**Library Manager**

**Progetto Software**

**Autore:***Gabriele Vianello*

**Anno Accademico** 2024 – 2025

Politecnico di Milano

**Indice**

**Introduzione**

Il progetto prevede la realizzazione di un applicativo software, denominato *Library Manager*, che permetta di gestire una biblioteca. Per gestione si intende l’aggiunta, modifica, ricerca, eliminazione di libri ed utenti, così come per i prestiti dei libri agli utenti; le informazioni saranno gestite da un istanza locale di database SQL relazionale.

Poiché l’applicativo è realizzato esclusivamente in java potrà lavorare su differenti sistemi operativi/piattaforme[[1]](#footnote-1), gestendo anche più connessioni simultaneamente. L’interfaccia console sarà utilizzata per messaggi di debug e log (sia lato server che client), mentre la GUI sarà realizzata utilizzando JavaFX.

Un importante requisito che il sistema dovrà rispettare sarà quello della disconnessione sicura dove, in presenza di un malfunzionamento o mancata connettività di un client, il server continuerà a rispondere alle richieste dei client ancora attivi.

Saranno realizzati appositi test di funzionamento per i modelli e tutte quelle classi che saranno predisposte per la loro manipolazione.

Ulteriore documentazione, come JavaDoc, diagramma UML completo del sistema e percentuali di coverage dei test si trovano nell’apposita cartella all’interno del progetto.

Immagine che contiene design

Descrizione generata automaticamente

**Library Manager**

**Architettura generale**

Il sistema si compone di due eseguibili, uno denominato *Server*, sempre attivo e pronto ad accettare nuove connessioni da parte di applicativi client. Il secondo ed ultimo eseguibile è *LibraryApplication* che avvia l’applicazione client basata su JavaFX.

Per garantire la modularità del sistema sono stati adottati il DAO Pattern per l’accesso al database e l’MVC per un’interazione ottimale ed efficiente sia con utente finale che con il server.

Le entità previste ed implementate sono *Books*, *Lends* e *Customers*; contengono, oltre ai necessari attributi, metodi per l’elaborazione dei dati.

Ogni connessione server client è indipendente e viene gestita lato server come un thread che riceve e risponde alle richieste del client assegnatogli. Le richieste/risposte sfrutteranno un apposito sistema di messaggistica che permette di scambiare oggetti od altre informazioni.

Le operazioni previste dovranno essere richieste dal client al server, il tutto passando per delle classi intermedie necessarie all’elaborazione dei dati, incapsulando la richiesta in un messaggio che sarà serializzato e spedito al server. Allo stesso modo, le risposte in ingresso ripercorrono la stessa strada ma al contrario; in quest’ultimo caso, a rimanere sarà l’oggetto richiesto.

Immagine che contiene testo, diagramma, Piano, schematico

Descrizione generata automaticamente

**Comunicazione**

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamentePer garantire una comunicazione efficace tra applicazione client e server, è stato implementato un sistema di messaggistica basato su oggetti serializzabili e caratterizzati – essenzialmente – da un’operazione ed un contenuto.

Quindi i modelli dei libri, utenti prestiti (o qualunque altra struttura dati) possono essere facilmente trasmessi attraverso *ObjectInputStream e ObjectOutputStream*.

Saranno i metodi *send(Object)* e *receive()* a rappresentare rispettivamente output ed input per server e client.

Dall’UML si può osservare come tutti i messaggi estendano la classe astratta *Message*; la distinzione nei differenti tipi di messaggio avviene per operazione e soprattutto per tipo di contenuto che si vuole trasmettere.

Ad esempio, *BookMessage* avrà come *message* (attributo che contiene l’oggetto da inviare) un oggetto di tipo *Book*, mentre *BooksListMessage* sarà utilizzato per inviare ArrayList<Book>(). Inoltre, per costruire dinamicamente i messaggi viene adottato il *Factory* design pattern.

