione anagrafica utenti e mantiene un riferimento a **UtentiPanel**, che rappresenta la schermata grafica con form, tabella e pulsanti; tra UtentiController e UtentiPanel c'è quindi una relazione di associazione in cui il controller gestisce la view.

➔ 2.4.5 PrestitiPanel CLASS



Attributi:

-tabellaPrestit: Table :
Tabella che visualizza l'elenco dei prestiti (ad esempio quelli attivi).

-campoMatricola: TextField :
Campo per l'inserimento della matricola dell'utente che richiede il prestito.

-campoCodiceIsbn: TextField :
Campo per l'inserimento del codice ISBN del libro.

-campoDataPrevista: TextField :

Campo di inserimento della data prevista di restituzione (o componente specifico per la data).

-bottoneNuovoPrestito : Button :
Pulsante per registrare un nuovo prestito.

-bottoneRestituzione: Button :
Pulsante per registrare la restituzione del prestito selezionato.

-bottoneBlacklist: Button :
Pulsante per inserire in blacklist l'utente selezionato o associato al prestito.

-controller: PrestitiController :
Controller che gestisce le azioni relative ai prestiti.

Metodi:

+PrestitiPanel():
Costruttore che inizializza i componenti grafici del pannello prestiti.

+mostraPrestiti(List<Prestito> prestiti): void :
Aggiorna il contenuto della tabella con l'elenco dei prestiti passato.

+getMatricolaInserita(): int :

Restituisce la matricola inserita per creare o filtrare un prestito.

+getCodiceIsbnInserito(): int :

Restituisce il codice ISBN inserito per creare o filtrare un prestito.

+getDataPrevistaInserita(): LocalDate :

Restituisce la data prevista di restituzione inserita, convertita in LocalDate.

+getPrestitoSelezionato(): Prestito :

Restituisce il prestito selezionato nella tabella, se presente.

+mostraMessaggio(String messaggio): void :

Mostra un messaggio informativo o di errore relativo alle operazioni sui prestiti.

+pulisciCampi(): void :

Svuota i campi di input del pannello.

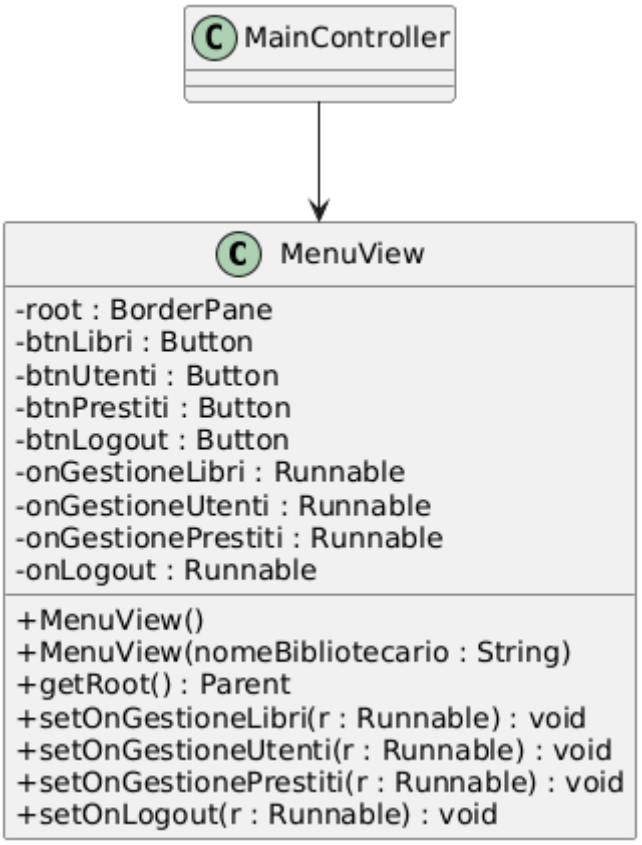
+setController(PrestitiController controller): void :

Registra il controller che gestirà le azioni del pannello.

Relazioni:

La classe **PrestitiController** è il controller della sezione prestiti e mantiene un riferimento a **PrestitiPanel**, che rappresenta l'interfaccia grafica per visualizzare e gestire i prestiti. Il controller legge dalla view i valori dei campi (matricola, ISBN, data prevista), aggiorna la tabella e collega i tre pulsanti alle operazioni sul modello; per questo tra PrestitiController e PrestitiPanel c'è una relazione di associazione in cui il controller gestisce la view.

➔ 2.4.6 MenuView CLASS



Attributi:

-root: BorderPane :

Pannello principale che contiene la struttura grafica del menu dell'applicazione.

-btnLibri: Button :

Pulsante che permette di accedere alla sezione di gestione dei libri.

-btnUtenti: Button :

Pulsante che permette di accedere alla sezione di gestione degli utenti.

-btnPrestiti: Button :

Pulsante che permette di accedere alla sezione di gestione dei prestiti.

-btnLogout: Button :

Pulsante che consente al bibliotecario di effettuare il logout dall'applicazione.

-onGestioneLibri: Runnable :

Riferimento all'azione da eseguire quando l'utente seleziona la gestione dei libri.

-onGestioneUtenti: Runnable :

Riferimento all'azione da eseguire quando l'utente seleziona la gestione degli utenti.

-onGestionePrestiti: Runnable :

Riferimento all'azione da eseguire quando l'utente seleziona la gestione dei prestiti.

-onLogout: Runnable :

Riferimento all'azione da eseguire quando l'utente preme il pulsante di logout.

Metodi:

+MenuView() :

Costruttore che crea e inizializza l'interfaccia del menu senza parametri aggiuntivi.

+MenuView(nomeBibliotecario: String) :

Costruttore che crea il menu inizializzando eventualmente elementi grafici legati al nome del bibliotecario loggato.

+getRoot() : Parent :

Restituisce il nodo radice dell'interfaccia del menu, da impostare come contenuto della scena JavaFX.

+setOnGestioneLibri(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione dei libri (clic sul pulsante corrispondente).

+setOnGestioneUtenti(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione degli utenti.

+setOnGestionePrestiti(r: Runnable) : void :

Imposta l'azione da eseguire quando viene selezionata la gestione dei prestiti.

+setOnLogout(r: Runnable) : void :

Imposta l'azione da eseguire quando viene premuto il pulsante di logout.

Relazioni:

La classe **MainController** crea e gestisce la **MenuView**, che rappresenta il menu principale dopo il login; per questo tra MainController e MenuView c'è una relazione di associazione in cui il controller controlla il comportamento della schermata di menu.

2.5 Coesione, accoppiamento e principi di buona progettazione.

➔ 2.5.1 Coesione (Alta – funzionale)

La coesione misura quanto gli elementi all'interno dello stesso modulo sono legati tra loro (intra-modulo).

Nel nostro sistema ogni classe e ogni package ha un compito ben definito:

Nel package **model** abbiamo un'alta coesione funzionale:

le classi di dominio **Libro**, **Utente**, **Prestito**, **Autenticazione** rappresentano singoli concetti del dominio e incapsulano solo i dati e i metodi che li riguardano;

le classi di servizio **GestioneLibri**, **GestioneUtenti** e **GestionePrestiti** si occupano rispettivamente della gestione del catalogo libri, dell'anagrafica utenti e del ciclo di vita dei prestiti, senza mischiare responsabilità;

eventuali classi di supporto per la persistenza (es. **DatiBiblioteca**) raggruppano solo i dati da salvare/caricare.

Questa separazione netta del modello si riflette anche negli altri package:

nel package **controller** ogni controller gestisce una sola funzionalità (**LibriController** solo libri, **UtentiController** solo utenti, **PrestitiController** solo prestiti), evitando un "mega-controller" che fa tutto;

nel package view ogni pannello grafico (**LibriPanel**, **UtentiPanel**, **PrestitiPanel**, ecc.) è responsabile solo della propria parte di interfaccia.

