DOCUMENTAZIONE LEGATA AL DESIGN DI DETTAGLIO

Nel file .pdf allegato è presente il diagramma delle classi. Quest'ultimo ci permette di individuare l'ossatura di un sistema software, poiché mette in evidenza le classi, i loro attributi (cioè i loro dati), i loro metodi (cioè le azioni/operazioni) e le relazioni tra di esse.

Nel nostro caso, abbiamo effettuato una serie di scelte per garantire una progettazione affidabile e duratura, con buoni livelli di coesione ed accoppiamento tra le classi.

COESIONE: permette di misurare quanto fortemente le parti che sono incluse nello stesso "modulo" lavorano insieme per raggiungere un unico scopo.

1) Per quanto riguarda le entità del nostro progetto, vi è una coesione funzionale molto alta, poiché tutti gli attributi ed i metodi permettono di rappresentare e gestire un singolo concetto di dominio (come un libro, uno studente o un prestito), senza che intervenga la GUI.

Dato che non ci sono sequenze ferree di passi da eseguire in un ordine preciso all'interno della stessa classe, la coesione sequenziale è molto meno evidente; né si vede coesione temporale, dato che le funzioni dei vari moduli sono indipendenti tra di loro).

Invece per quanto riguarda la coesione logica, ci potrebbero essere degli "esempi", poiché le classi Libro e Studente operano in maniera quasi simile per quanto concerne la raccolta dei dati. Inoltre, non vi è sicuramente coesione coincidentale, poiché le entità ed i loro metodi non sono assolutamente "scollegate" tra di loro.

2) Per quanto riguarda le collezioni, Catalogo ed Elenco (entrambi rispettivamente insiemi di Libro e Studente) presentano un'alta coesione funzionale, questo perché tutte le operazioni ruotano attorno alla gestione della collezione specifica. Vi è sicuramente coesione comunicazionale, per via della stessa struttura dati (TreeSet) che condividono entrambe le collezioni.

E dato che non appaiono gruppi di metodi/azioni prettamente generiche, escludiamo coesione coincidenziale e logica, oltre che temporale (per lo stesso discorso fatto per le entità).

3) Per quanto riguarda i controller, Menu_BibliotecaController, GestioneLibriController, GestioneStudentiController, Interfaccia_nuovoPrestitoController, VisualizzaStudente_viewController, vi è alta coesione funzionale, in parte legata alla schermata che controllano, in parte legata i dati su cui operano).

Vi è coesione anche per comunicazione, questo perché i metodi operano sugli stessi componenti GUI e spesso anche sugli stessi oggetti di dominio.

In alcuni controller vi è inoltre coesione sequenziale poiché certi metodi devono essere richiamati in un determinato ordine cronologico (ad es. in Interfaccia_nuovoPrestitoController facciamo: inserisci dati -> valida dati -> crea prestito -> aggiorna -> mostra notifica).

Non vi è una marcata coesione temporale e sicuramente non c'è coesione coincidentale, poiché le entità ed i loro metodi non sono assolutamente "scollegate" tra di loro.

ACCOPPIAMENTO: permette di misurare quanto un "modulo" dipende da un altro.

1) Per quanto riguarda le entità del nostro progetto, tra di loro sussiste un buon livello di accoppiamento: quello suoi dati, il livello più basso possibile (senza ovviamente considerare la possibile perfezione dovuta all'assenza totale di accoppiamento, una grande utopia). Questo perché la classi si passano parametri semplici e riferimenti tipati a oggetti di dominio e sfruttano i loro metodi pubblici, garantendo quindi che non vi sia accoppiamento per contenuti.

Non segnaliamo esempi di accoppiamento per contenuti (il peggiore), questo perché nel nostro progetto nessuna classe manipola direttamente le variabili private di un'altra; né segnaliamo esempi di accoppiamento per aree comuni, questo perché non riscontriamo l'uso di variabili globali o accessi ad un parametro da più classi contemporaneamente.

2) Per quanto riguarda i controller e la GUI, i primi sono accoppiati tramite dati alle entità e alle collezioni, seguendo l'esatto accoppiamento previsto in una architettura MVC / Entity-Control-Boundary, garantendo di ridurre il rischio che eventuali modifiche GUI "rompano" le entità.

Vi è tuttavia una piccola parte di accoppiamento per controllo, poiché i controller decidono il flusso dei casi d'uso e quindi possono invocare in sequenza delle operazioni su più entità (ad es. verifiche su Studente, l'aggiornamento su Prestito, notifica per una modifica, etc.) -> determinando un controllo più "diretto" da parte della GUI.

Per quanto riguarda l'accoppiamento per timbro, esso potrebbe "apparire", più che a livello di class diagram, a livello delle firme dei metodi (poichè nel passaggio delle strutture dati potrebbero capitare anche informazioni non necessarie).

PRINCIPI DI BUONA PROGETTAZIONE:

1) Le classi di dominio si occupano solo del modello dati e di operazioni legate al dominio biblioteca, senza mischiare GUI e file. Seguono quindi l'applicazione del "Single Responsability Principle": in questo modo le eventuali modifiche toccano solo una classe/interfaccia del programma, e non un intero modulo di esso.

2) Riteniamo che è stato seguito anche l'"Open-Closed Principle", poiché I nostri moduli, le classi e le funzioni a loro associate risultano essere più propense all'estensione piuttosto che alla modifica.

3) Abbiamo sicuramente fatto affidamento all' "Interface Segregation Principle", poiché abbiamo preferito un'implementazione con più interfacce, più specifiche e compatte, rispetto ad una implementazione più rigorosa ma meno efficace, permettendo così di ridurre l'accoppiamento tra componenti e facilitando la sostituzione di parti del sistema senza impatti sulle altre.

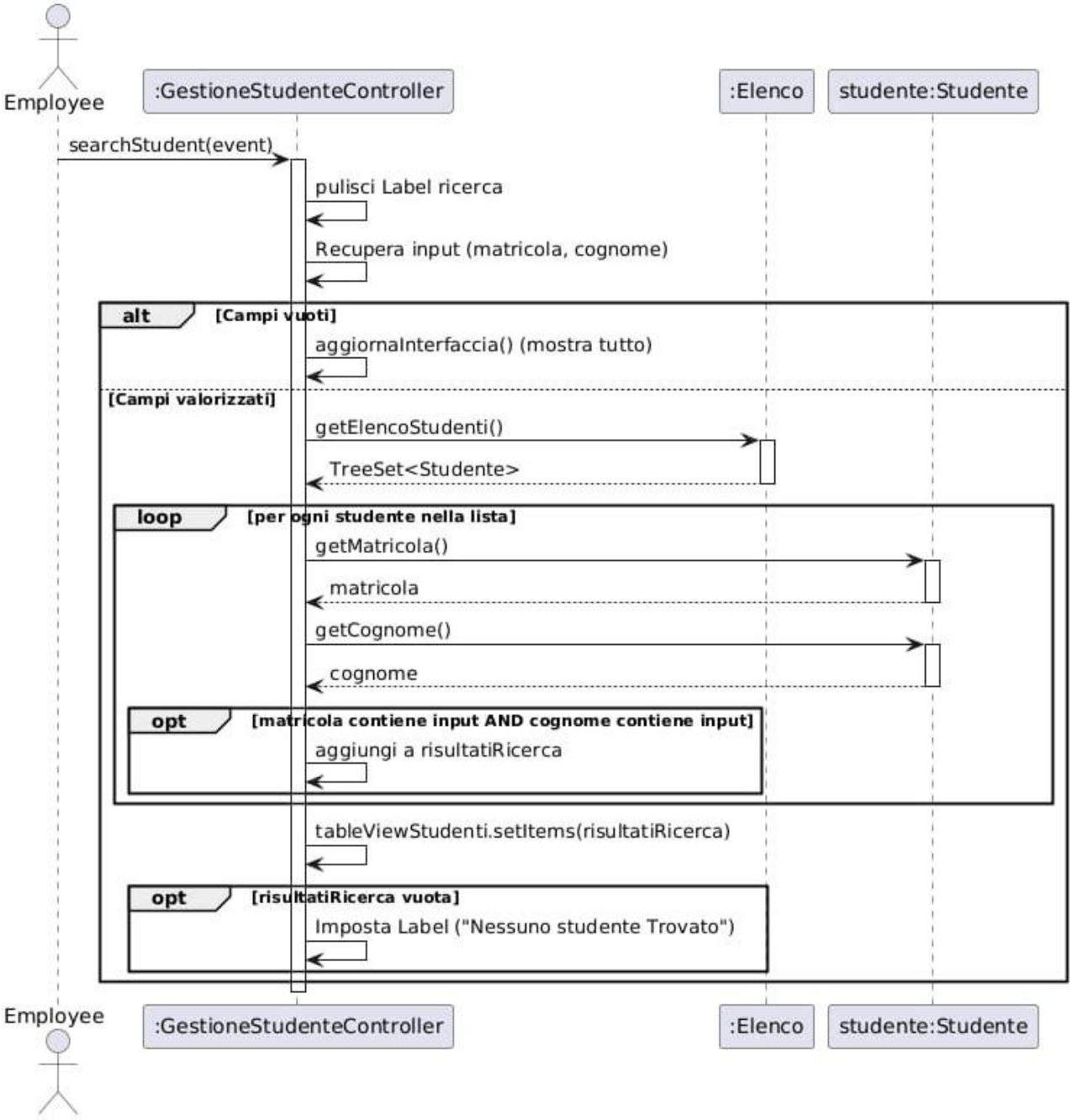
4) Ovviamente abbiamo seguito anche i principi di correttezza e robustezza: quindi il nostro sistema rispetto formalmente tutte le specifiche proposte dal progetto ed è in grado di gestire le eccezioni e di funzionare correttamente anche in presenza di input anomali.