Immagine che contiene testo, schermata, linea, Carattere

Descrizione generata automaticamente

**MVC (Model-View-Controller)**

L'MVC è un pattern architetturale utilizzato principalmente nelle applicazioni con interfaccia utente (UI). Divide l'applicazione in tre componenti principali:

1. **Model (Modello):** rappresenta la logica di business e i dati dell'applicazione. È responsabile della gestione dello stato dell'applicazione dove gli oggetti rappresentano entità, che in questo caso possono essere *libri*, *prestiti* o *utenti*.
2. **View (Vista):** è la parte che gestisce l'interfaccia utente, mostra i dati all'utente e invia i comandi dell'utente al *Controller*; un esempio di *View* è la finestra JavaFX.
3. **Controller (Controllore):** gestisce l'interazione tra il Model e la View; riceve input dall'utente tramite la View, elabora i dati (eventualmente con l'aiuto del Model) e aggiorna la View. Concettualmente, lavora come un intermediario, tra la view ed il modello ottenuto a partire dai dati contenuti nel Database.

In questo modo – oltre a garantire la separazione della logica di business da quella di visualizzazione – facilita il mantenimento e la testabilità del codice, oltre ad una maggiore flessibilità in caso di aggiornamenti futuri.

**DAO (Data Access Object)**

Il DAO è un pattern che si concentra sull'accesso ai dati. È una classe o un insieme di classi che fornisce un'interfaccia astratta per interagire con un database o un'altra fonte di dati.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteLa gestione della vera e propria connessione è affidata alla classe *DatabaseController*: contiene URL e credenziali per accedere alla base di dati MySQL con JDBC, oltre ad appositi metodi per instaurare e chiudere la connessione.

**Componenti principali del DAO:**

1. **Interfaccia DAO:** definisce i metodi per operazioni CRUD (Create, Read, Update, Delete). Un esempio di interfaccia è *BookDAOInterface* che conterrà le dichiarazioni dei metodi che saranno utilizzati per l’accesso/estrazione dei dati dal DB.
2. **Implementazione DAO:** contiene l'implementazione concreta dell'interfaccia, interagendo con il database grazie alle query configurate in ciascun metodo[[2]](#footnote-2). Nasconde la complessità della gestione del database, come query e ritorno dei risultati della ricerca.

L’adozione ed implementazione del DAO favorisce la riutilizzabilità del codice; dona anche una maggiore modularità al sistema, oltre a garantire un certo grado di facilità nel testing.

Di seguito saranno introdotte le classi dei modelli e relative implementazioni DAO; si noti che per ogni modello è presente un attributo *id* intero: rappresenta l’identificativo utilizzato per individuare un certo dato (libro, prestito o cliente) nella base di dati.

**Gestione dei libri (Books):**

* I libri saranno rappresentati dagli oggetti della classe *Book*.