In questo modo ogni modulo ha una responsabilità chiara e coerente, e all'interno dello stesso modulo le parti lavorano tutte allo stesso obiettivo.

➔ 2.5.2 Accoppiamento (Basso / Gestito)

L'accoppiamento è una caratteristica che misura il grado di interdipendenza tra moduli diversi.

Nel sistema l'accoppiamento è mantenuto basso e controllato:

Le dipendenze seguono una struttura gerarchica e unidirezionale:

view → controller → model → persistence

i livelli inferiori non conoscono quelli superiori (ad esempio il **model** non dipende mai dalla **GUI**).

Nelle classi di servizio usiamo principalmente un accoppiamento per dati: si passano ai metodi solo le informazioni strettamente necessarie (es. stringhe, id, parametri primitivi), riducendo l'impatto delle modifiche.

Tra le classi di dominio utilizziamo un accoppiamento per timbro (stamp coupling) quando ha senso passare l'intero oggetto, sfruttando i suoi metodi e il polimorfismo: ad esempio **Prestito** mantiene riferimenti a **Utente** e **Libro**

per poter chiedere direttamente i dati necessari (nome utente, titolo libro, ecc.).

Le poche dipendenze trasversali (ad esempio **GestionePrestiti** che usa **GestioneLibri** e **GestioneUtenti** per verificare disponibilità e stato in blacklist) sono esplicite e motivate dal dominio, e sono comunque concentrate nei servizi di gestione, non disperse in tutta l'applicazione.

➔ 2.5.3 Principi di buona progettazione

Essendo un sistema progettato con l'approccio object-oriented, sono stati adottati alcuni principi di buona progettazione per ridurre il debito tecnico ed evitare complessità inutili, seguendo linee guida come KISS (Keep It Simple, Stupid) e DRY (Don't Repeat Yourself).

Il principio maggiormente rispettato è il Single Responsibility Principle (SRP): ogni classe svolge un solo compito ben definito (es. **GestioneLibri** gestisce solo libri, **GestioneUtenti** solo utenti, **GestionePrestiti** solo prestiti, **ArchivioFile** solo la persistenza), e ogni package ha una responsabilità chiara (**model, controller, view, persistence**).

Grazie a questa organizzazione, il sistema è anche vicino al principio di Open/Closed: è possibile estendere le funzionalità (ad esempio aggiungendo nuovi controlli, nuovi tipi di ricerche o nuove viste) senza dover modificare pesantemente le classi esistenti.

L'uso di ereditarietà e polimorfismo consente di scrivere codice che lavora in termini di interfacce/tipi generali, rispettando i principi SOLID più rilevanti per questo progetto (in particolare SRP e, in parte, OCP e ISP).

Infine, sono stati inseriti controlli sugli input e gestione degli errori (ad esempio ritorni di false, eccezioni controllate e messaggi alla GUI) per proteggere lo stato interno degli oggetti: dati non validi non compromettono la stabilità del sistema. Questo va nella direzione del principio di robustezza, rendendo il comportamento prevedibile anche in caso di input errati.

