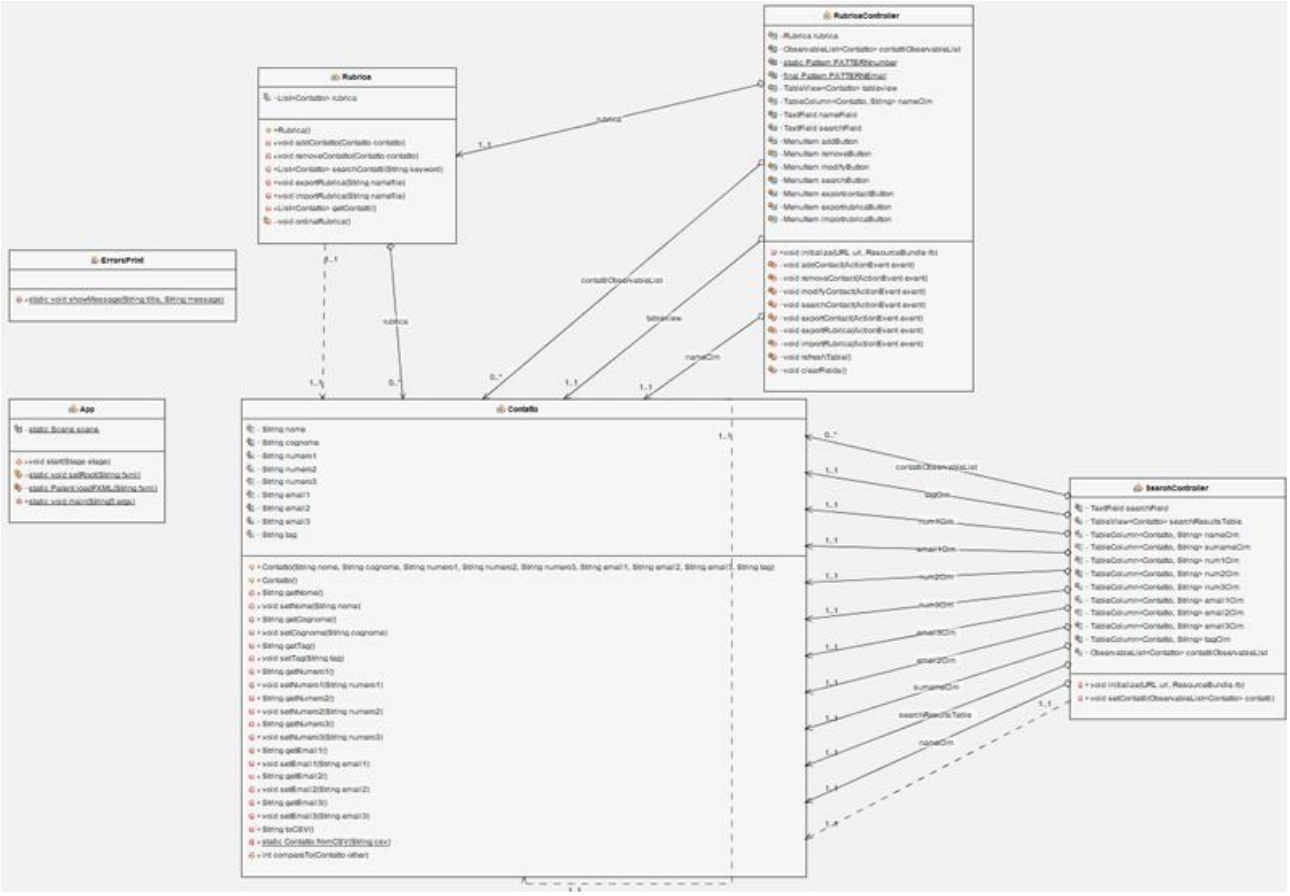
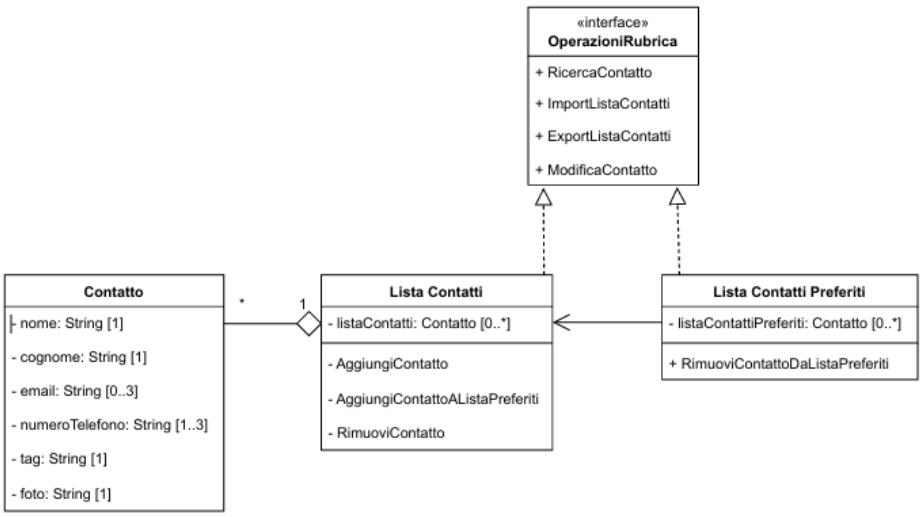
1.Diagramma delle classi