Immagine che contiene testo, schermata, numero, software

Descrizione generata automaticamente

* Interfaccia e relativa implementazione DAO

Immagine che contiene testo, schermata, numero, Carattere

Descrizione generata automaticamente

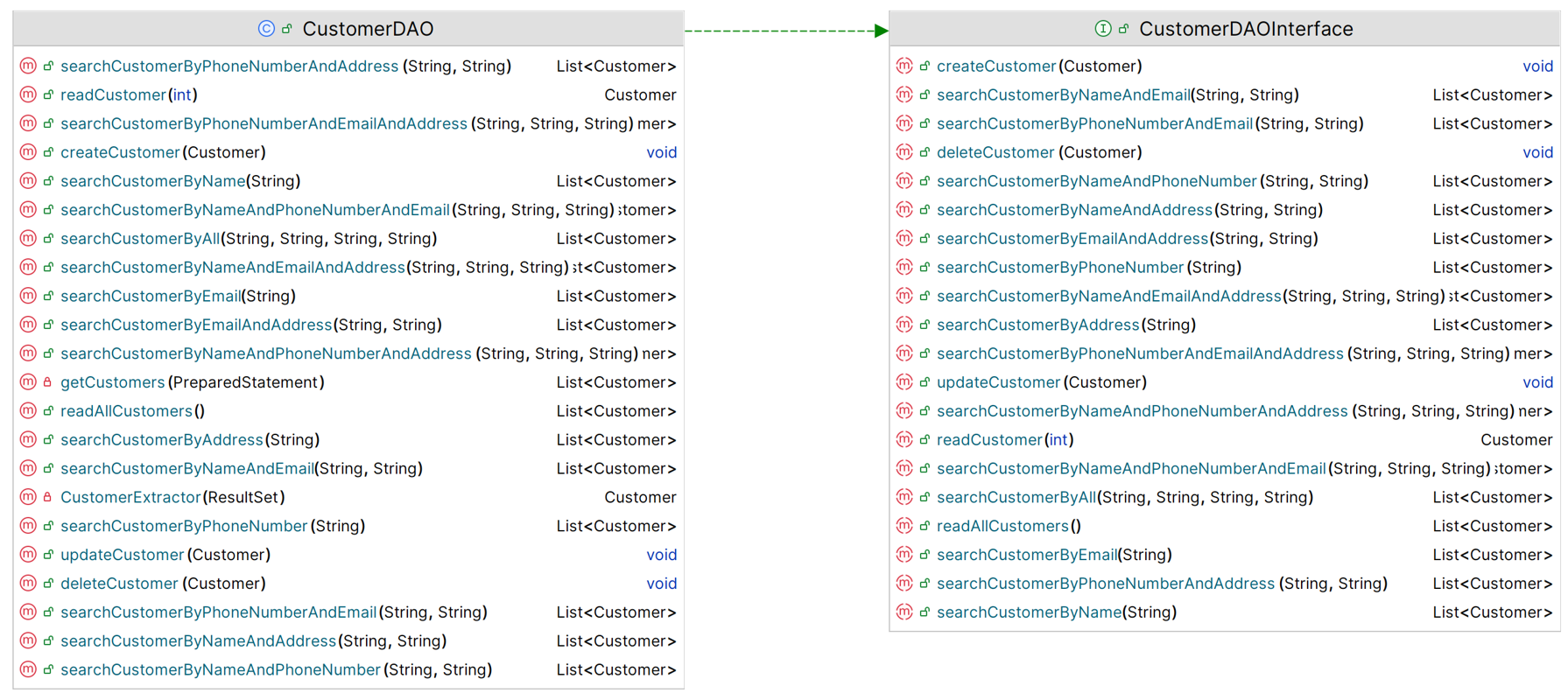
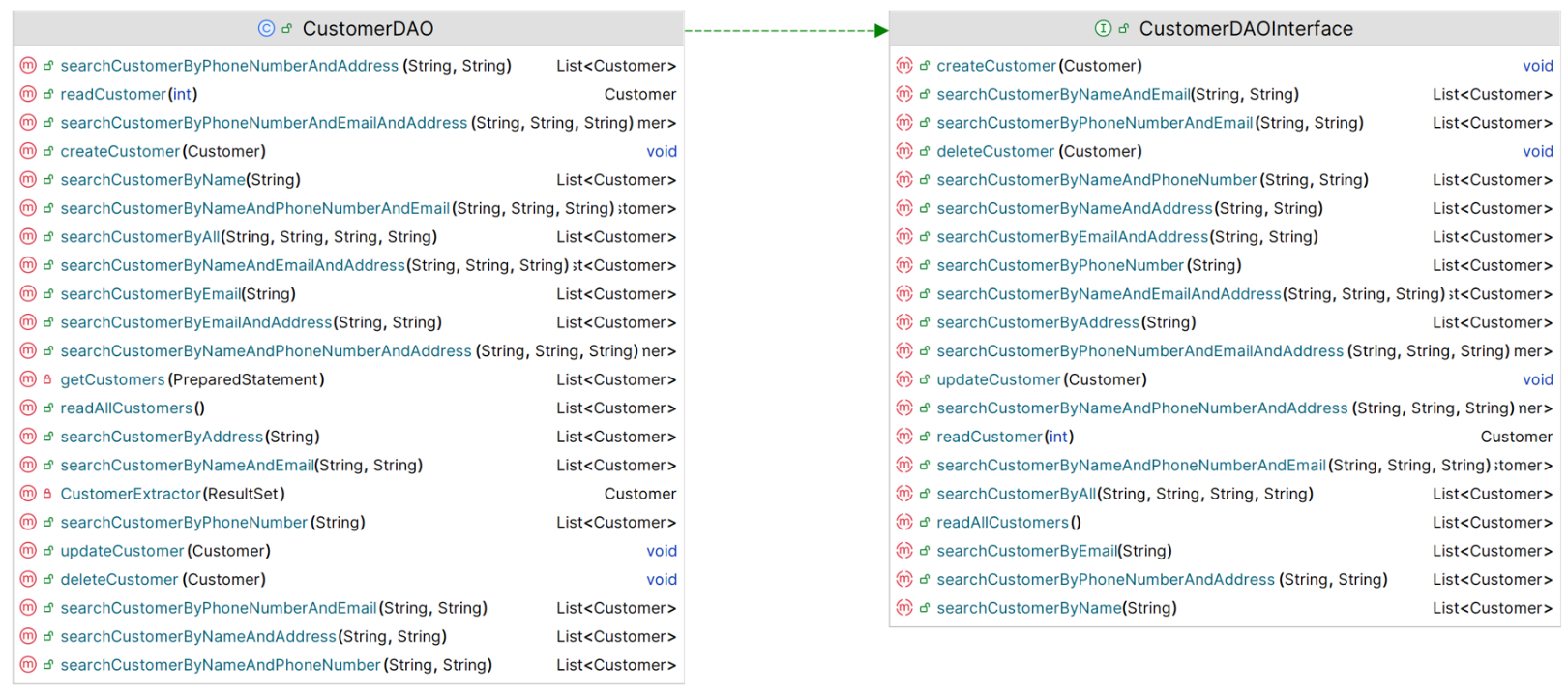
**Modulo di gestione clienti (Customers)**