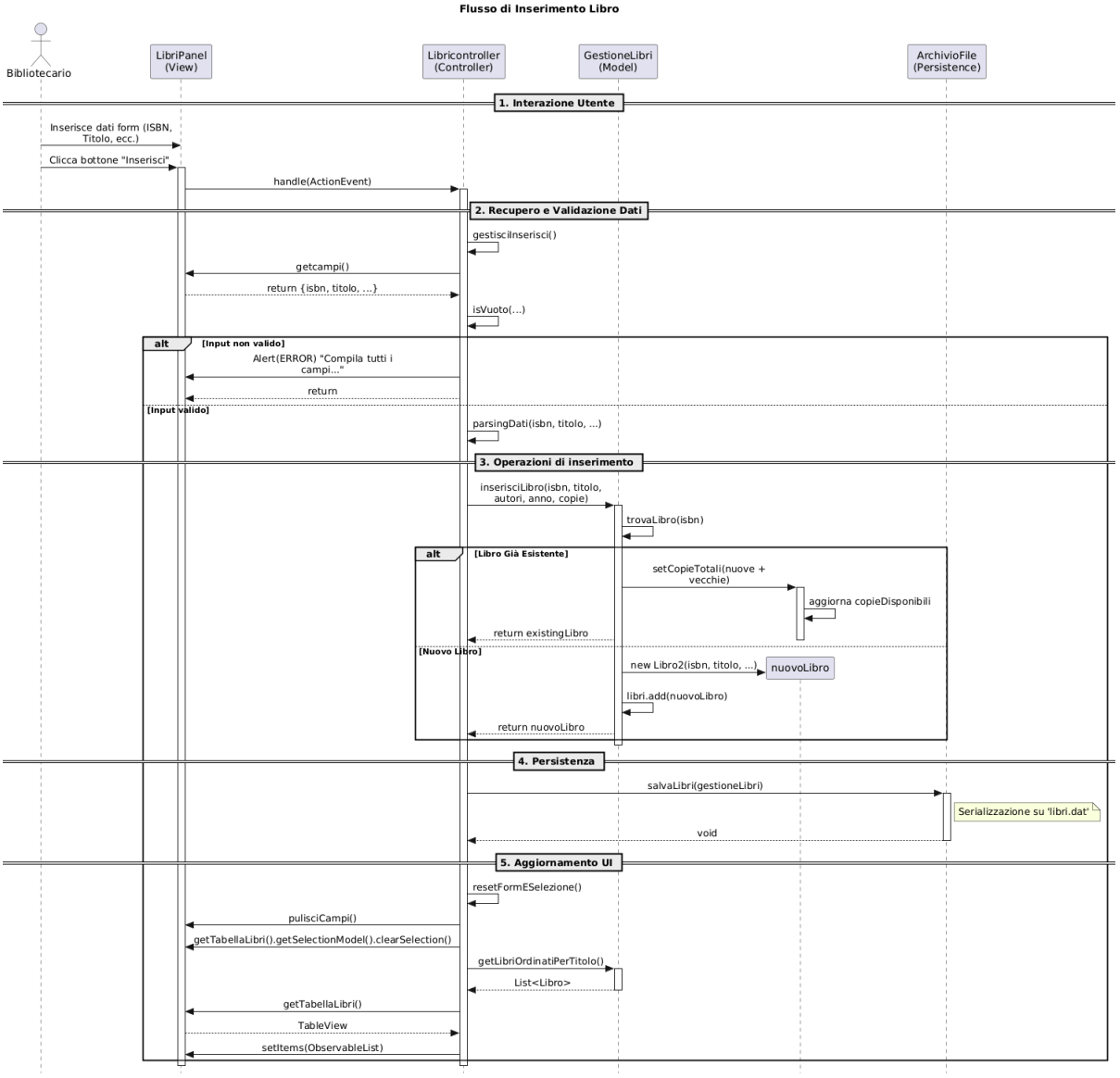
3. Modello dinamico

3.3 Diagrammi di interazione

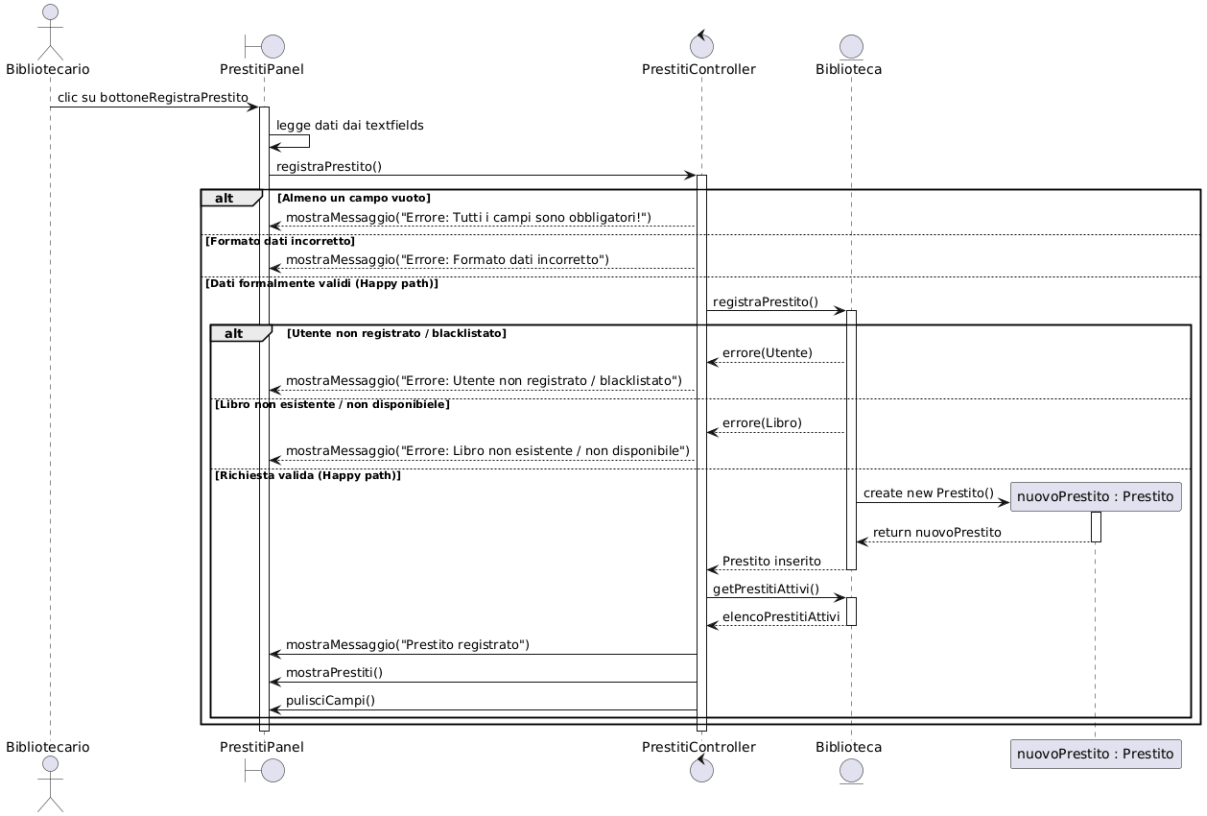
DIAGRAMMI DI SEQUENZA

Diagrammi di sequenza: si concentrano sulla sequenza dei messaggi tra gli oggetti.

