2024/2025

Ingegneria del Software

BricoRubrica

Gaetano Carbone , Mario Pellegrino Ambrosone, Michele Cetraro e Simone Grimaldi

Contents

1 Meta Architecture

Il sistema software BricoRubrica è progettato per essere un'applicazione di dimensioni ridotte e complessità limitata. Come evidenziato durante la fase di Requirements Engineering, il sistema deve:

1. Gestire una lista dinamica di contatti modificabili.
2. Consentire il salvataggio e il caricamento dei contatti da un file esterno.

Dati i requisiti attuali, non si prevede che il sistema necessiti di ulteriori funzionalità in futuro. Questa considerazione consente di adottare un'architettura progettata per rispondere alle esigenze specifiche del sistema, anche a costo di una ridotta flessibilità dei componenti.

Architectural Style - M.V.C.

Lo stile architetturale adottato è Model-View-Controller poiché particolarmente adatto per applicazioni con interfacce grafiche (GUI).
Nello specifico il Model si costituisce della sezione di logica gestionale (gestione dei contatti) e rappresenta i dati, la View si compone dei file FXML che forniscono l'interfaccia grafica, con tutte le sue sotto-interfacce, e il Controller funge da intermediario regolando le interazioni dell'utente con l'interfaccia e la logica di background.

Architectural Views

La progettazione del sistema di basa su due viste architetturali principali, elencate di seguito, selezionate per rispondere alle "ridotte" esigenze del progetto e alla sua contenuta complessità.

- *Logical View*, per la decomposizione in moduli, l'analisi e lo sviluppo delle classi (class diagram) e la divisione delle classi in package in funzione dei moduli ricavati (packages diagram);
- *Process View*, per lo sviluppo dei diagrammi di sequenza, al fine di chiarire i flussi di esecuzione principali.

2 Logical View

2.1 Module Design

Per la decomposizione in moduli si è optato per un approccio misto, con un primo step di tipo funzionale, continuando poi con modalità object oriented, al fine di definire meglio le componenti interne dei moduli e semplificare lo sviluppo delle classi. Figure 1 contiene la prima divisione in funzionalità, dove abbiamo, da sx verso dx:

- Interface, modulo dedicato all'interfaccia e ai controller ad essa associati;
- Logic, modulo dedicato alla logica di controllo e di gestione dei contatti;
- Data, modulo dedicato alla persistenza dei dati.

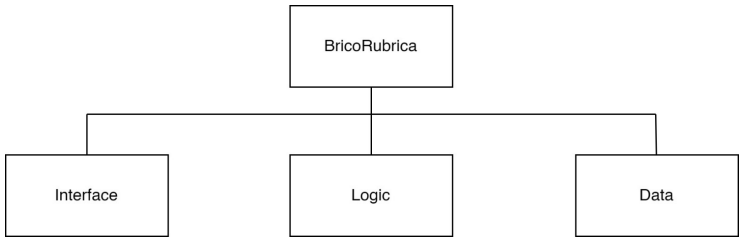

Figure 1: Functional Decomposition - 1^{st} step

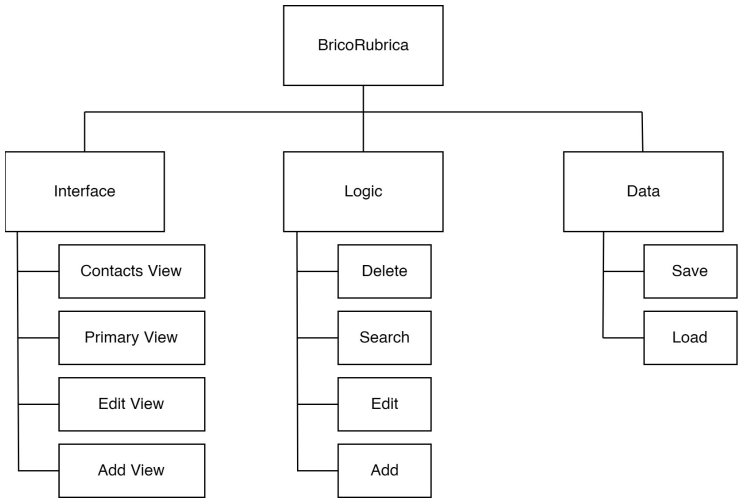

Figure 2: OO Decomposition - 2^{nd} step

Procediamo con un approccio OO per semplificare i moduli, ancora troppo generali, e iniziare a definire le prime classi del sistema (vedi Figure 2).
Il modulo Interface si divide in 4 sotto moduli, ognuno atto alla gestione di una partizione specifica dell'interfaccia, ovvero Primary View, Contacts View, Edit View e Add View.
Anche il modulo Logic si divide in 4 sotto moduli, ognuno dei quali sviluppa una delle funzionalità manageriali sui contatti, ovvero Delete, Search, Edit e Add. Il modulo Data infine si compone di 2 sotto moduli Save e Load, rispettivamente per il salvataggio della rubrica su file esterno e per il caricamento della rubrica salvata nell'ultima sessione.

L'ultimo step della progettazione dei moduli (vedi Figure 3) serve a precisare la logica di funzionamento dei moduli Edit e Add che, seppur non rappresenti una classe assestante, è comune solo ai due moduli appena citati.