* I clienti saranno rappresentati dalla classe *Customers*

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamente

* Di seguito l’interfaccia e l’implementazione del *CustomerDAO*.



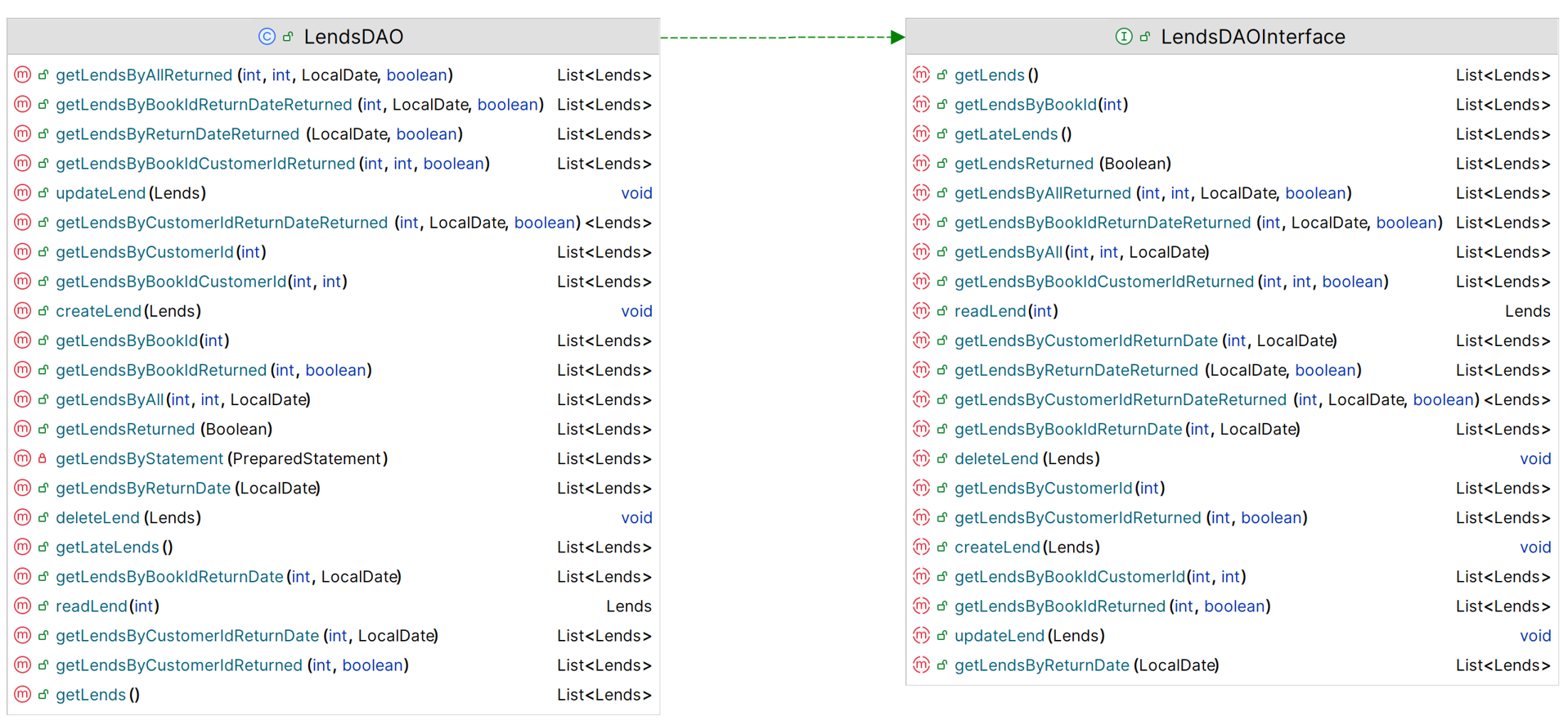
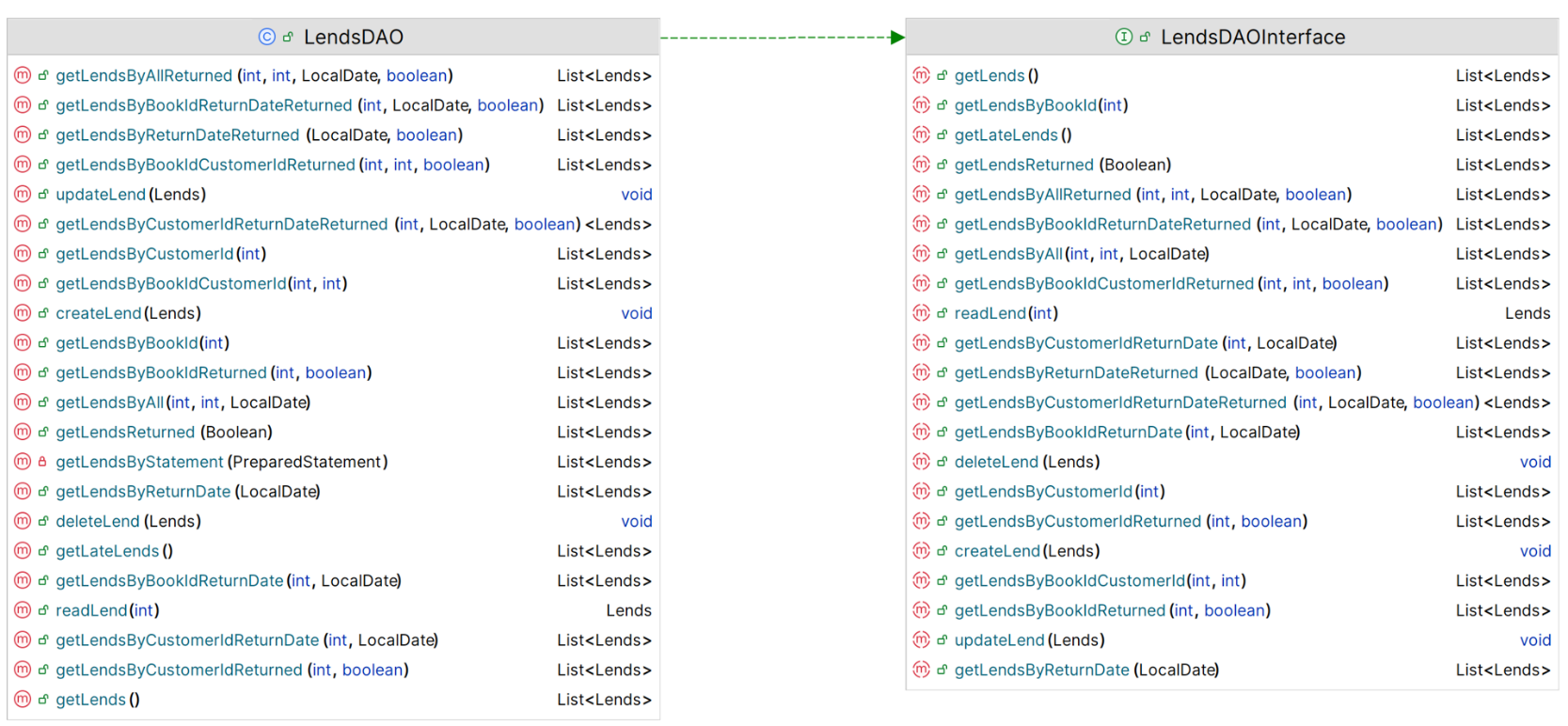
**Modulo di gestione prestiti (Lends)**