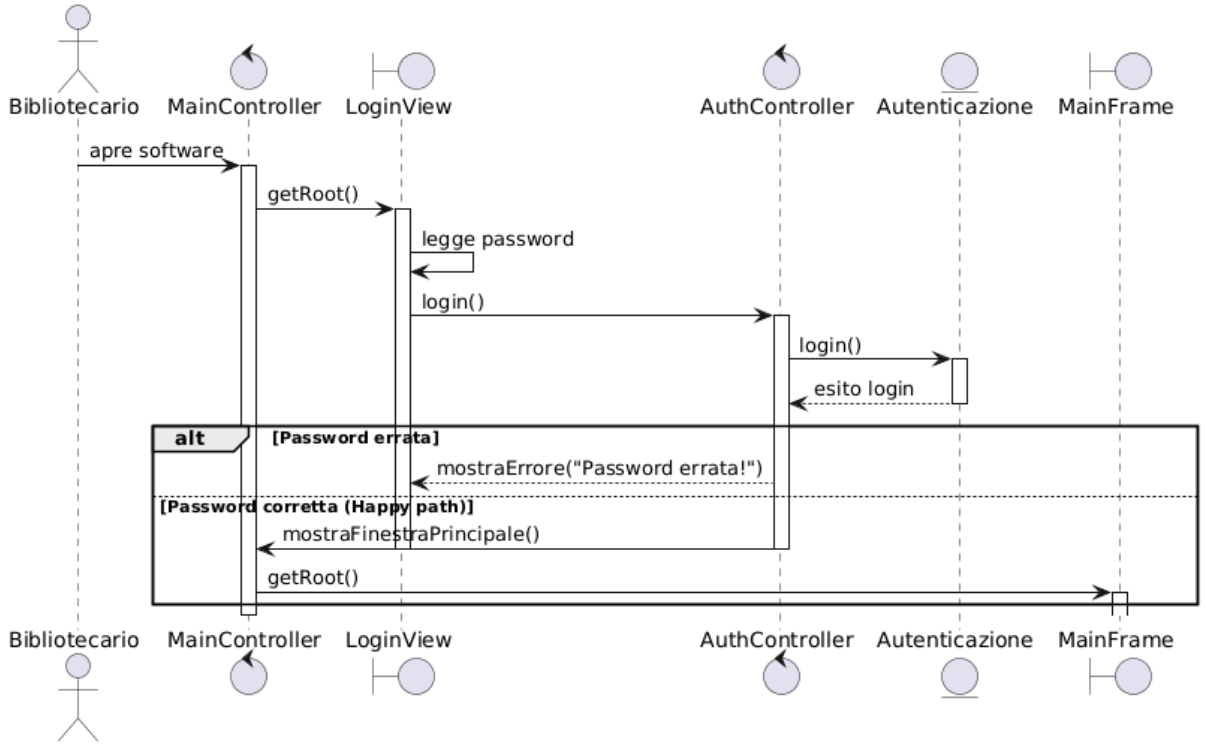
UC 01 Ins libro



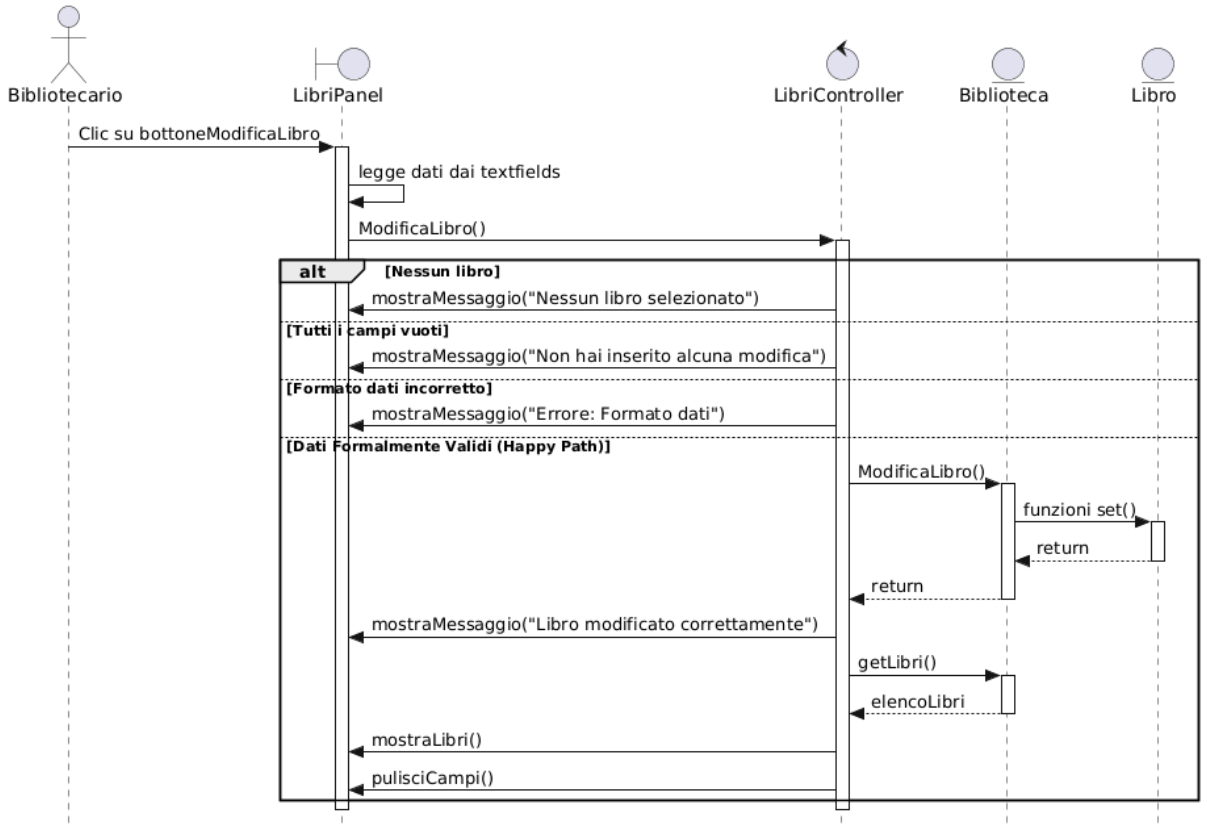
uc 10 registrazione prestito



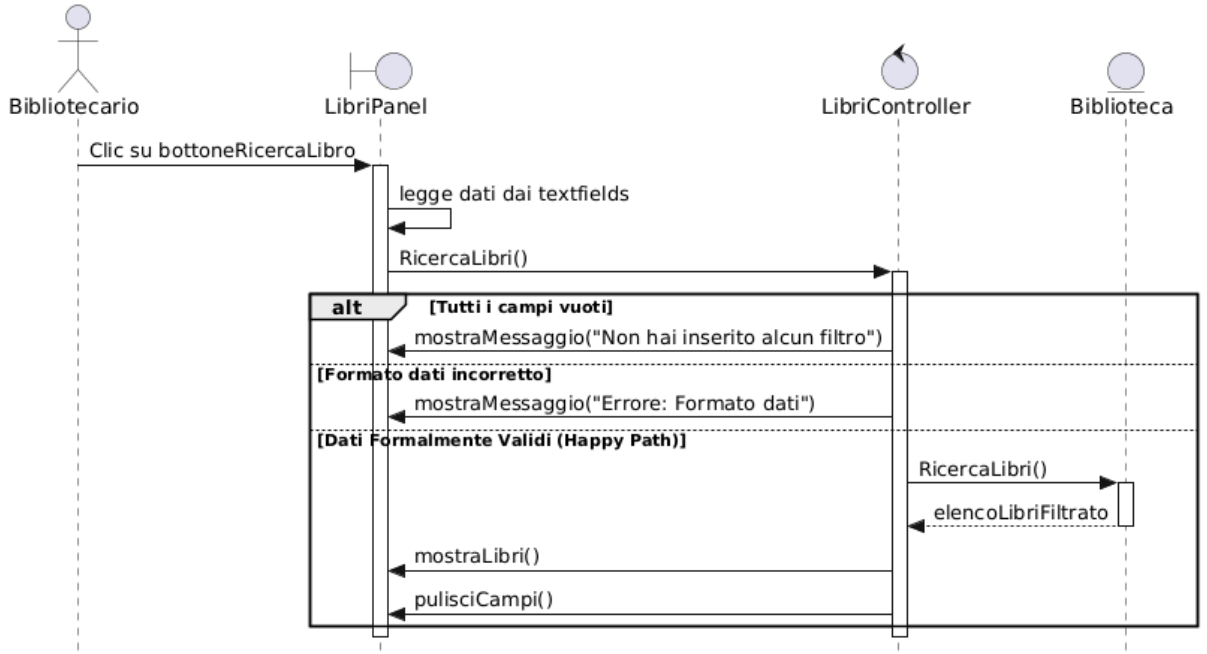
UC Login

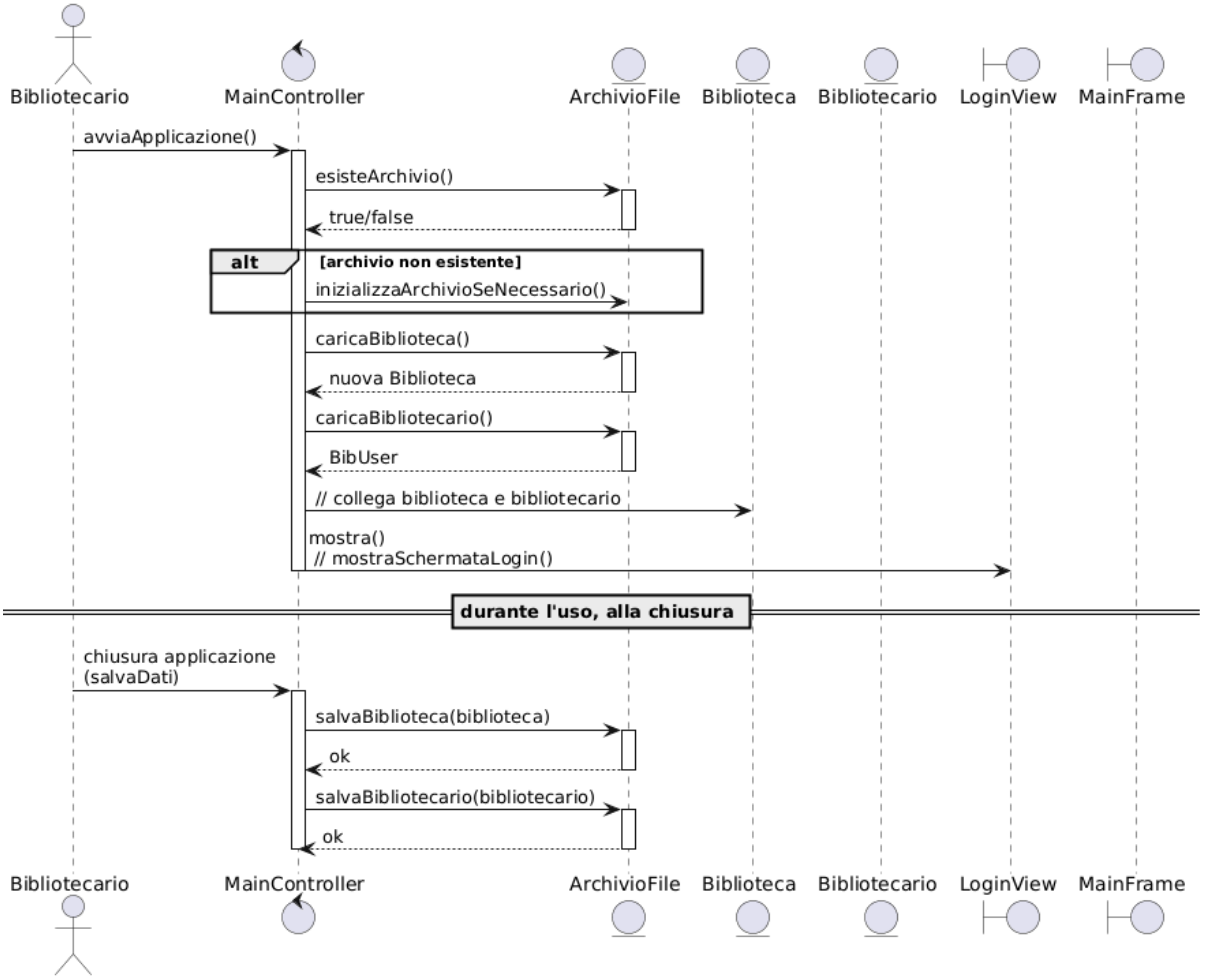


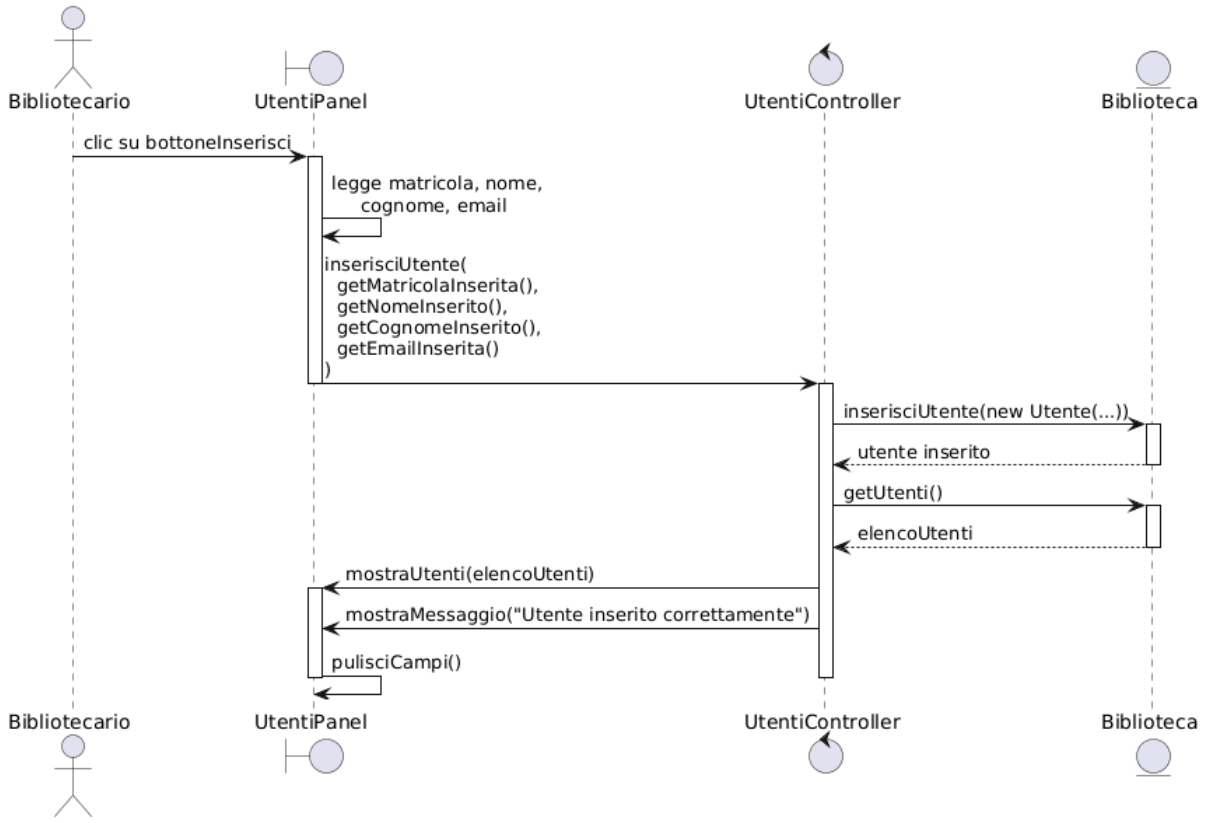
UC Modifica libro



UC ricerca libro




Bibliotecario
MainController
ArchivioFile
Biblioteca
Bibliotecario
LoginView
MainFrame
avviaApplicazione()
esisteArchivio()
true/false
alt
[archivio non esistente]
inizializzaArchivioSeNecessario()
caricaBiblioteca()
nuova Biblioteca
caricaBibliotecario()
BibUser
// collega biblioteca e bibliotecario
mostra()
// mostraSchermataLogin()
durante l'uso, alla chiusura
chiusura applicazione
(salvaDati)
salvaBiblioteca(biblioteca)
ok
salvaBibliotecario(bibliotecario)
ok
Bibliotecario
MainController
ArchivioFile