1.1Diagramma delle classi generato tramite EasyUML



1.2 Diagramma delle classi semplificato senza EasyUML



Breve descrizione delle classi

- Contatto: utilizzata per salvare le informazioni relative ad un contatto, contiene tutti i metodi setter e getter riguardo ai suoi attributi oltre ai 2 metodi per esportare un contatto in CSV ed importare un contatto da file CSV

- Rubrica: utilizzata per contenere più contatti al suo interno, ordinati alfabeticamente per cognome e nome, contiene tutti i metodi per aggiungere, rimuovere, modificare, cercare ed esportare contatto, oltre che ai metodi per l' export e l'import dell'intera rubrica e a due metodi per pulire i campi di inserimento nella GUI e ricaricare la rubrica sulla table della GUI
- RubricaController: gestisce tutte le operazioni che l'utente vuole effettuare sulla GUI (chiamata ai metodi di addContact, removeContact ecc...) e il file xml relativo alla parte grafica

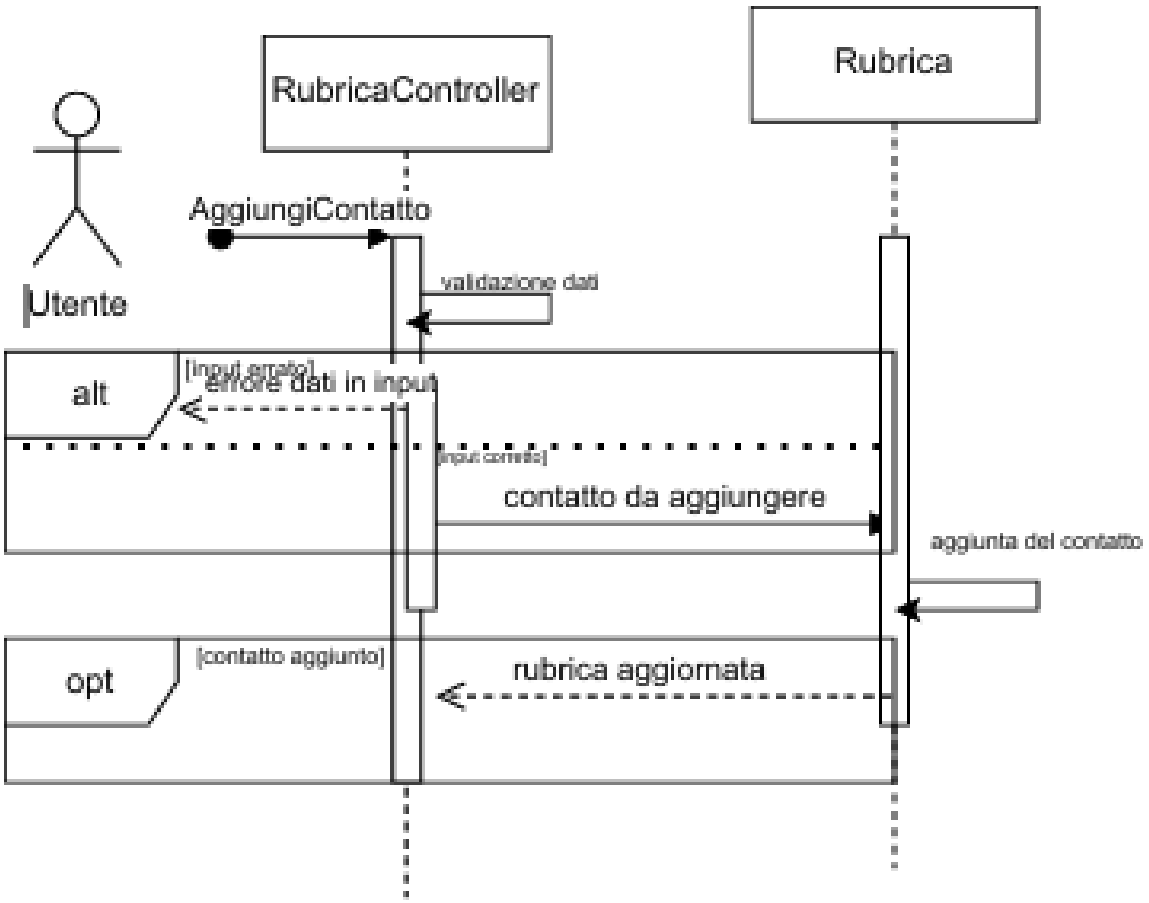
- SearchController: gestisce l'operazione di ricerca dei contatti all'interno della rubrica e gestisce anche il file xml relativo alla parte grafica del software