5) Inoltre, tra di noi abbiamo seguito la regola del boy scout: ogni qualvolta uno di noi modificava il codice, prestava la massima attenzione per non comprometterne la qualità (abbiamo evitato quindi che eventuali errori si fossero protratti nel tempo, nel corso dei commit e delle push).

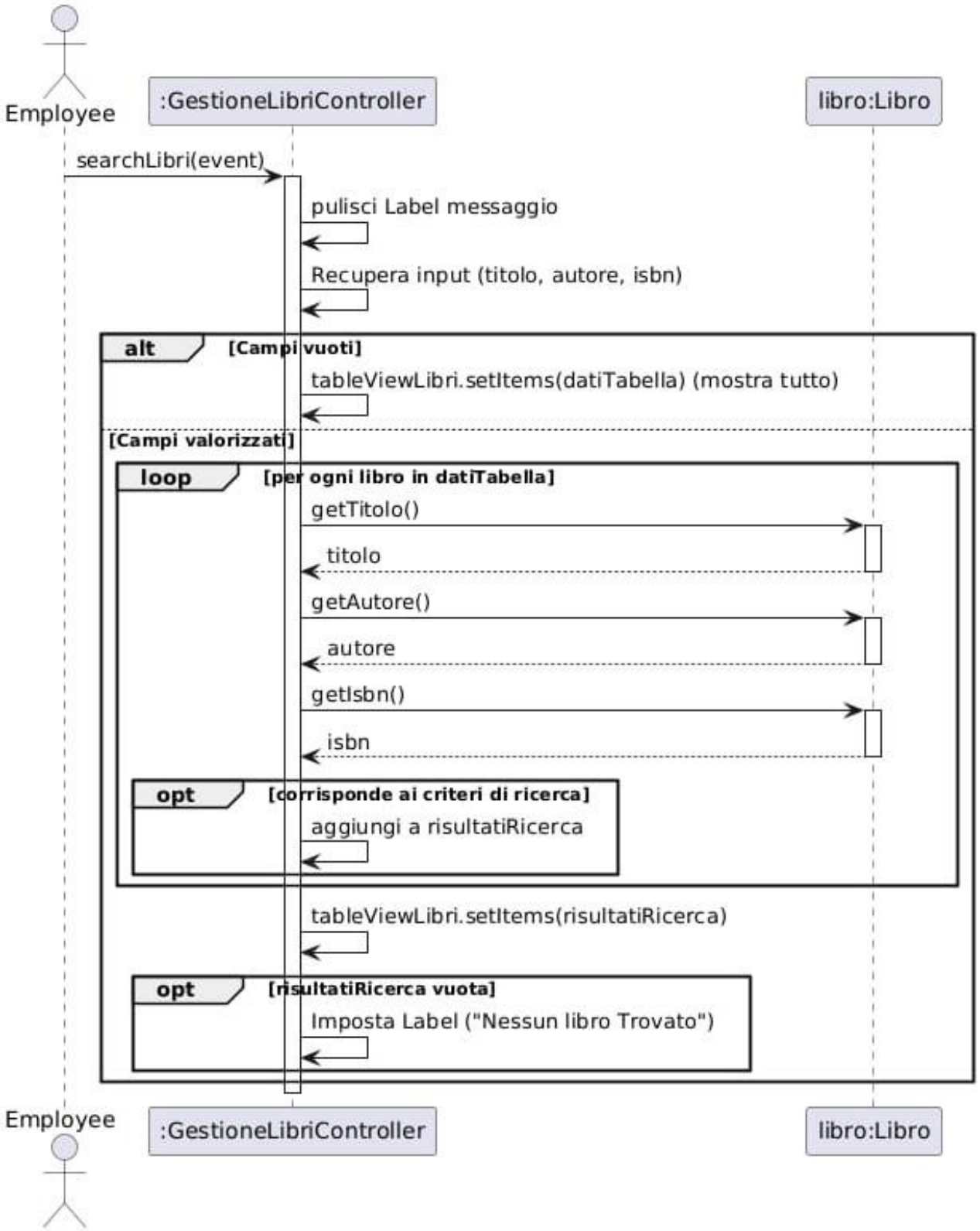
DIAGRAMMI DI SEQUENZA: questo tipo di diagramma metti in risalto il comportamento (di un sottoinsieme, di una sottoclasse) di un sistema software, in un singolo scenario. Cioè mostra quali oggetti sono coinvolti e quali messaggi vengono scambiati tra questi oggetti.

Qui il tempo segue l'ordine cronologico e scorre dall'alto verso il basso mentre gli oggetti sono elencati da sinistra verso destra e i messaggi sono rappresentati tramite frecce che indicano la direzione in cui viaggia il messaggio stesso.

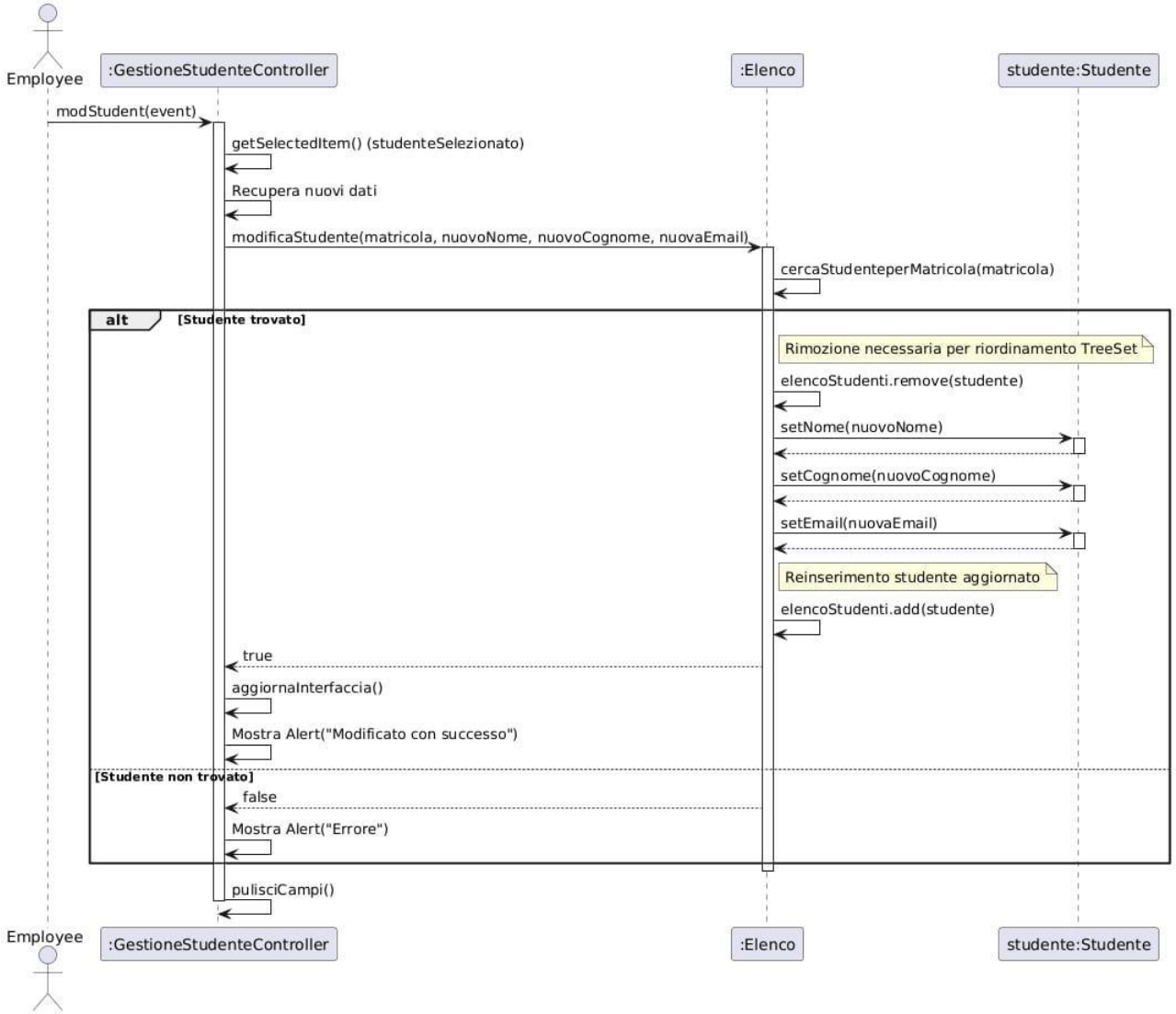
1. Cerca Studente:



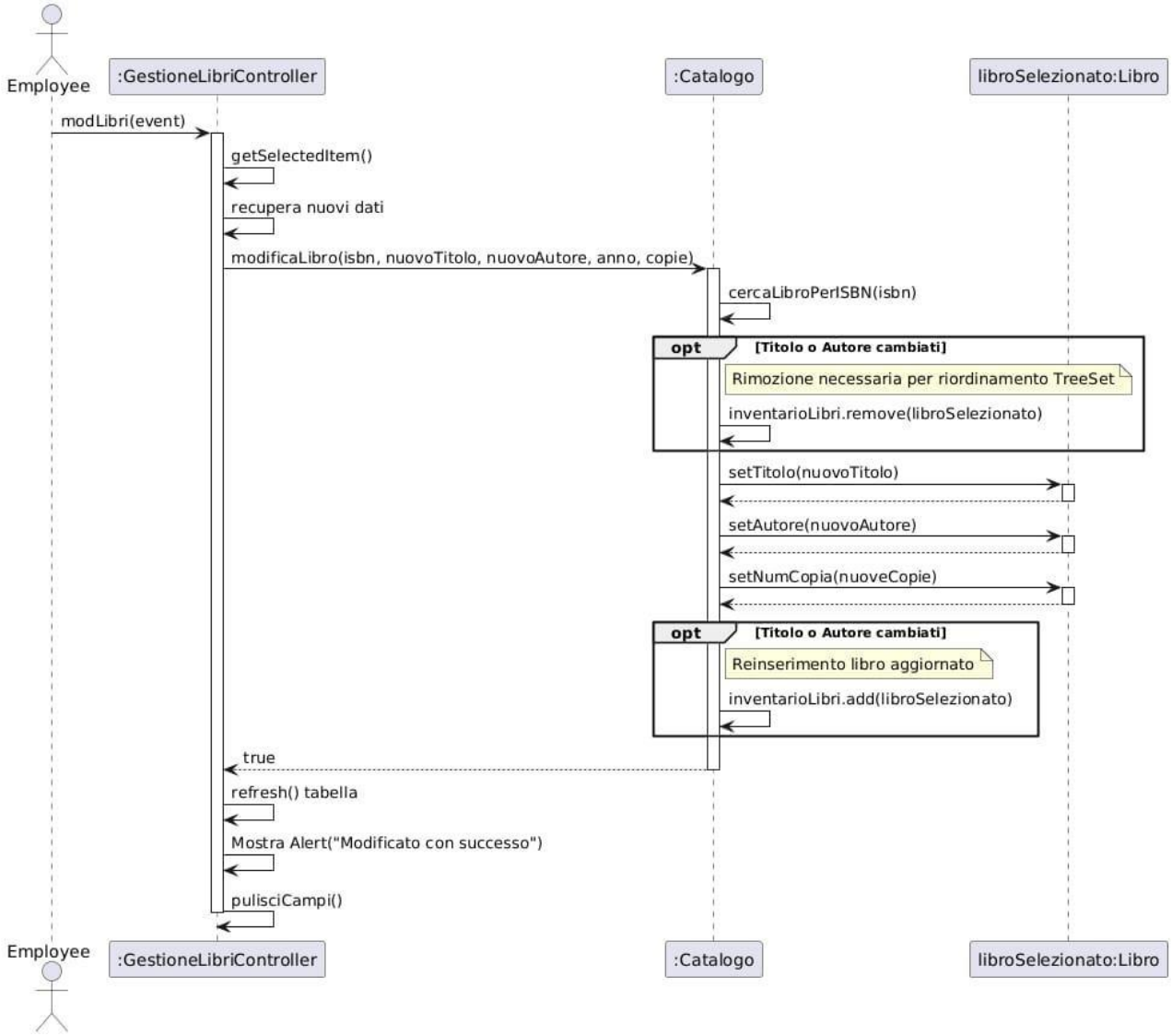
2. Cerca Libro:



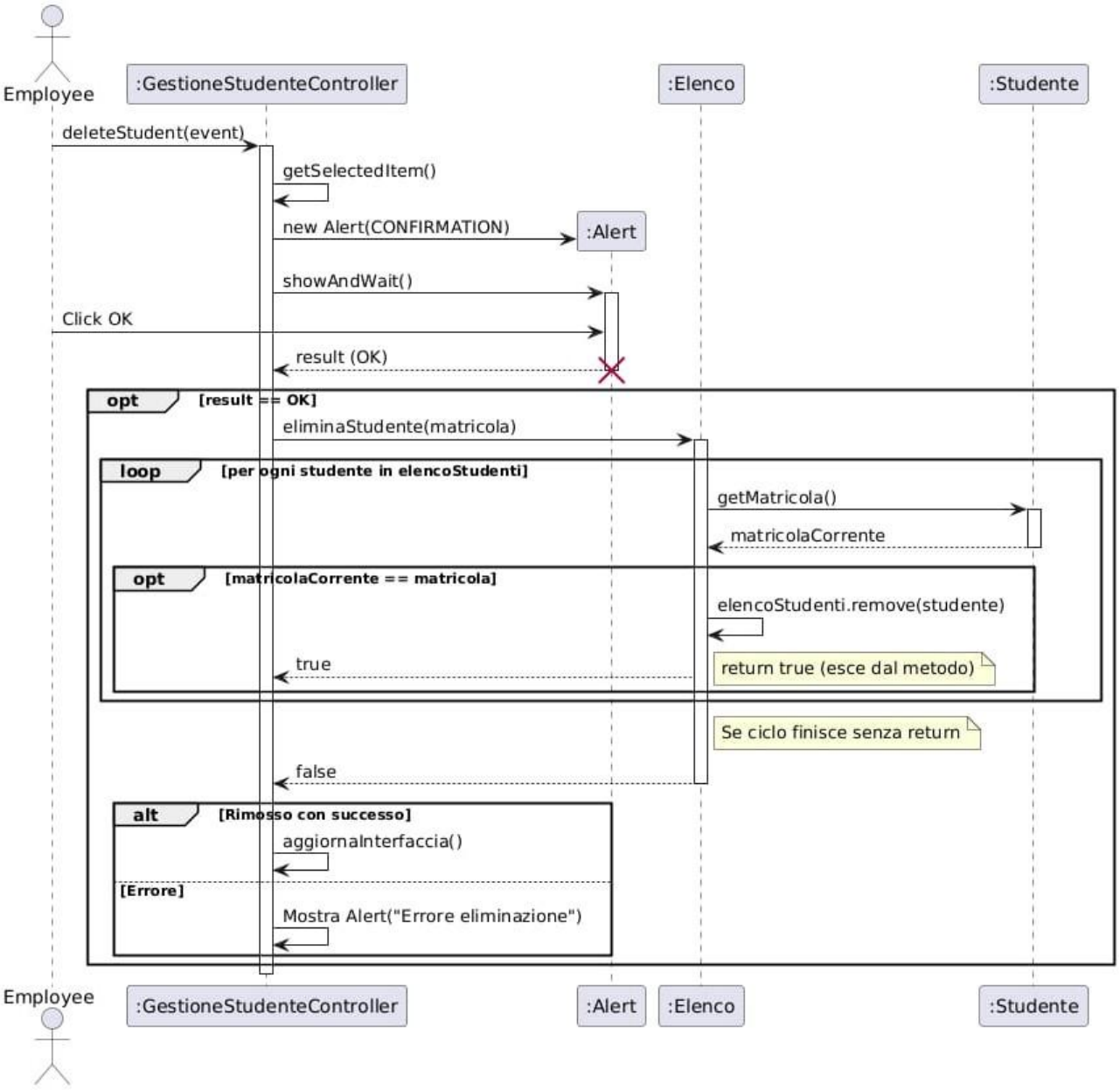
3. Modifica Studente:



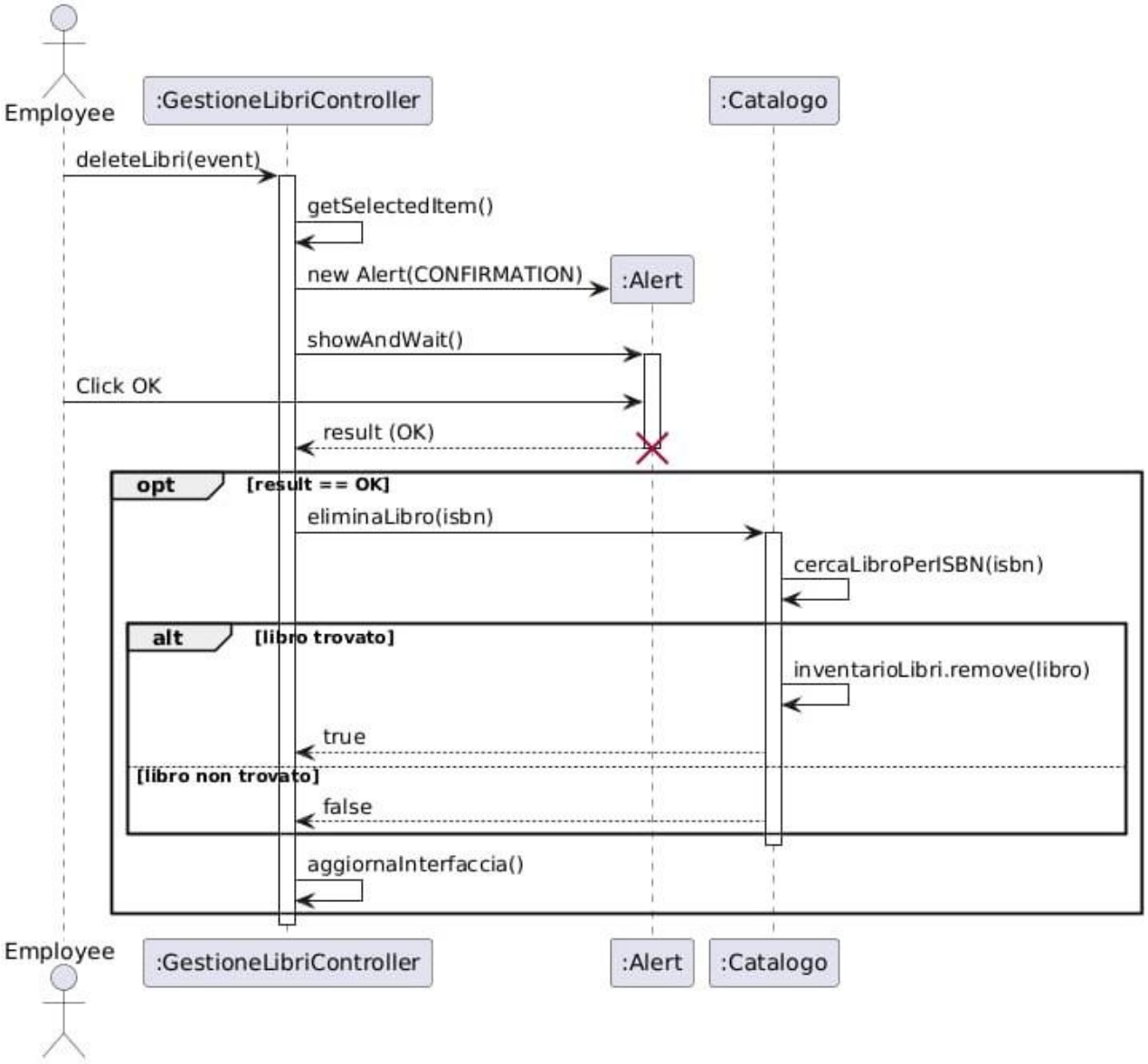
4. Modifica Libro:



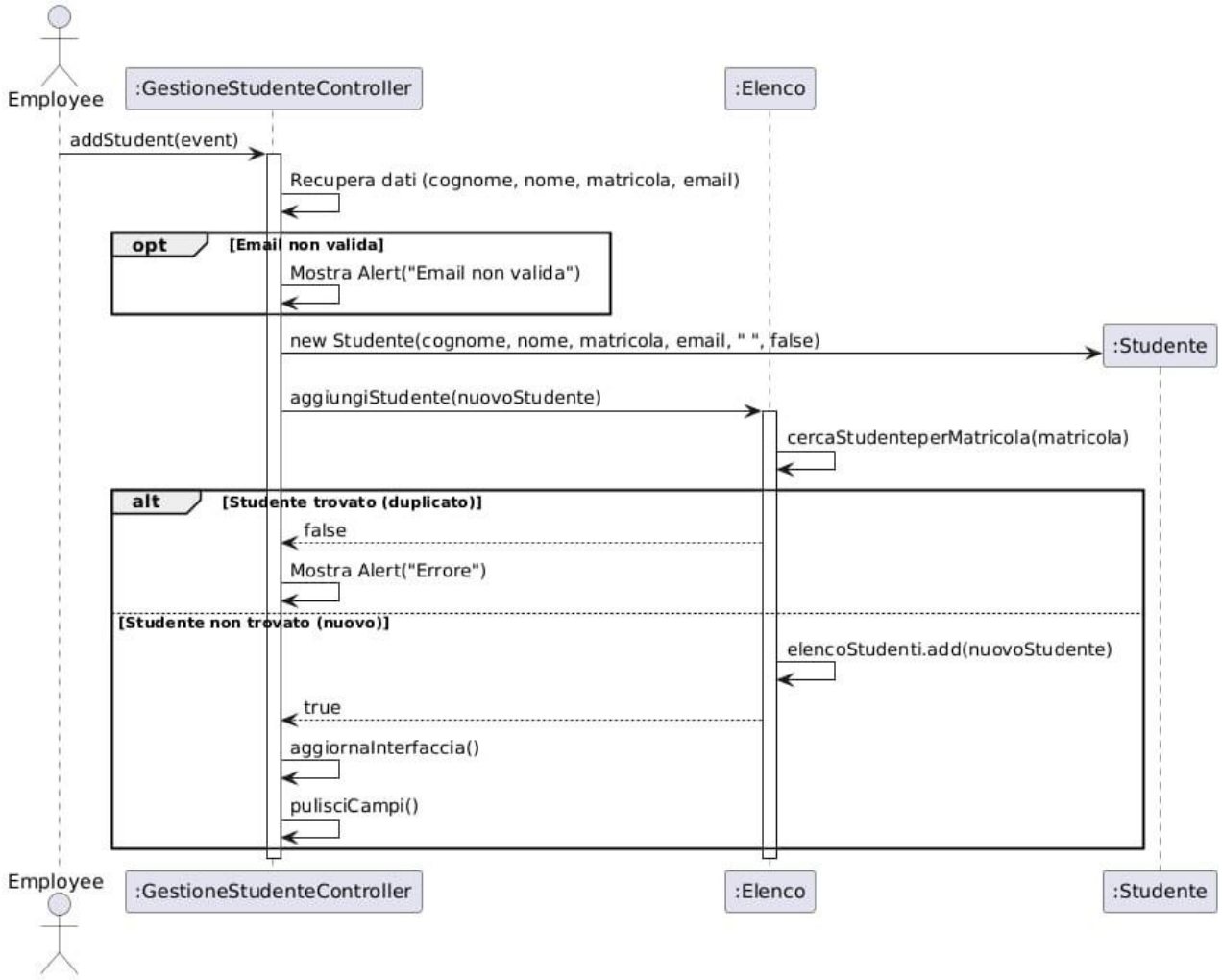
5. Elimina Studente:



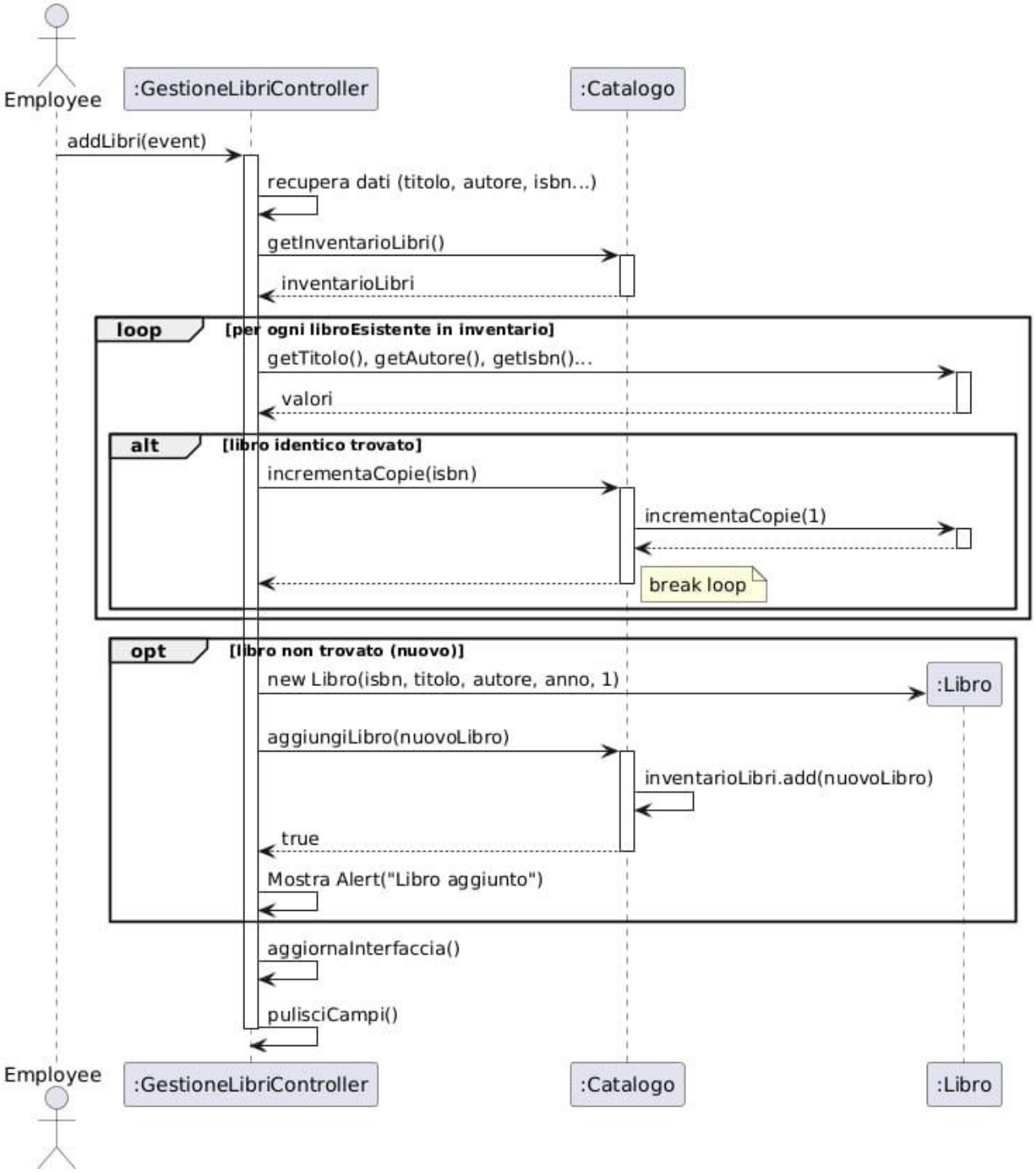
6. Elimina Libro:



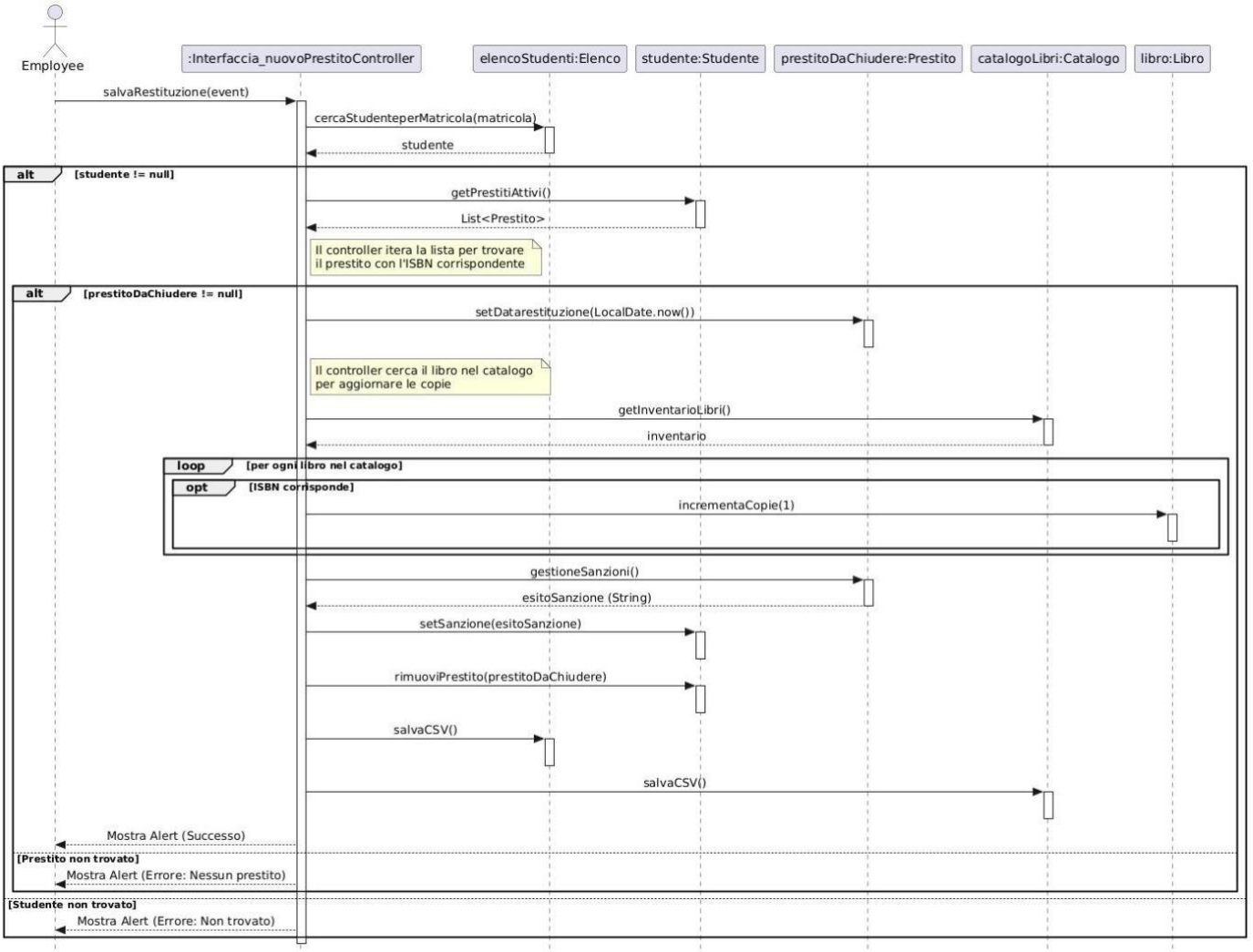
7. Aggiungi Studente:



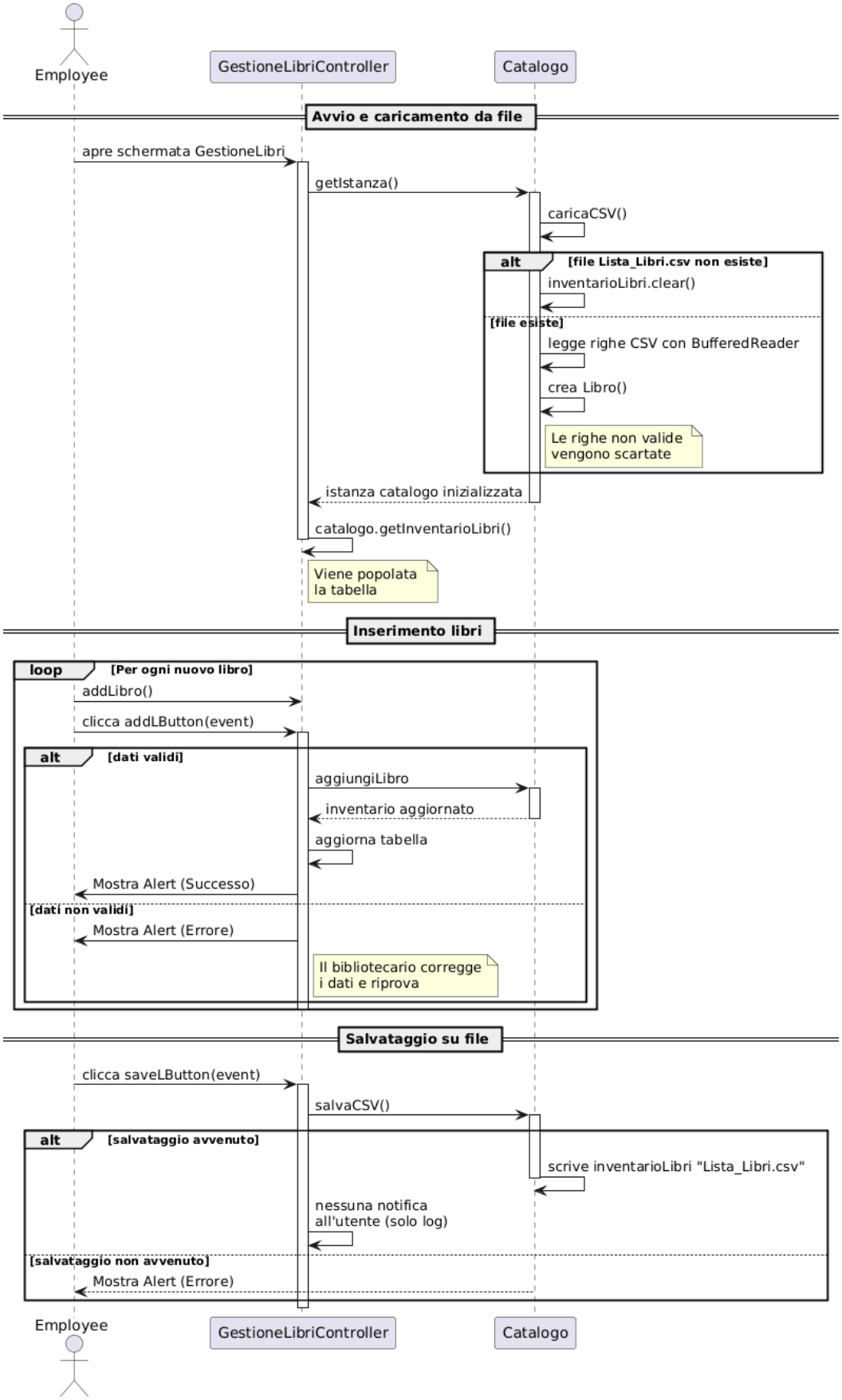
8. Aggiungi Libro:



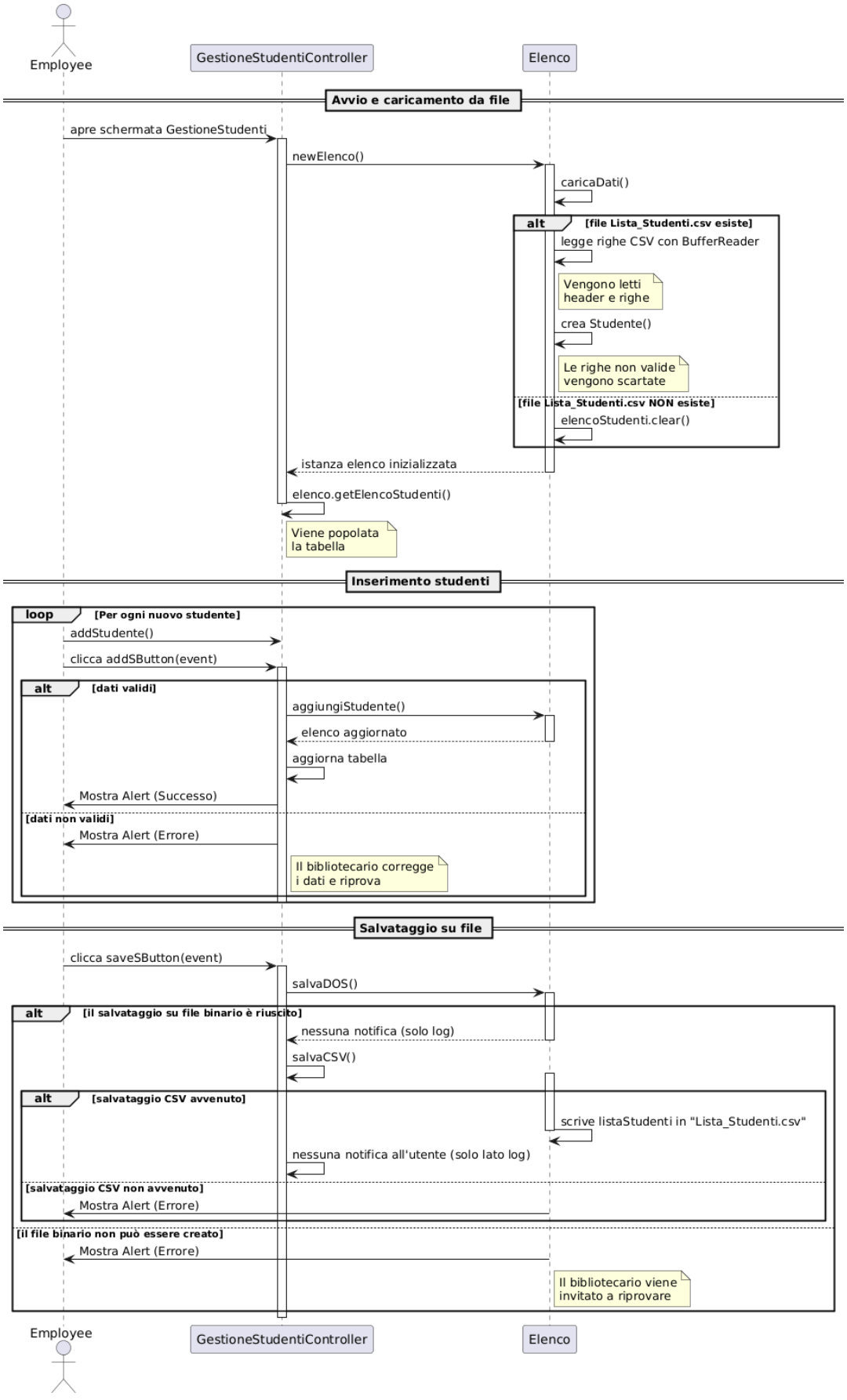
9. Salva Restituzione:



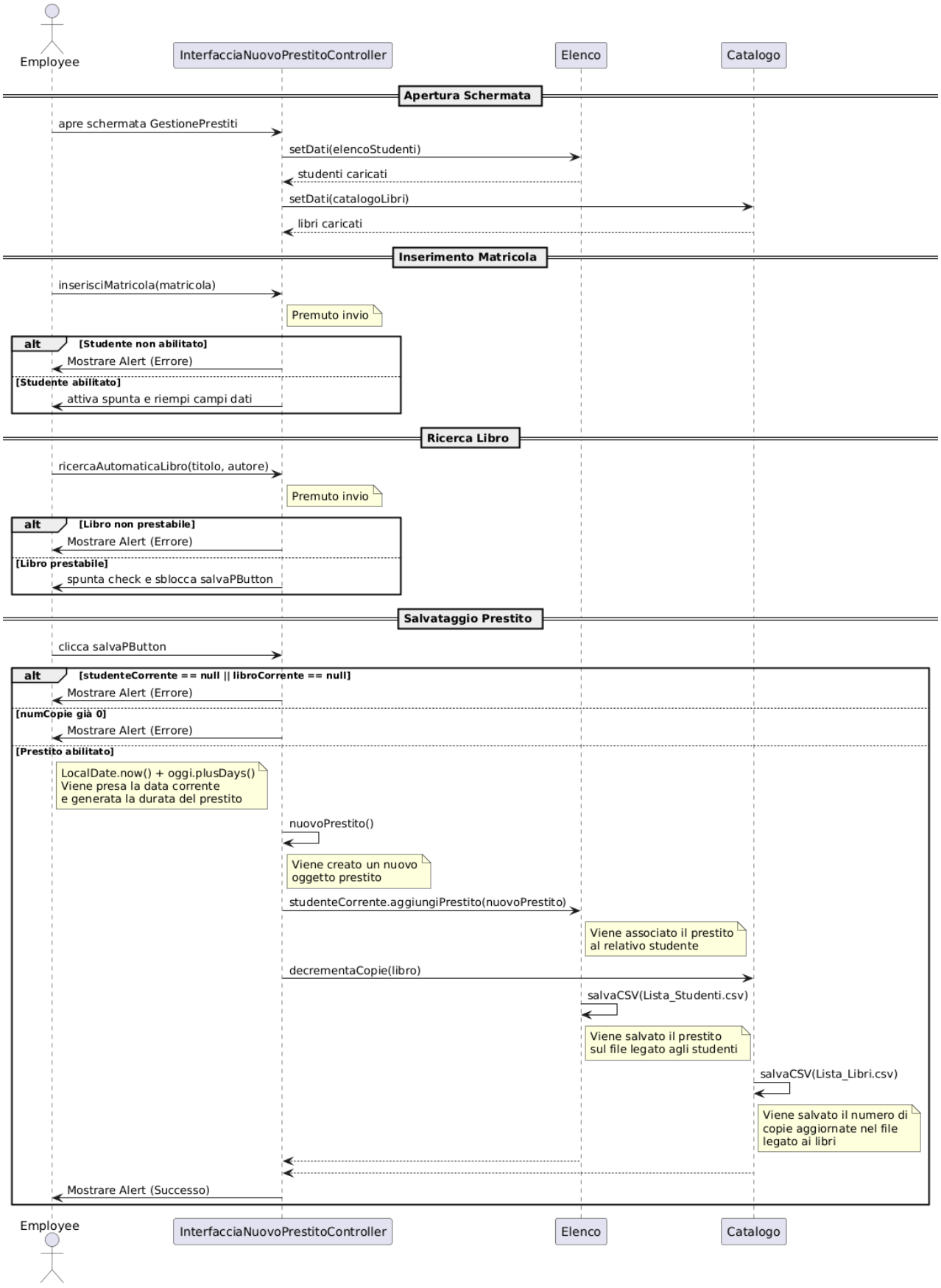
10. Salvataggio su file (GestioneLibri):



11. Salvataggio su file (GestioneStudenti):



12. Salvataggio su file (Interfaccia_nuovoPrestitoController):



COMMENTI AI DIAGRAMMI:

1) Questo diagramma di sequenza evidenzia come il controller della schermata GestioneStudenti filtra la lista degli studenti a partire dall'input inserito dal bibliotecario.

I partecipanti sono: l'actor Employee, il controller GestioneStudenteController, l'oggetto Elenco e un generico studente della lista.

Il primo messaggio mostrato è searchStudent(event), che parte da Employee e va verso il controller e rappresenta il click / l'azione di ricerca di un determinato studente

Successivamente, il controller pulisce la label di ricerca e poi recupera l'input inserito dal bibliotecario (la matricola e/o il cognome) dai campi dell'interfaccia.

A questo punto l'employee decide cosa fare in base al fatto che i campi siano vuoti o compilati, grazie al blocco alt : nella situazione dei "Campi vuoti", il controller chiama il metodo aggiornaInterfaccia() per mostrare tutti gli studenti, senza filtrare per matricola e/o cognome; al contrario, nel ramo "Campi valorizzati" il controller richiede alla classe Elenco la struttura con tutti gli studenti (getElencoStudenti() che restituisce TreeSet<Studente>).

All'interno del ramo "Campi valorizzati" il blocco loop garantisce l'iterazione "per ogni studente presente nella lista": In ognuna di esse, il controller invoca il metodo getMatricola() ed il metodo getCognome() sul singolo studente per leggere i valori da confrontare con l'input.

NOTA: dentro il loop c'è un blocco opt con il seguente controllo: "matricola contiene input CC cognome contiene input" che rappresenta la condizione di match: se quest'ultima è vera, il controller aggiunge quello studente alla lista risultatiRicerca.

Terminato il loop, Il controller popola la TableView chiamando tableViewStudenti.set items(risultati ricerca) affinché l'interfaccia mostri solo gli studenti filtrati.

Infine, un blocco opt con il seguente controllo: "risultatiRicerca vuota" gestisce il caso in cui non ci siano match e imposta la label con il messaggio "Nessuno studente trovato".

2) Questo diagramma di sequenza evidenzia come il controller della schermata GestioneLibri filtra i libri mostrati in tabella in base all'input inserito dal bibliotecario.

I partecipanti sono: l'actor Employee, il controller GestioneLibriController ed un generico oggetto Libro.

Il primo messaggio mostrato è searchLibri(event) che parte da Employee e va verso il controller e rappresenta il click / l'azione di ricerca di un determinato libro.

Successivamente, il controller prima pulisce la label del messaggio e poi recupera l'input inserito dal bibliotecario (il titolo e/o l'autore e/o l'ISBN) dai campi dell'interfaccia.

A questo punto l'employee decide cosa fare in base al fatto che i campi siano vuoti o compilati, grazie al blocco alt: nella situazione dei "Campi vuoti", il controller chiama il metodo tableViewLibri.setItems(datiTabella), per mostrare tutti i libri; nel ramo dei "Campi valorizzati", il controller procede con il filtraggio dei libri.

All'interno del ramo "Campi valorizzati", il blocco loop garantisce l'iterazione "per ogni libro contenuto in datiTabella": in ognuna di esse, il controller chiama i metodi getTitolo(), getAutore() e getIsbn(), per leggere i valori da confrontare con l'input.

NOTA: all'interno del loop c'è un blocco opt che scatta se il libro "corrisponde ai criteri di ricerca": in tal caso il libro viene aggiunto alla collezione risultatiRicerca.

Una volta finito il blocco loop, il controller aggiorna la tabella grazie al metodo tableViewLibri.setItems(risultatiRicerca) e, con un ultimo blocco opt, gestisce il caso "risultatiRicerca vuota" impostando la label con il messaggio di errore"Nessun libro trovato".

3) Questo diagramma di sequenza evidenzia il flusso di modifica dei dati di uno studente a partire dalla selezione in tabella fino all'aggiornamento della struttura dati e dell'interfaccia.