Biblioteca
Bibliotecario
LoginView
MainFrame

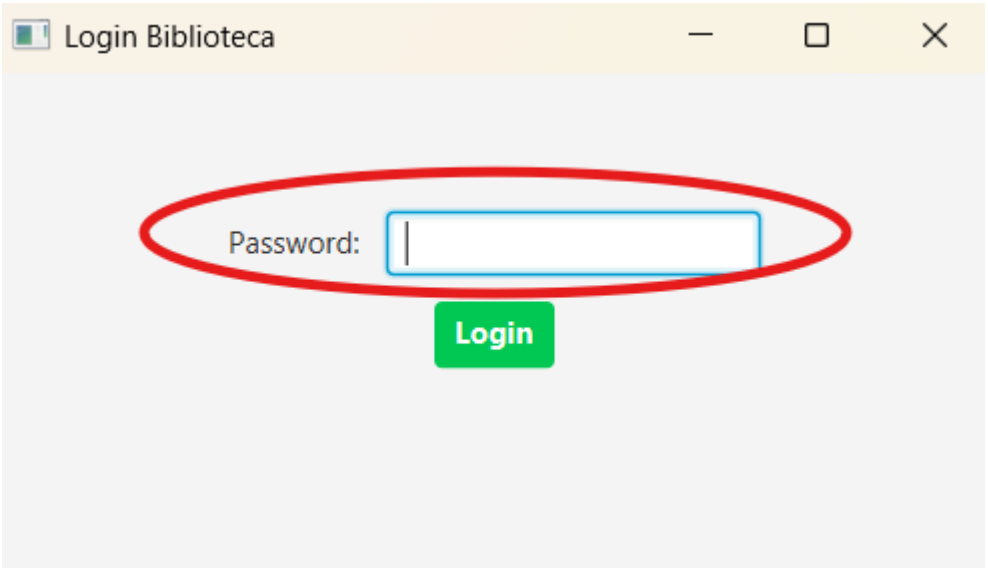

Bibliotecario
UtentiPanel
UtentiController
Biblioteca
clic su bottoneInserisci
legge matricola, nome,
cognome, email
inserisciUtente(
getMatricolaInserita(),
getNomeInserito(),
getCognomeInserito(),
getEmailInserita()
)
inserisciUtente(new Utente(...))
utente inserito
getUtenti()
elencoUtenti
mostraUtenti(elencoUtenti)
mostraMessaggio("Utente inserito correttamente")
pulisciCampi()
Bibliotecario
UtentiPanel
UtentiController
Biblioteca

4. User Interface

In questa sezione viene mostrata una sequenza di screen nei quali sono visualizzate le possibili interazioni dell'utente con la rubrica.

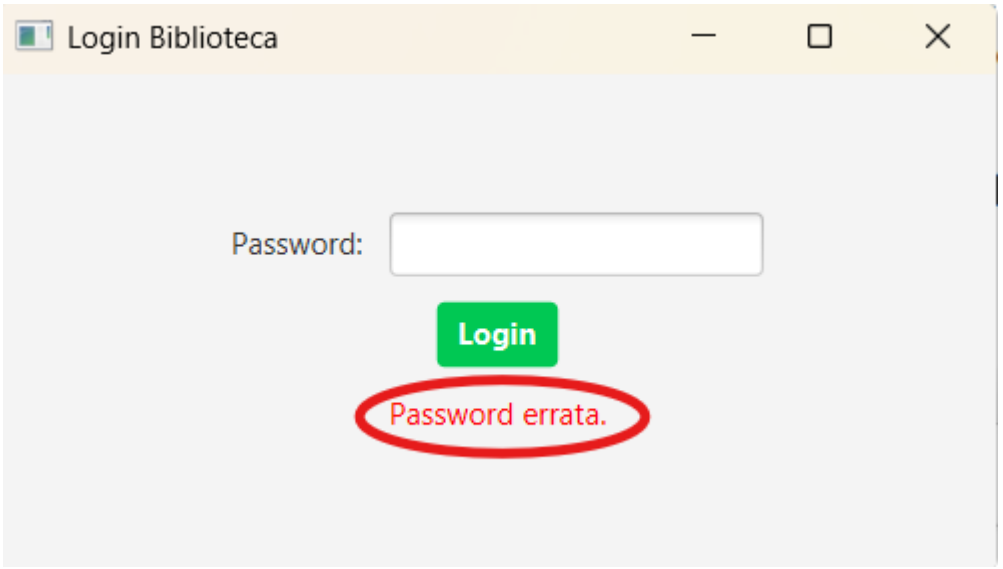
4.1 Login

Prima: l'utente inserisce nel textField una sequenza di caratteri (password).