- ErrorsPrint: è una classe di appoggio per la creazione di una nuova finestra grafica ogni qual volta c'è da mostrare un' errore a video

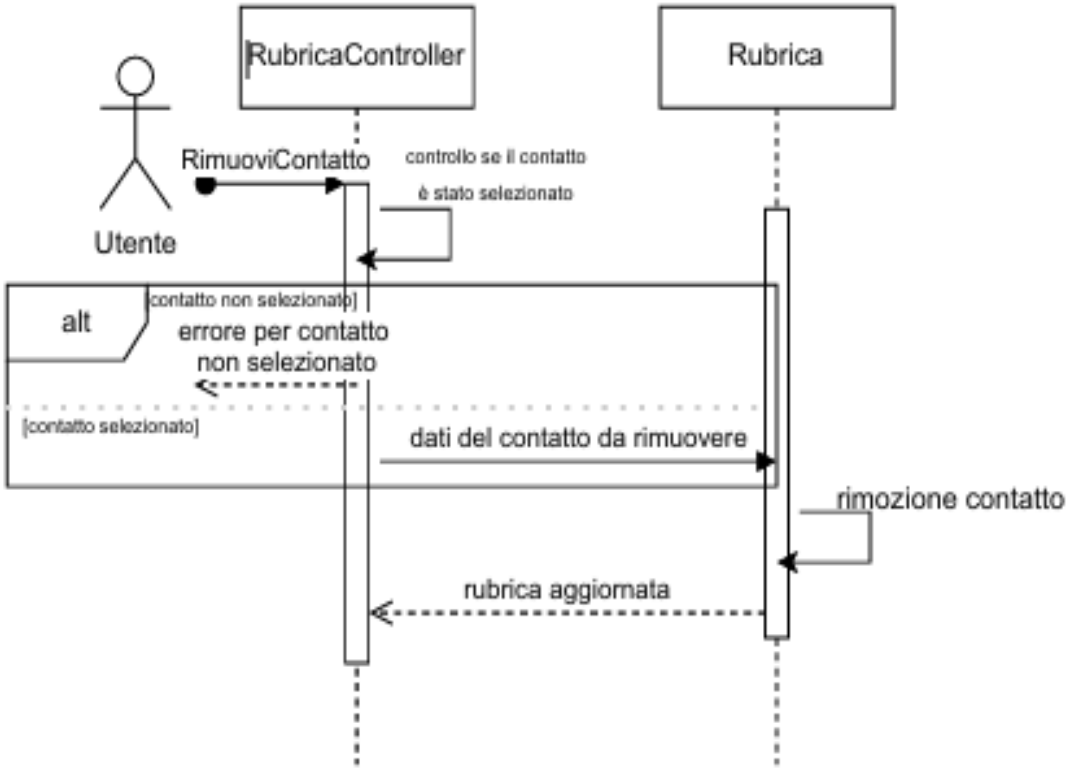
- App: è la classe main per avviare il software