I partecipanti sono: l'actor Employee, il controller GestioneStudenteController, l'oggetto Elenco e un generico studente della lista.

Il primo messaggio è modStudent(event), metodo che viene invocato dall'Employee sul controller GestioneStudenteController, il quale recupera lo studente selezionato tramite il metodo getSelectedItem() e poi legge i nuovi dati inseriti (matricola, nuovo nome, nuovo cognome, nuova email) dall'input del bibliotecario.

Successivamente, il controller chiama il metodo modificaStudente(matricola, nuovoNome, nuovoCognome, nuovaEmail) che tramite il metodo cercaStudentePerMatricola(matricola), interroga la classe Elenco per trovare l'oggetto studente da aggiornare.

Nel blocco alt il ramo "Studente trovato" rappresenta il caso in cui la ricerca va a buon fine: qui lo studente viene rimosso dal TreeSet (grazie a elencoStudenti.remove(studente)), così che dopo la modifica possa avvenire in sicurezza il riordino; una volta rimosso, si aggiornano i singoli attributi dello studente tramite i metodi set.

Poi lo studente aggiornato viene reinserito nella struttura tramite il metodo elencoStudenti.add(studente), completando la fase di "re-inserimento studente aggiornato" nel TreeSet: se non ci sono stati errori, il metodo restituisce true, il controller richiama il metodo aggiornaInterfaccia() e notifica all'Employee un alert di conferma "Modificato con successo".

Invece, nel ramo "Studente non trovato" viene restituito false, il controller notifica all'Employee un alert di errore, quindi esegue pulisciCampi() per ripulire i campi di input e termina il flusso.

4) Questo diagramma di sequenza descrive il flusso di modifica di un libro nel catalogo, mostrando come il controller aggiorna l'oggetto libro e il TreeSet che compone l'inventario.

I partecipanti sono: l'actor Employee, il controller GestioneLibriController, l'oggetto Catalogo e un generico libro della lista.

Il primo messaggio mostrato è da parte dell'Employee, il quale invoca il metodo modLibri(event) sul controller per la gestione dei libri: questo metodo permette di recuperare la riga selezionata in tabella tramite getSelectedItem() e poi legge i nuovi dati inseriti (isbn, nuovo titolo, nuovo autore, anno, copie) dal bibliotecario stesso.

Successivamente, il controller chiama il metodo modificaLibro(isbn, nuovoTitolo, nuovoAutore, anno, copie) e richiede al Catalogo il libro da modificare: tutto questo tramite il metodo cercaLibroPerISBN(isbn), ottenendo il riferimento a libroSelezionato:Libro.

C'è poi un blocco opt con il seguente controllo: "Titolo o Autore cambiati". Rappresenta il caso in cui le modifiche hanno un peso specifico sull'ordinamento del TreeSet. In questo caso, prima viene eseguita la "rimozione necessaria per il riordinamento del TreeSet", tramite inventarioLibri.remove(libroSelezionato), poi vengono aggiornati gli attributi del libro con setTitolo(nuovoTitolo), setAutore(nuovoAutore) e setNumCopia(nuoveCopie).

Un secondo blocco opt (sempre condizionato a "Titolo o Autore cambiati") modella il "reinserimento del libro aggiornato" nella struttura tramite inventarioLibri.add(libroSelezionato), così il TreeSet si riordina correttamente.

Dopo aver terminato l'operazione viene restituito true al controller, il quale esegue il refresh() della tabella, mostra un alert "Modificato con successo" all'utente e infine chiama pulisciCampi() per resettare i campi della form.

5) Questo diagramma di sequenza descrive il flusso di eliminazione di uno studente dalla lista, includendo la conferma tramite finestra di dialogo e la gestione dell'esito (successo o errore).

I partecipanti sono: l'actor Employee, il controller GestioneStudenteController, l'oggetto Elenco e un generico studente della lista.

Come primo messaggio mostrato vi è l'invocazione da parte dell'Employee del metodo deleteStudent(event) sul controller GestioneStudenteController, il quale recupera l'elemento selezionato con getSelectedItem() e costruisce un Alert(CONFIRMATION) per chiedere conferma.

Successivamente, Il controller chiama il metodo showAndWait() sull'alert e, dopo il click su OK, ottiene un result che, se uguale a OK, abilita il blocco opt legato al controllo result == OK: tutto ciò per procedere con eliminaStudente(matricola) verso la classe Elenco.

Dentro il metodo eliminaStudente(matricola) c'è un blocco loop "per ogni studente in elencoStudenti" che modella l'iterazione su tutti gli studenti.

Per ognuno di loro, Elenco chiama il metodo getMatricola() e nel blocco opt col controllo "matricolaCorrente == matricola", rimuove l'oggetto corrispondente con elencoStudenti.remove(studente) e ritorna true, uscendo immediatamente dal metodo come indicato dalla nota "return true (esce dal metodo)".

NOTA: se il ciclo termina senza trovare alcuna corrispondenza, il metodo restituisce false , come spiegato dalla nota "Se ciclo finisce senza return".

Al ritorno nel controller, un blocco alt permette di distinguere 2 situazioni: "Rimosso con successo" (valore true, chiamata a aggiornaInterfaccia() per aggiornare la tabella) e "Errore" (valore false, notifica all'Employee un alert di errore e chiude poi il flusso verso l'Employee.

6) Questo diagramma di sequenza rappresenta il flusso di eliminazione di un libro dal catalogo, con la conferma via finestra di dialogo e la gestione del caso libro trovato/non trovato.

I partecipanti sono: l'actor Employee, il controller GestioneLibriController, l'oggetto Catalogo e un generico libro della lista.

Come primo messaggio mostrato vi è l'invocazione da parte dell'Employee del metodo deleteLibri(event) sul controller GestioneLibriController, il quale recupera il libro selezionato tramite getSelectedItem() e crea un Alert(CONFIRMATION) per chiedere conferma all'utente.

Dopo la chiamata a showAndWait(), quando l'utente clicca OK, il controller riceve un result (OK) e, se result == OK, entra nel blocco opt che richiama il metodo eliminaLibro(isbn), operando direttamente sul Catalogo.

NOTA: all'interno di eliminaLibro(isbn) il Catalogo cerca il libro tramite cercaLibroPerISBN(isbn) e il blocco alt distingue tra libro trovato e libro non trovato.

Se il libro viene trovato, inventarioLibri.remove(libro) rimuove l'oggetto e il metodo restituisce true ; se così non rosse, restituisce false. Il controller, ricevuto il boolean, chiama comunque aggiornaInterfaccia() per aggiornare la tabella.

7) Questo diagramma di sequenza rappresenta il flusso di inserimento di un nuovo studente, includendo la validazione dell'email e il controllo di duplicati sulla matricola.

L'Employee richiama il metodo addStudent(event) su GestioneStudenteController, che permette di recuperare i dati inseriti nella form..

Vi è poi un blocco opt legato al controllo "Email non valida" che modella la validazione: se l'email non rispetta i criteri, viene mostrato un alert "Email non valida" e il flusso può interrompersi senza creare lo studente.

Se l'email è valida, il controller crea un nuovo oggetto Studente con i dati forniti e con i parametri aggiuntivi (ad esempio uno spazio e false per altri attributi), poi invoca aggiungiStudente(nuovoStudente) sulla classe Elenco; quest'ultima esegue una ricerca tramite cercaStudentePerMatricola(matricola) e, nel blocco alt , distingue tra Studente trovato (duplicato) e Studente non trovato (nuovo).

Esito: errore o inserimento riuscito:

Nel ramo "Studente trovato (duplicato)" il metodo restituisce false al controller, che mostra un alert di errore generico ("Errore"), segnalando che esiste già uno studente con quella matricola.

Nel ramo "Studente non trovato (nuovo)", Elenco aggiunge lo studente alla struttura con elencoStudenti.add(nuovoStudente) , restituisce true , il controller chiama il metodo aggiornaInterfaccia() per aggiornare la tabella e infine pulisciCampi() per resettare il form.

8) Questo diagramma di sequenza descrive il flusso di aggiunta di un libro al catalogo, distinguendo tra libro già presente (incremento copie) e libro completamente nuovo.

I partecipanti sono: l'actor Employee, il controller GestioneLibriController, l'oggetto Catalogo e un generico libro della lista.

Il primo messaggio che viene mostrato riguarda l'invocazione da parte dell'Employee del metodo addLibri(event) sul controller GestioneLibriController, il quale recupera i dati inseriti (titolo, autore, isbn, ecc.) in input dal bibliotecario.

Successivamente, Il controller invoca getInventarioLibri() sul Catalogo e ottiene la collezione inventarioLibri su cui poter operare.

NOTA: un blocco loop "per ogni libroEsistente in inventario" permette di gestire l'iterazione sui libri già presenti: per ognuno di essi, il controller confronta i valori ottenuti da getTitolo(), getAutore() e getIsbn(), con i dati inseriti in input; nel blocco alt "libro identico trovato" chiama il metodo incrementaCopie(isbn) sul Catalogo, che a sua volta esegue incrementaCopie(1) sull'oggetto libro e poi esce dal ciclo (seguendo la nota "break loop").

Se il ciclo termina senza trovare un libro identico, entra in gioco il blocco opt col controllo "libro non trovato (nuovo)": qui viene creato un nuovo oggetto Libro(isbn, titolo, autore, anno, 1), il quale viene passato ad aggiungiLibro(nuovoLibro), che lo inserisce in inventarioLibri tramite inventarioLibri.add(nuovoLibro); il metodo restituisce true, il controller notifica all'Employee un alert "Libro aggiunto", aggiorna l'interfaccia con aggiornaInterfaccia() ed infine chiama il metodo pulisciCampi() per resettare i valori del form.