Dopo: se la password è corretta, l'autenticazione va a buon fine, e viene aperta la finestra del menu.

Altrimenti: se la password è errata la finestra di login rimane visibile e viene mostrato un messaggio di errore

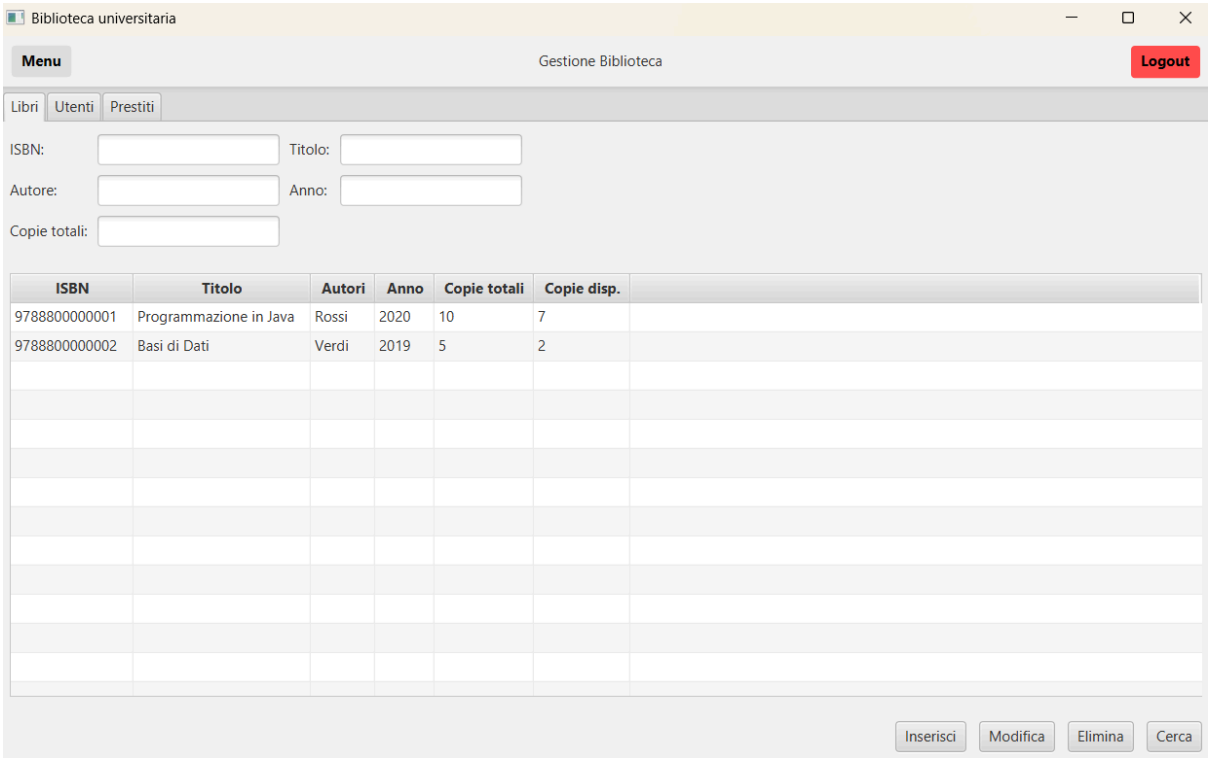


4.2 Navigazione dal menu principale

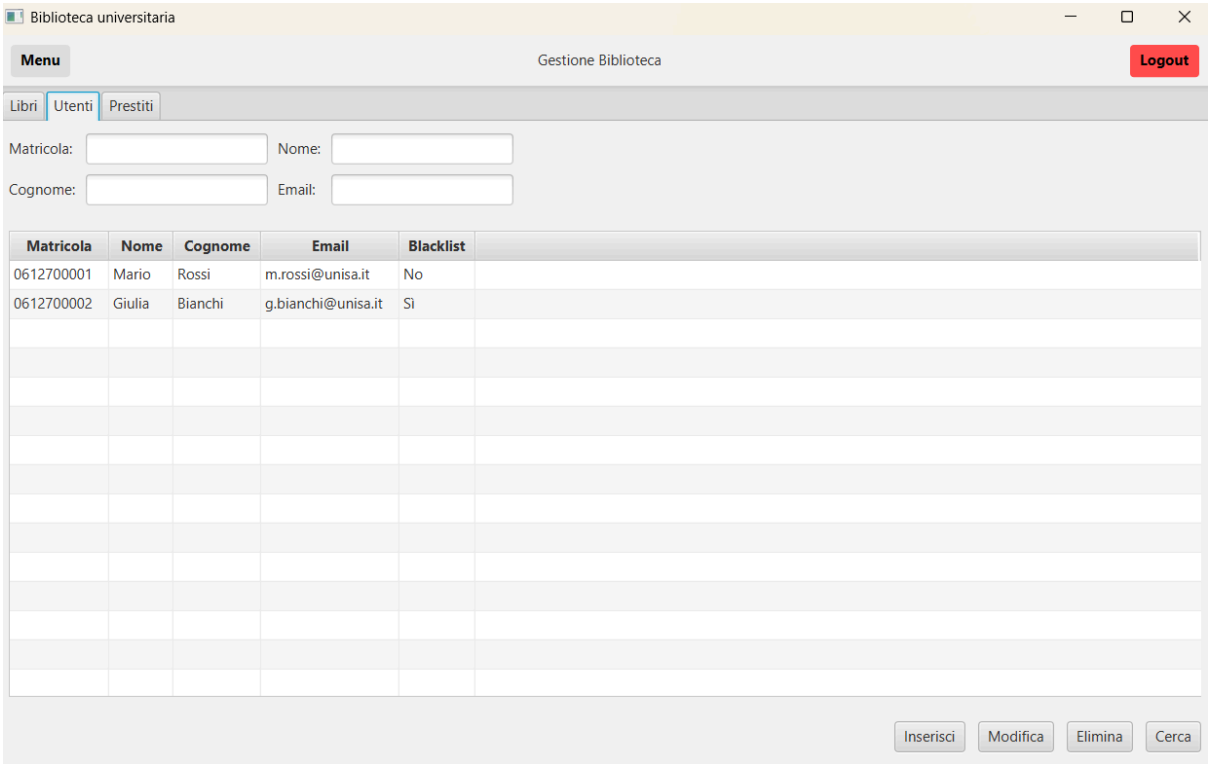
Prima: Viene mostrata la finestra Mnu Biblioteca, con i pulsanti Gestione Libri, Gestione Utenti, Gestione Prestiti e Logout.

Dopo: Se il bibliotecario clicca su uno dei pulsanti di gestione, si apre la corrispondente schermata.