2.Diagrammi di sequenza

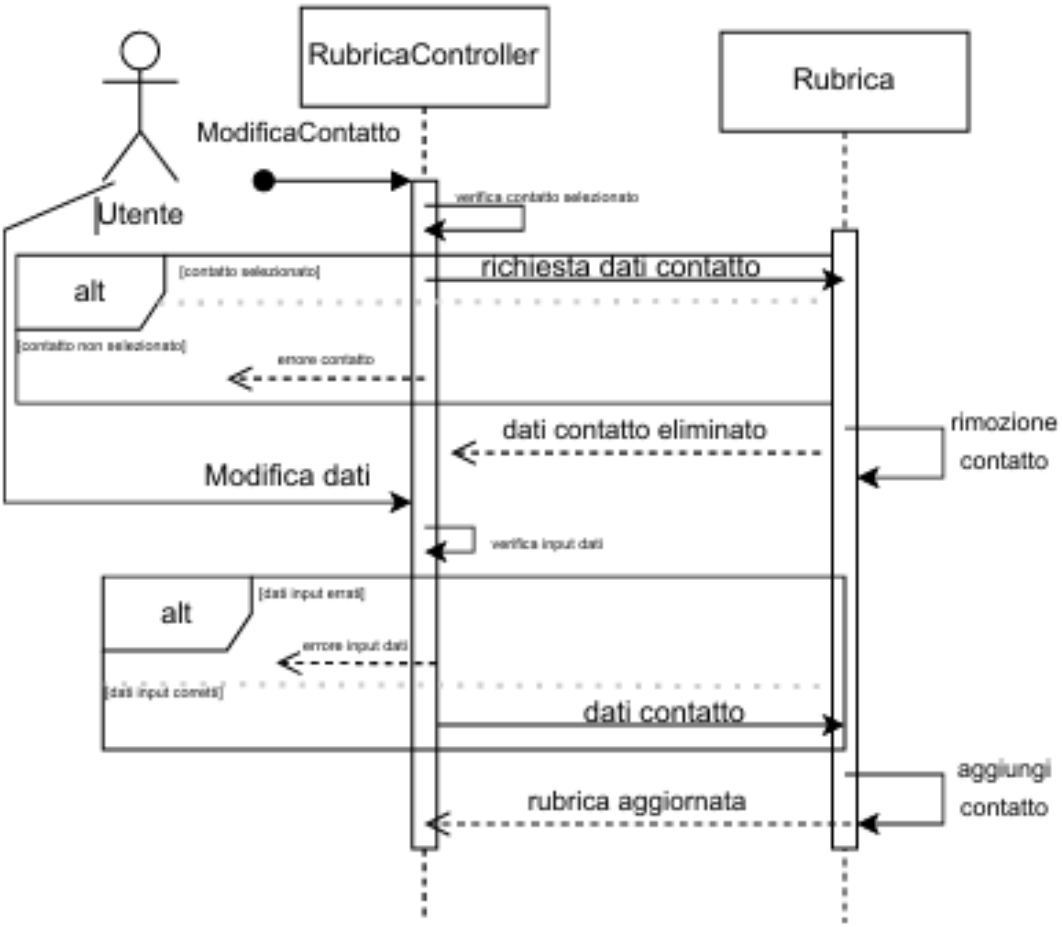
- Aggiungi contatto:



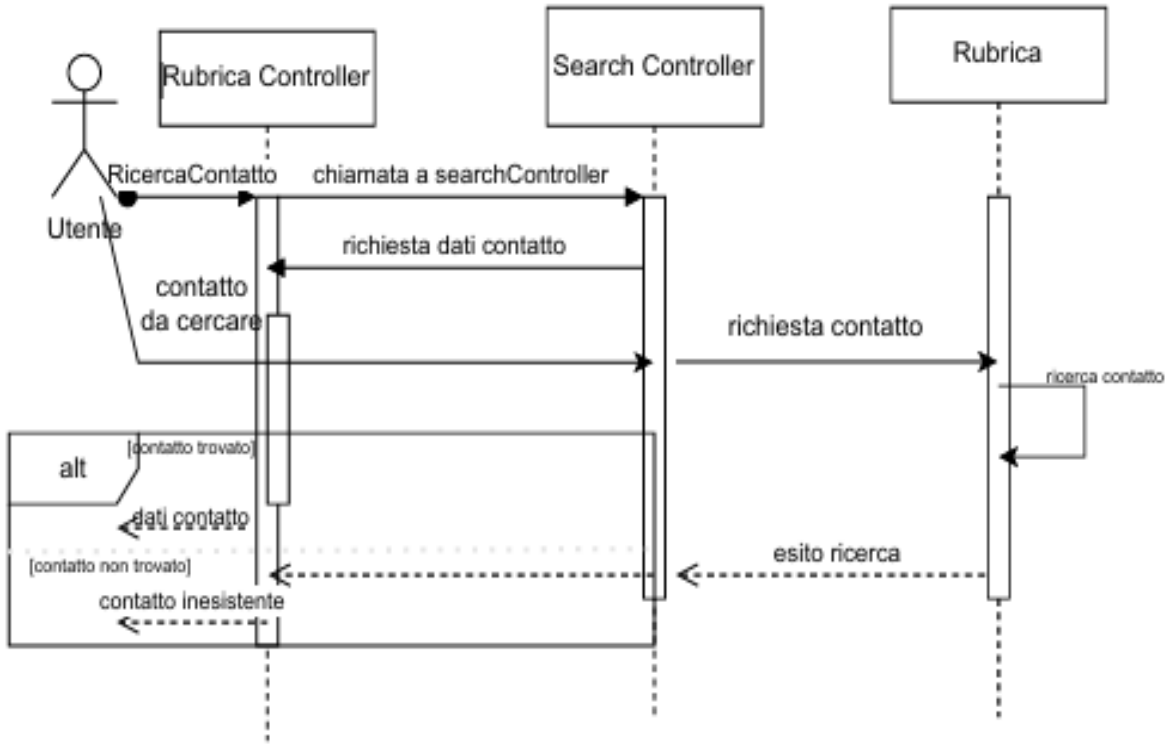
- Rimuovi contatto:



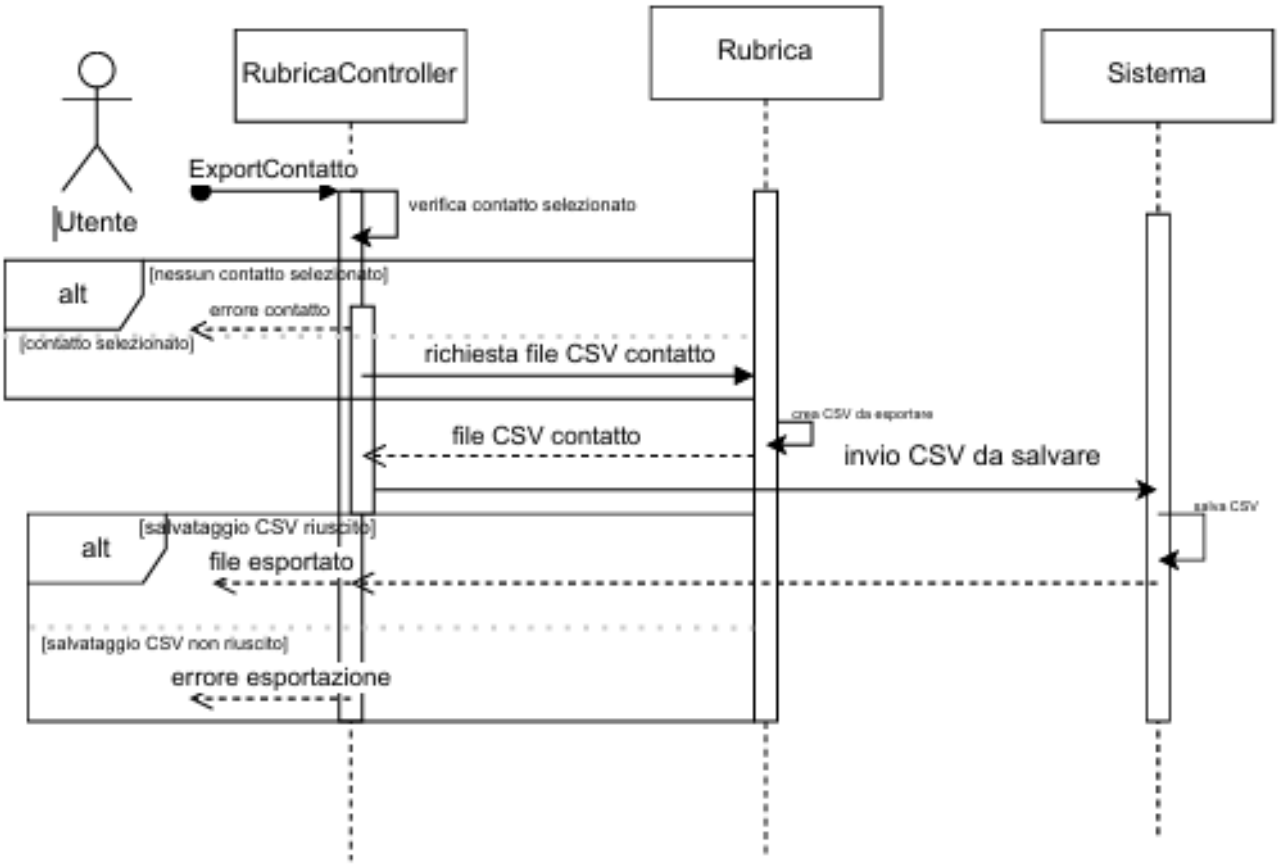
- Modifica contatto



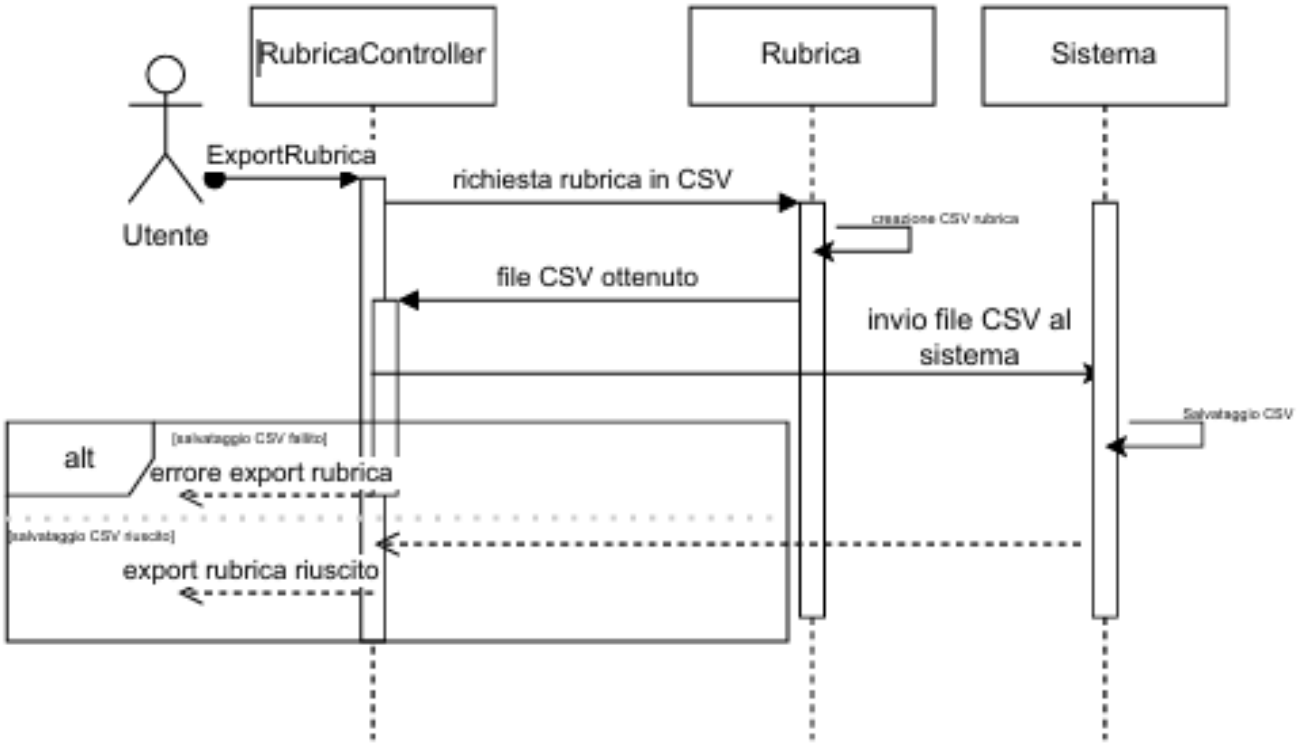
- Cerca contatto:



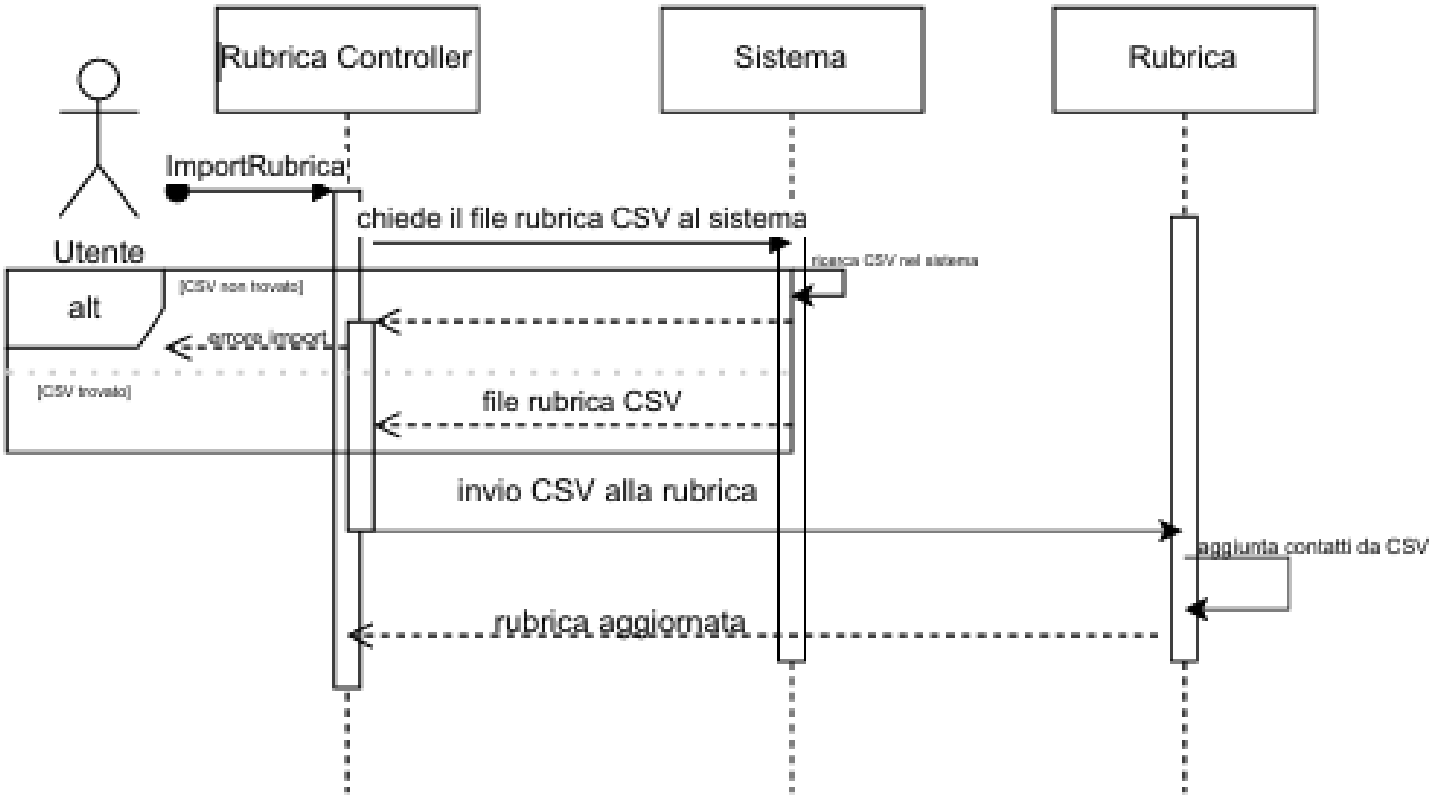
- Export contatto



- Export rubrica:



- Import rubrica:



3. Commenti ai diagrammi di sequenza

Per motivi legati alle tempistiche strette siamo riusciti a raggiungere un certo livello di astrazione che non è il più alto. Abbiamo cercato di mantenere comunque degli standard per quanto riguarda le varie classi un determinato livello di coesione, accoppiamento e principi di buona progettazione