* Descrizione delle entità e delle operazioni CRUD sui prestiti di libri

Immagine che contiene testo, schermata, numero, software

Descrizione generata automaticamente

* Implementazione del DAO per l'interfaccia con il database



**Server**

L’applicativo server svolge due operazioni fondamentali, all’avvio apre il *socket* e si prepara ad accettare nuove connessioni dove assegnerà ad ogni client collegato un proprio *ClientHandler*. Questa particolare classe implementa *Runnable* e gestirà il client associato finché la connessione non sarà interrotta; tale gestione prevede invio e ricezione dei messaggi, di seguito l’UML della classe.

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamenteAl costruttore viene passato il *socket* su cui è attivo il client (che a sua volta viene passato con *super(…)* al costruttore della *NetworkInterface*).

Il metodo *run()* contiene il codice necessario gestione dei messaggi in arrivo dal client. A seconda del tipo di richiesta richiama uno dei metodi contrassegnati con *handle…Operation* a seconda del messaggio inviato*.*

Ciascuno dei metodi che gestiscono le operazioni provvederà ad inviare un opportuna risposta sfruttando sia la funzione *send(Object)* della *NetworkInterface* che il *createMessage(…)* della classe *MessageFactory*.

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

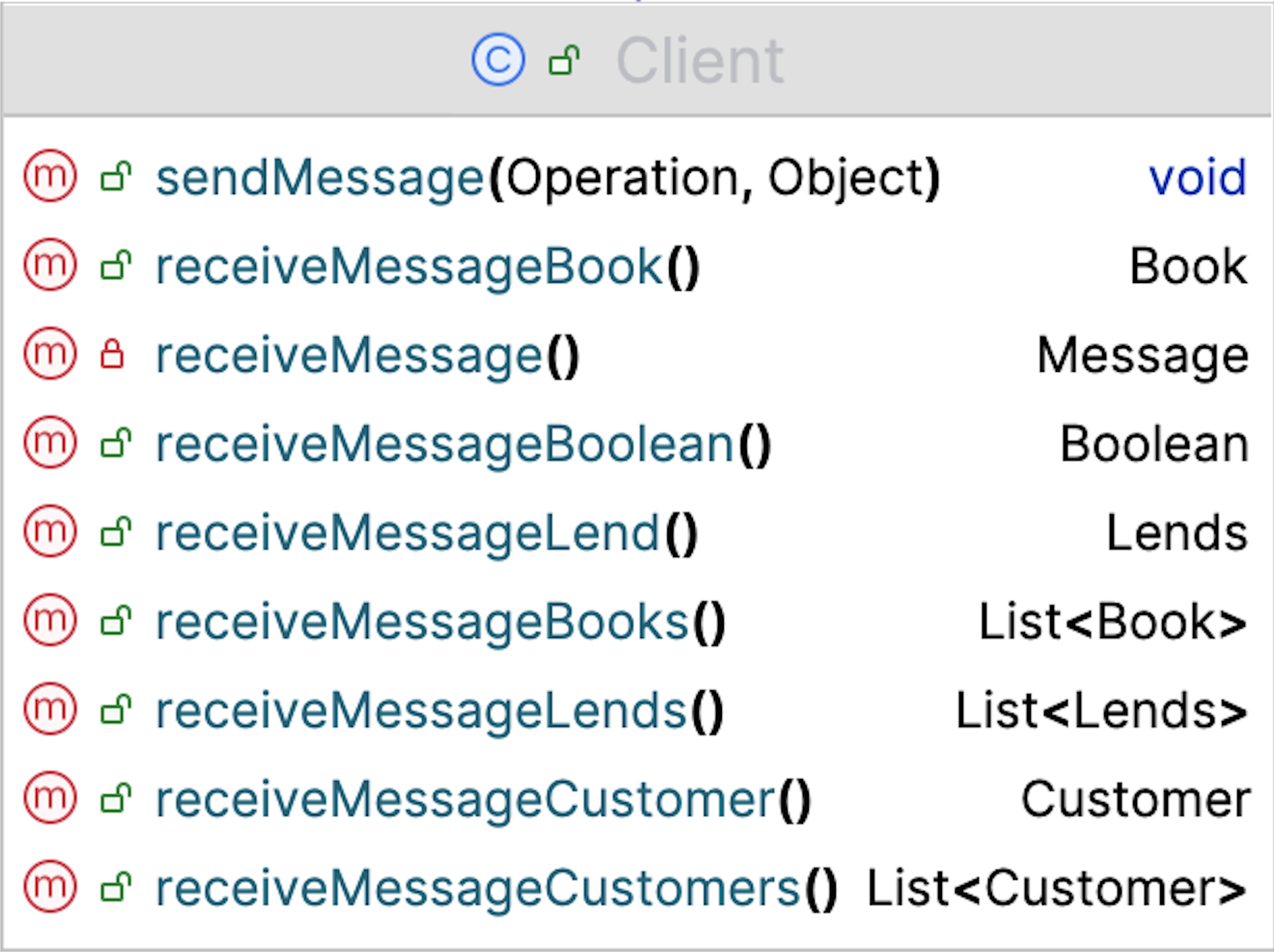
Come anticipato nelle righe precedenti, la classe *NetworkInterface* implementa tutti quei metodi utilizzati per gestire l’invio e la ricezione di messaggi (senza trattarne la tipologia, solo come oggetti serializzabili), così come la disconnessione sicura.