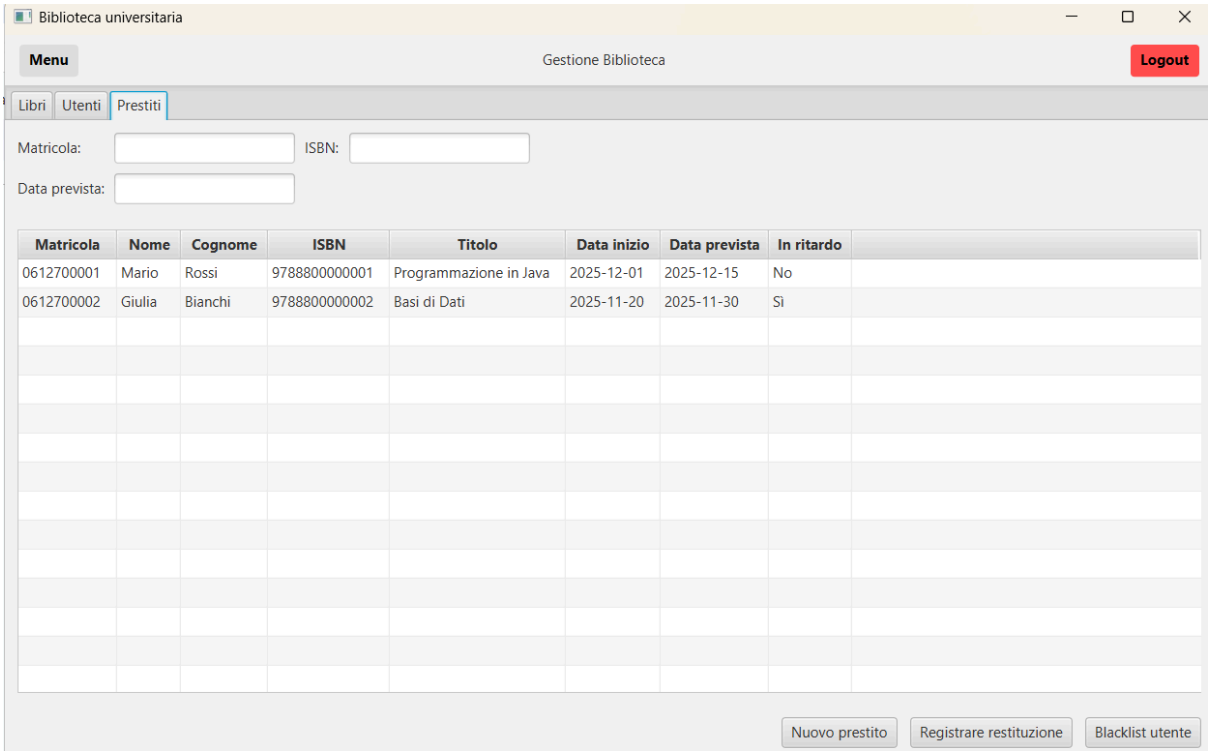
4.3 Gestione Libri



4.4: Gestione Utenti



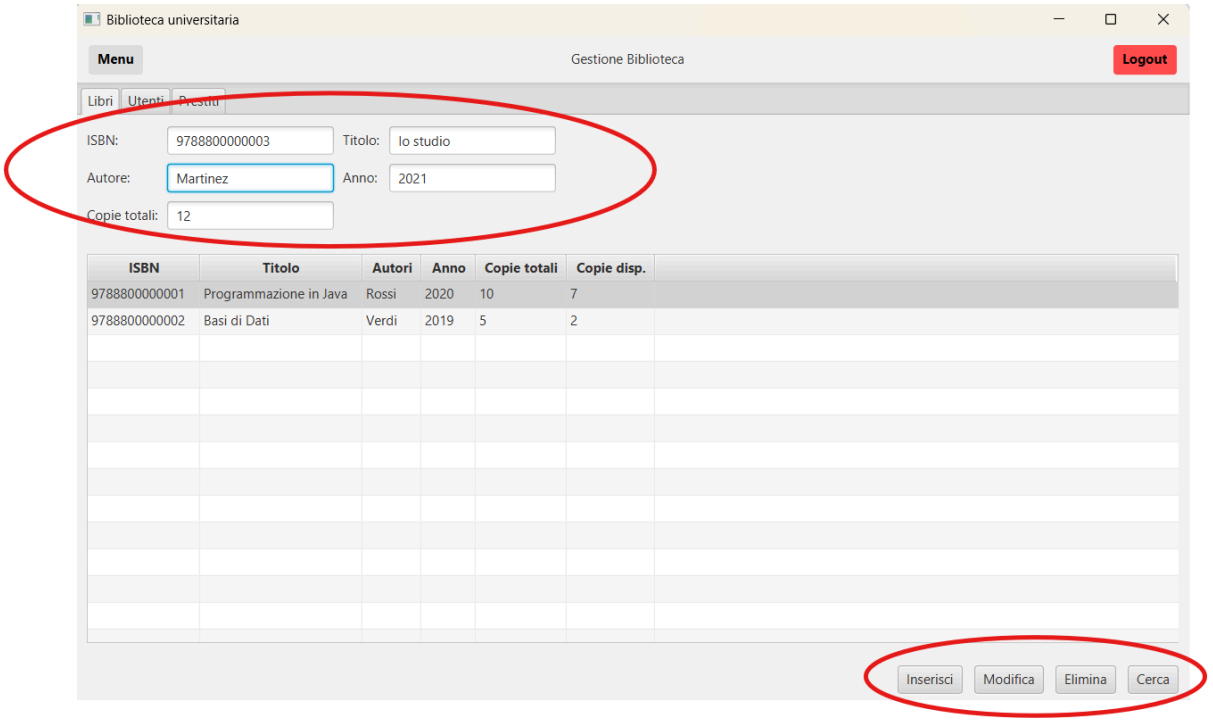
4.5: Gestione Prestiti



Altrimenti: se il bibliotecario clicca su logout, la finestra del menu si chiude e l'applicazione torna alla schermata di login (4.1).

4.3.1: Panoramica scheda gestione libri

Prima: Nella scheda **Libri** il bibliotecario inserisce nei campi ISBN, Titolo, Autore, Anno e Copie totali i dati del libro (oppure seleziona un libro dalla tabella) e clicca sul pulsante **Inserisci**, **Modifica**, **Elimina** oppure **Cerca**.



Dopo: La tabella dei libri viene aggiornata: in caso di inserimento o modifica mostra la nuova riga con i dati corretti, in caso di eliminazione il libro scompare dall'elenco, mentre con **Cerca** rimangono visibili solo i volumi che soddisfano i criteri indicati.

4.3.2 Inserimento libro

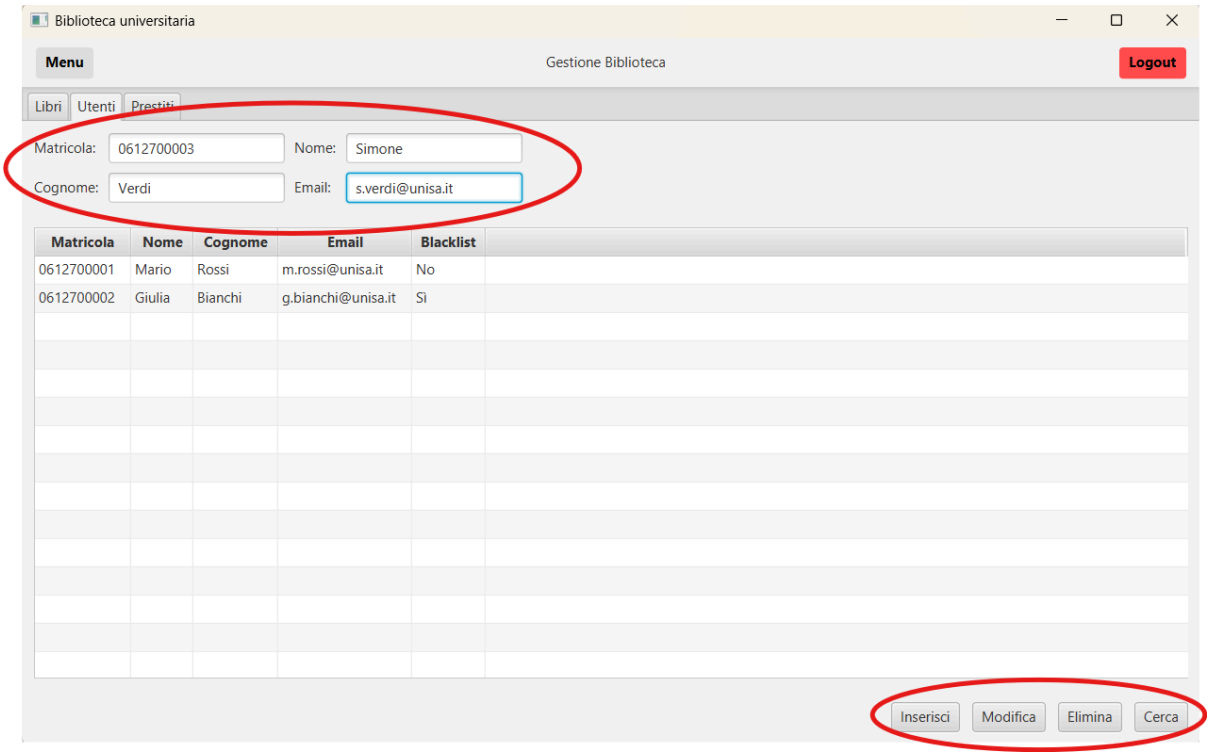
4.3.3 Modifica libro

4.3.4 Eliminazione libro

4.3.5 Ricerca libro

4.4.1 Panoramica scheda gestione utenti

Prima: Nella scheda **Utenti** il bibliotecario compila i campi Matricola, Nome, Cognome ed Email (oppure seleziona una riga dalla tabella) e clicca su **Inserisci**, **Modifica**, **Elimina** o **Cerca**.