9) Questo diagramma di sequenza descrive il flusso di salvataggio di una restituzione di prestito, dalla ricerca dello studente fino all'aggiornamento delle copie del libro, delle sanzioni e dei file CSV.

I partecipanti sono: l'actor Employee, il controller Interfaccia_nuovoPrestitoController, elencoStudenti:Elenco (collezione che gestisce gli studenti e i loro prestiti), studente:Studente, prestitoDaChiudere:Prestito (il prestito specifico che viene chiuso), catalogoLibri:Catalogo (il catalogo dei libri della biblioteca), libro:Libro.

Come primo messaggio mostrato abbiamo l'invocazione da parte dell'Employee del metodo salvaRestituzione(event) sul controller Interfaccia_nuovoPrestitoController, il quale chiede a elencoStudenti lo studente associato a una matricola tramite cercaStudentePerMatricola(matricola), ottenendo un oggetto studente.

C'è un primo blocco alt "studente != null" che gestisce il caso in cui lo studente esista: il controller richiede la lista dei prestiti attivi tramite getPrestitiAttivi() e, in maniera totalmente iterativa, individua il prestitoDaChiudere con l'ISBN corrispondente; se non lo trova si attiverà il ramo "Prestito non trovato".

Nel ramo alt "prestitoDaChiudere != null" viene impostata la data di restituzione tramite setDataRestituzione(LocalDate.now()) sul prestito: successivamente, il controller chiede al catalogoLibri l'inventario con getInventarioLibri(), quindi entra in un blocco loop per ogni libro nel catalogo e, nel blocco opt "ISBN corrisponde", invoca il metodo incrementaCopie(1) sul libro, aumentando di una unità le copie disponibili di quel determinato libro.

Dopo aver aggiornato le copie, il controller calcola l'eventuale penalità con gestioneSanzioni() sul prestito, riceve esitoSanzione e lo registra invocando setSanzione(esitoSanzione) ---> Poi rimuove il prestito chiuso dalla lista dello studente tramite rimuoviPrestito(prestitoDaChiudere) e salva le modifiche chiamando salvaCSV() sia su elencoStudenti (modificando quindi il file "Lista_Studenti.csv") sia su catalogoLibri (modificando quindi il file "Lista_Libri.csv").

Se tutto va a buon fine, il controller mostra un alert di successo ("Mostra Alert (Successo)").

NOTA: in caso di prestito non trovato viene notificato all'Employee un alert (Errore: Nessun prestito)", mentre se lo studente non viene trovato si attiva il ramo "Studente non trovato" con l'alert (Errore: Non trovato), segnalando all'Employee dove si è interrotto il flusso.

10) Questo diagramma di sequenza mostra il ciclo completo per il caricamento su file delle informazioni legate ai libri: caricamento da file all'avvio, inserimento di nuovi libri e salvataggio del catalogo su CSV.

I partecipanti sono l'actor Employee, il controller GestioneLibriController e la classe Catalogo.

Il primo messaggio che viene mostrato riguarda il cliccare un button da parte dell'Employee: si apre la schermata GestioneLibri e il controller GestioneLibriController richiede l'istanza del Catalogo tramite getIstanza(), poi Catalogo invoca caricaCSV() .

Nel blocco alt viene gestito se il file Lista_Libri.csv "non esiste": ciò garantisce la pulizia dell'inventario con inventarioLibri.clear()); oppure se il file Lista_Libri.csv "esiste": in questo caso vengono lette le righe CSV tramite un BufferedReader, vengono creati gli oggetti Libro() e vengono scartate le righe non valide.

Dopo aver inizializzato il catalogo, il controller ottiene l'inventario con catalogo.getInventarioLibri() e popola la tabella dell'interfaccia.

NOTA: nella sezione "Inserimento libri" c'è un blocco loop "per ogni nuovo libro" che indica che per ogni click sul pulsante addLButton(event) che chiama addLibro(): se i dati sono validi, aggiungiLibro aggiorna l'inventario e la tabella e viene notificato all'Employee un alert di successo; se sono invalidi, viene notificato all'Employee un alert di errore e il bibliotecario corregge i dati e riprova.

Nella sezione "Salvataggio su file", l'Employee preme saveLButton(event) e il controller chiama il metodo salvaCSV() sul Catalogo.

-->Il blocco alt "salvataggio avvenuto" definisce il caso in cui la scrittura del CSV vada a buon fine (scrittura di inventarioLibri in "Lista_Libri.csv" senza notifiche all'utente, solo log); nel ramo "salvataggio non avvenuto" viene invece notificato all'Employee un alert di errore per segnalare il problema di persistenza.

11) Questo diagramma di sequenza illustra l'intero ciclo per il caricamento su file delle informazioni legate agli studenti: caricamento da file all'avvio, inserimento di nuovi studenti e salvataggio finale su file binario e CSV.

I partecipanti sono l'actor Employee, il controller GestioneStudentiController e la classe Elenco.

Il primo messaggio che viene mostrato riguarda il cliccare un button da parte dell'Employee: si apre la schermata GestioneStudenti e qui il controller crea un nuovo Elenco (newElenco()) e gli chiede di caricare i dati tramite il metodo caricaDati().

Nel blocco alt viene gestito il caso in cui il file Lista_Studenti.csv "esista": vengono lette righe e header, creati gli oggetti Studente validi e vengono scartate le righe non valide; se il file "non esiste" viene eseguito elencoStudenti.clear() per inizializzare una lista vuota.

--> Dopo questa prima fase di inizializzazione, il controller ottiene la struttura tramite elenco.getElencoStudenti() e popola la tabella.

Nella sezione "Inserimento studenti" c'è un blocco loop "per ogni nuovo studente", il quale mostra che ogni click su addSButton(event) scatena addStudente(): se i dati sono validi, l' Elenco aggiorna la propria collezione, la tabella viene aggiornata e viene notificato all'Employee un alert di successo; se i dati non sono validi, viene notificato all'Employee un alert di errore e il bibliotecario corregge e riprova.

Nella sezione finale, "Salvataggio su file", l'Employee clicca saveSButton(event) e il controller invoca il metodo salvaDOS() su Elenco per il file binario.

Successivamente, un blocco alt gestisce se il salvataggio binario è riuscito o meno: in caso di successo non c'è notifica all'utente (solo log), in caso di errore viene notificato all'Employee un alert di errore.

Infine, Elenco chiama salvaCSV(): se il salvataggio CSV va a buon fine, il sistema scrive listaStudenti in "Lista_Studenti.csv" senza notifiche visive, altrimenti viene notificato all'Employee un alert di errore e, come indicato nella nota, il bibliotecario è invitato a riprovare.

12) Questo diagramma di sequenza descrive l'intero flusso di caricamento su file di un nuovo prestito: dalla schermata iniziale, al controllo di studente e libro, fino al salvataggio su file e alla notifica di successo.

i partecipanti sono l'actor Employee, il controller Interfaccia_nuovoPrestitoController , Elenco (struttura che contiene e gestisce gli studenti), Catalogo (struttura che contiene e gestisce i libri).

Il primo messaggio che viene mostrato indica il premere un bottone da parte dell'Employee: facendo ciò, egli apre la schermata GestionePrestiti e l'Interfaccia_nuovoPrestitoController inizializza i dati invocando setDati(elencoStudenti) e setDati(catalogoLibri) verso Elenco e Catalogo, i quali forniscono liste di studenti e libri.

NOTA: la nota "Viene popolata la tabella" indica che a questo punto l'interfaccia può mostrare le informazioni necessarie per creare un prestito.

Successivamente, nella sezione "Inserimento Matricola" l'Employee chiama il metodo inserisciMatricola(matricola) e, dopo il "Premuto invio", un blocco alt distingue tra "Studente non abilitato", con notifica all'Employee di un alert di errore, e "Studente abilitato" , in cui il sistema attiva la spunta e riempie automaticamente i campi dello studente.

Nella sezione di "Ricerca Libro" la chiamata ricercaAutomaticaLibro(titolo, autore) ed il successivo invio attivano un altro blocco alt: se il libro è "non prestabile" viene notificato all'Employee un alert di errore, se è "prestabile" viene spuntata la check e sbloccato il bottone salvaPButton.

Alla pressione di salvaPButton, la sezione "Salvataggio Prestito" usa un blocco alt con tre rami: "studenteCorrente == null || libroCorrente == null" notifica all'Employee un alert di errore; "numCopie già 0" segnala che non ci sono copie disponibili; "Prestito abilitato" esegue il flusso positivo degli eventi.

--> Nel caso positivo viene calcolata la data del prestito tramite LocalDate.now() + oggi.plusDays(), viene poi creato nuovoPrestito(), associato allo studente tramite studenteCorrente.aggiungiPrestito(nuovoPrestito) ed infine viene decrementato il numero di copie del libro invocando decrementaCopie(libro).

Successivamente, Il controller chiede ad Elenco di salvare gli studenti aggiornati con salvaCSV(Lista_Studenti.csv) e a Catalogo di salvare le copie aggiornate con salvaCSV(Lista_Libri.csv), come indicato dalle note accanto ai messaggi.

NOTA: se non emergono errori di salvataggio, il flusso si chiude con "Mostrare Alert (Successo)", che notifica all'Employee l'avvenuta registrazione del prestito tramite un alert di successo.