**Disconnessione sicura:** data la natura dell’applicazione è necessario che il server rimanga attivo anche in presenza di disconnessione dal client; il primo passo è stato gestire ciascuna connessione come thread indipendente ed infine è stato aggiunto un apposito metodo. Una volta sollevata l’eccezione relativa all’errore in ricezione (sul server) allora viene chiamato *safeDisconnection()* che provvede a chiudere la comunicazione con il client ed interrompere il thread. In questo modo il sistema continuerà a funzionare senza disconnettere gli altri utenti.

**LibraryApplication**

L’applicativo client può essere concettualmente suddiviso in 3 livelli di cui uno deputato alla connessione/ricezione/invio, uno stadio di elaborazione intermedia , infine si trova il livello di presentazione

**Client:** classe che rappresenta il livello di connessione, instaura la connessione con il server e gestisce invio e ricezione dei messaggi (sfruttando i metodi di *NetworkInterface[[3]](#footnote-3)*).



**ClientController:** classe che elabora i dati/messaggi ricevuti dal client, così che possano essere utilizzati dal controller della view. Inoltre, prepara i messaggi da spedire al server, come le richieste di ricerca nel database.

Immagine che contiene testo, schermata, software, numero

Descrizione generata automaticamenteAlcuni metodi sono usati per elaborazioni più “pesanti”. Né è un esempio quanto accade con quelli utilizzati per la sezione relativa alle statistiche sui prestiti del sistema (indicate nell’UML a sinistra come *calculateCustomerLendingStats()*).

**ViewController:** continuare da qui.s

**Interfaccia grafica**

GUI, spiegazione.

**Tecnologie utilizzate**

Elenco delle principali tecnologie e librerie utilizzate nell'applicativo:

* Java
* JavaFX
* Database (es. MySQL, PostgreSQL)

**Macchina e Ambiente di Sviluppo:** il software è stato sviluppato interamente su macchina Windows 10 Pro con istanza Server di MySql8.0 e utilizzando IntelliJ IDEA come ambiente di sviluppo. Le query SQL sono state scritte su MySql WorkBench.

**Installazione e configurazione**

Istruzioni per l'installazione e la configurazione dell'applicativo, inclusi i requisiti di sistema e le dipendenze.

**Considerazioni finali**

Eventuali note aggiuntive o considerazioni sull'applicativo.

1. A patto che abbiano una Java Virtual Machine installata. [↑](#footnote-ref-1)
2. Che saranno poi inviate al DB attraverso la connessione ottenuta dal DatabaseController. [↑](#footnote-ref-2)
3. In maniera duale rispetto a quanto accade nella classe *ClientHandler*; si noti che entrambi estendono *NetworkInterface*. In questo modo si dona al sistema una maggiore separazione delle responsabilità. [↑](#footnote-ref-3)