Dopo: La tabella degli utenti viene aggiornata: viene aggiunta o modificata la riga scelta, in caso di eliminazione l'utente scompare, mentre con **Cerca** restano visualizzati solo gli utenti che soddisfano i dati inseriti (compreso lo stato **Blacklist**).

4.4.2 Inserimento utente

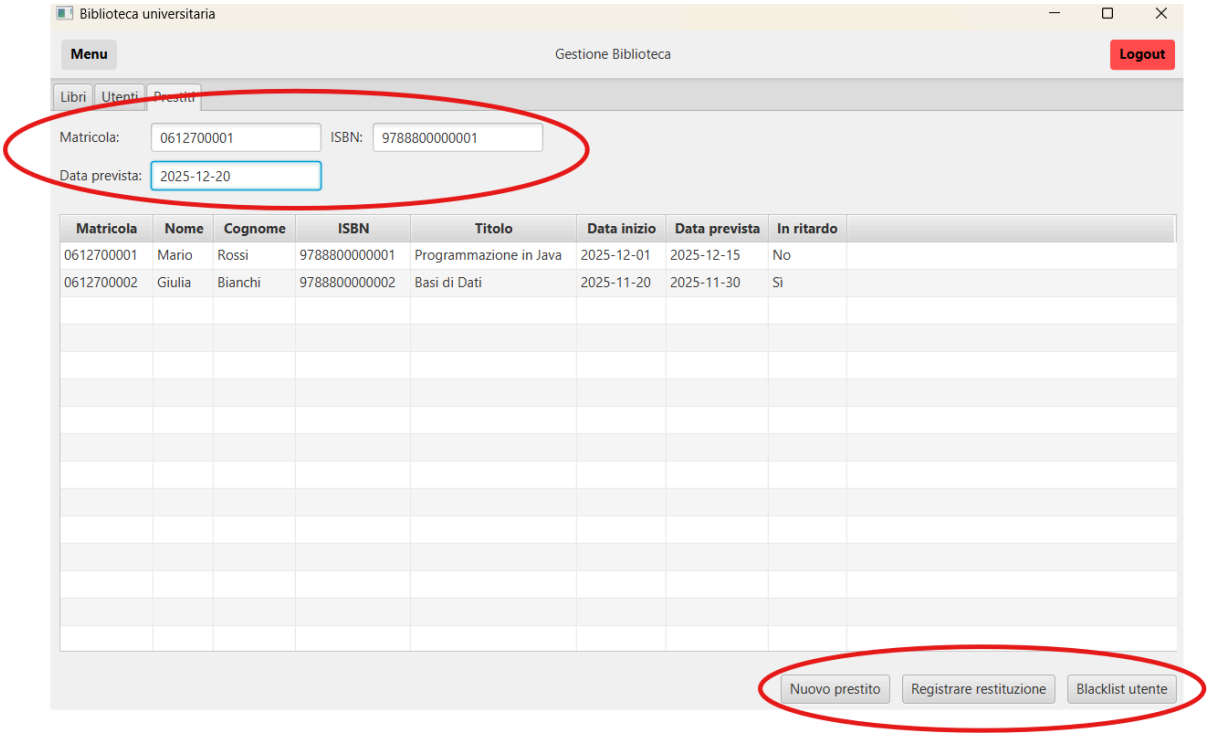
4.4.3 Modifica utente

4.4.4 Eliminazione utente

4.4.5 Ricerca utente

4.5.1 Panoramica scheda gestione prestiti

Prima: Nella scheda **Prestiti** il bibliotecario inserisce Matricola, ISBN e Data prevista, oppure seleziona un prestito dalla tabella, e preme **Nuovo prestito**, **Registrare restituzione** o **Blacklist utente**.



Dopo: La tabella dei prestiti viene aggiornata: viene aggiunta una nuova riga per il prestito registrato, contrassegnato con le date corrette e la colonna **In ritardo**, oppure il prestito selezionato viene chiuso/aggiornato o il relativo utente viene marcato in blacklist.

4.5.2 Inserimento di un nuovo prestito

4.5.3 Registrazione della restituzione

4.5.4 Gestione blacklist utente

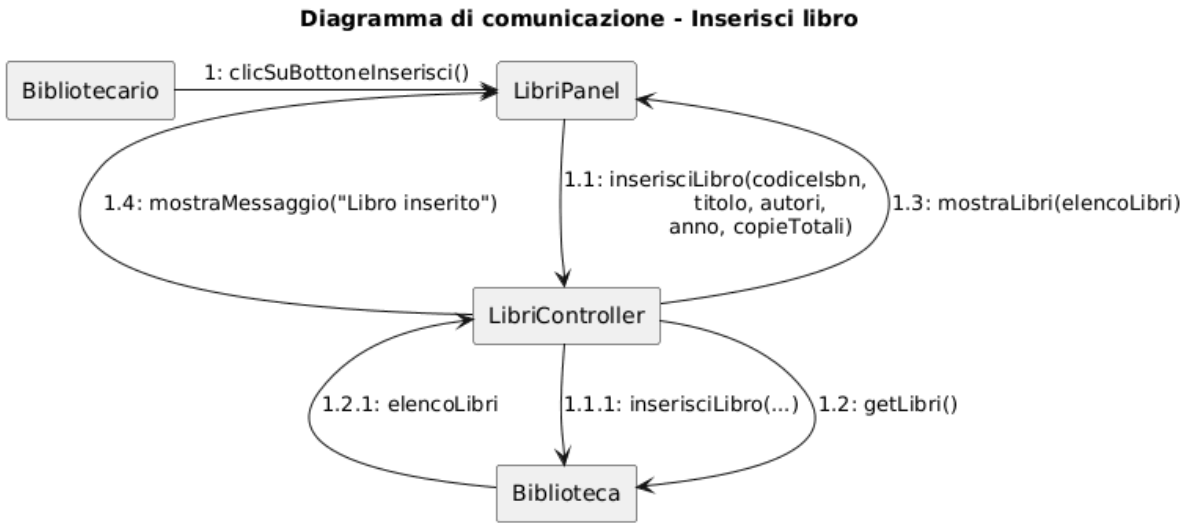
4.6 Barra superiore: Menu e Logout

Prima: In tutte le finestre di gestione è presente la barra superiore con il pulsante **Menu** e il pulsante **Logout**.

Dopo: Se il bibliotecario clicca su **Menu**, l'applicazione chiude la schermata corrente e ritorna al **Menu Biblioteca**(4.2).

Altrimenti: Se clicca su **Logout**, la sessione viene terminata e viene nuovamente mostrata la schermata di **login**(4.1).

Diagrammi di comunicazione: descrivono l'organizzazione strutturale degli oggetti che inviano e ricevono messaggi



Diagrammi delle attività

Diagrammi di macchina a stati

Diagrammi di distribuzione
NO