3.1 Coesione

- Classe “Contatto”: essendo focalizzata esclusivamente sulla rappresentazione di un contatto come entità (rappresentando comunque gli attributi definiti da un contatto), ha un’ottima coesione e poca dipendenza con altre classi. Definendo al suo interno solo gli attributi per rappresentare se stessa rispetta chiaramente anche il Principio di Responsabilità singola (utile per i principi di buona progettazione)

- Classe “Rubrica”: responsabile della gestione della lista dei contatto (rubrica) e dei metodi associati (quali aggiunta, rimozione, ricerca ecc...). Non assume troppa responsabilità per quanto riguarda la gestione logica degli input o della validazione dati proprio per poter astrarre il più possibile la progettazione. Questo intuitivamente ci dice che è

- Classe “RubricaController” e “SearchController” : sono le responsabili della gestione della parte grafica dell’applicazione ed hanno un alto livello di coesione dal momento che si occupano nello specifico di 2 grafiche diverse che hanno poco a che fare tra loro e non hanno metodi o azioni in comune tra loro. La cosa utile che si sarebbe potuta fare era di astrarre la parte di validazione dati (PATTERN) della RubricaController in un’ ulteriore classe per delegare la parte logica dei dati a qualcun altro ma per questione di tempi non si è riusciti.

3.2 Accoppiamento

- Rubrica e Contatto: la relazione tra le 2 classi è inevitabile e giusta dal momento che la rubrica gestisce una collezione di oggetti di tipo Contatto.
- RubricaController e Rubrica: la relazione tra le 2 classi è più o meno forte dal momento che RubricaController gestisce gli accessi ai dati ed ai metodi di rubrica mediante una lista osservabile nella parte grafica
- SearchController e Rubrica: la relazione è simile alla precedente in quanto SearchController sfrutta un metodo definito in rubrica per effettuare la ricerca di uno o più contatti all'interno della rubrica

3.3 Principi di Buona Progettazione

- Principio di Singola Responsabilità: la struttura generale rispetta questo principio in quanto le classi sono ben separate in termini di responsabilità tranne che per alcune funzionalità tra Rubrica e RubricaController

- Separazione delle Preoccupazioni: i dati sono stati ben separati dalla logica del loro utilizzo (Contatto, Rubrica da RubricaController e SearchController) implementando la logica nei controller e quindi dividendola da quelli che sono i "contenitori" dei dati

- Principio DRY: nella parte responsabile della logica non ci sono metodi duplicati, ovvero, tra RubricaController(che gestisce

aggiunta, rimozione, modifica,export dati ed inserimento) e SearchController(che gestisce solo una nuova finestra grafica per la ricerca di un contatto)

- Principio della minima sorpresa: il codice è stato scritto utilizzando le convenzioni java e riferendosi a determinati oggetti o metodi utilizzando lo stesso metro di giudizio e quindi senza lasciare spazio ad ambiguità (aggiunti anche commenti per ricordare al programmatore per cosa sia stato scritto un determinato modulo o attributo)

- Principio aperto/chiuso: le classi hanno tutte attributi privati (presenza dell'incapsulamento) per evitare modifiche involontarie a dati sensibili. I metodi accessbili da altre classi sono dichiarati public mentre quelli che non dovrebbero fornisce accesso esterno sono stati dichiarati private (e quindi incapsulato). Un'ulteriore possibile livello di astrazione per questo punto sarebbe potuto essere l'aggiunta di un pattern di livello superiore per modificare i parametri di ricerca in SearchController per non dover cambiare gli attributi di "Contatto". Oppure astrarre alcune funzionalità di RubricaController per supportare nuove funzionalità senza dover aggiungere metodi all'interno della classe Rubrica

Conclusione

Analizzando quelle che erano le nostre specifiche e il progetto teorico ci siamo concentrati sul progettare la base del nostro software, siamo poi, pian piano, saliti di livello di astrazione a seconda delle tempistiche disponibili e delle scadenze fornite dal nostro "cliente" fino ad arrivare a questo punto, rispettando i requisiti fondamentali

richiesti. Di seguito proponiamo prospettive di miglioramento per il software per renderlo più robusto e mantenibile:

- Implementare un maggiore livello di astrazione tra livelli di logica, dati ed interfaccia grafica
- Applicare pattern architetturali per favorire la modularità e la scalabilità
- Rivedere il design con focus su coesione, accoppiamento e aderenza ai principi SOLID