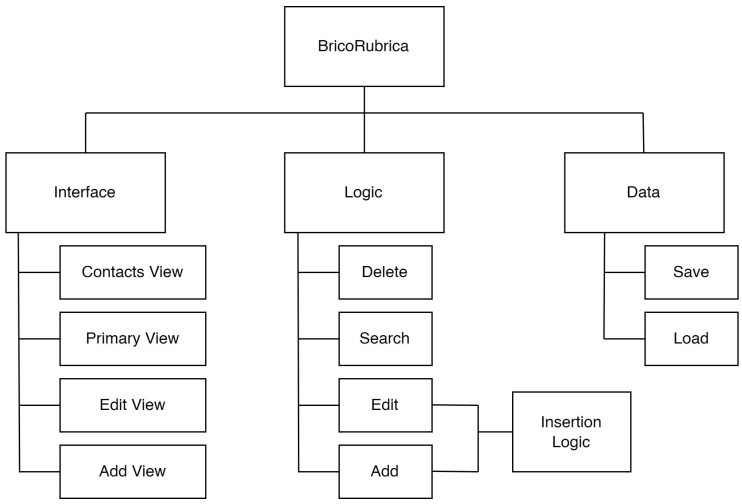

Figure 3: OO Decomposition - 3^{rd} step

2.2 Packages Diagram

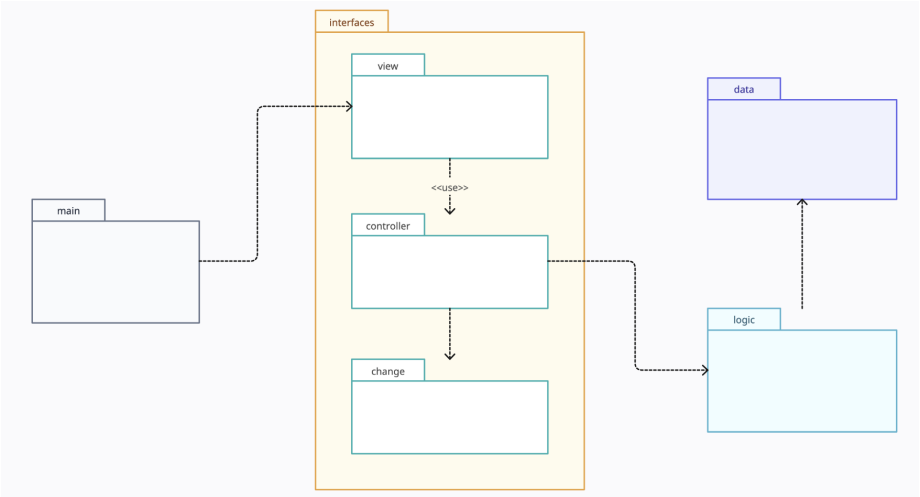

Figure 4: Packages Diagram

Il sistema risulta diviso su un totale di 4 packages contenuti in *it.unisa.diem.gruppo*9. La divisione (vedi Figure 4 - Packages Diagram) rispetta quanto visto in Figure 3 della sezione Module Design, con un package di interfaccia detto *interfaces*, un package di logica detto *logic*, un package di gestione file detto *data* e un package aggiuntivo detto *main*, contenente solo la classe BricoRubrica per l'avvio del programma.
Il package *interfaces* è il più complesso dei 4 poiché contiene i tre sottopackage:

- *change*
- *controller*
- *view*

La scelta richiama lo stile architetturale MVC, dove: il package *view* equivale alla View con i file FXML di interfaccia, il package *controller* equivale al Model con tutti i controller associati ai file FXML e il package *change* equivale al Controller con la classe ChangeView che regola le interazioni tra i vari file e i controller ad essi associati.

2.3 Class Diagrams

Per una vista d'insieme dei diagramma delle classi e per un diagramma delle classi più ordinato, visuallizare la versione alternativa proposta in `https://github.com/simooo6/Software_Engineering_Project/tree/master/Design`.

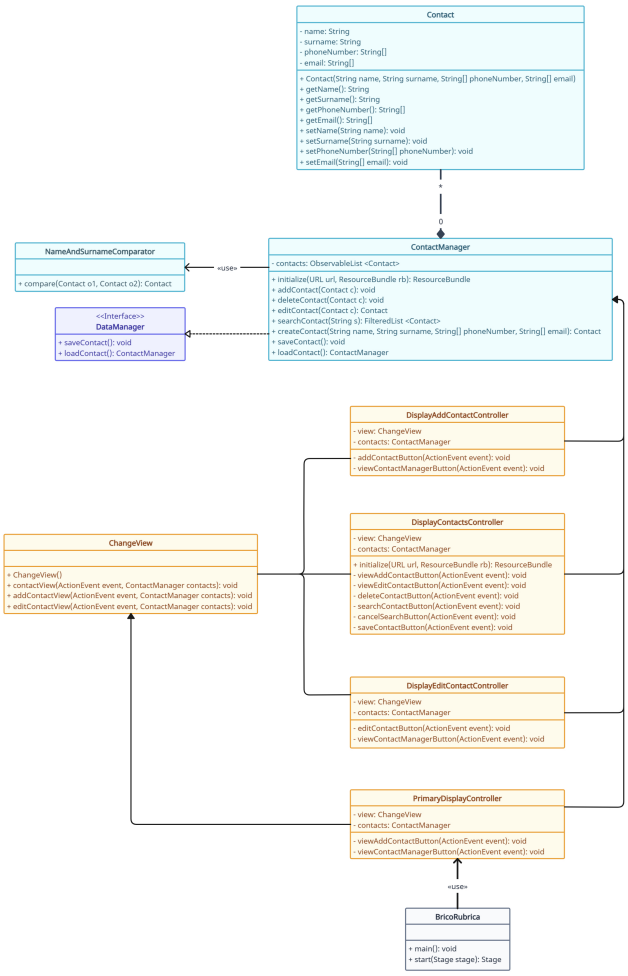

Figure 5: diagramma delle classi generale. I colori che contraddistinguono le classi sono indicatori del package di appartenenza; la legenda dei colori è quindi deducibile dal Packages Diagram in Figure 4.

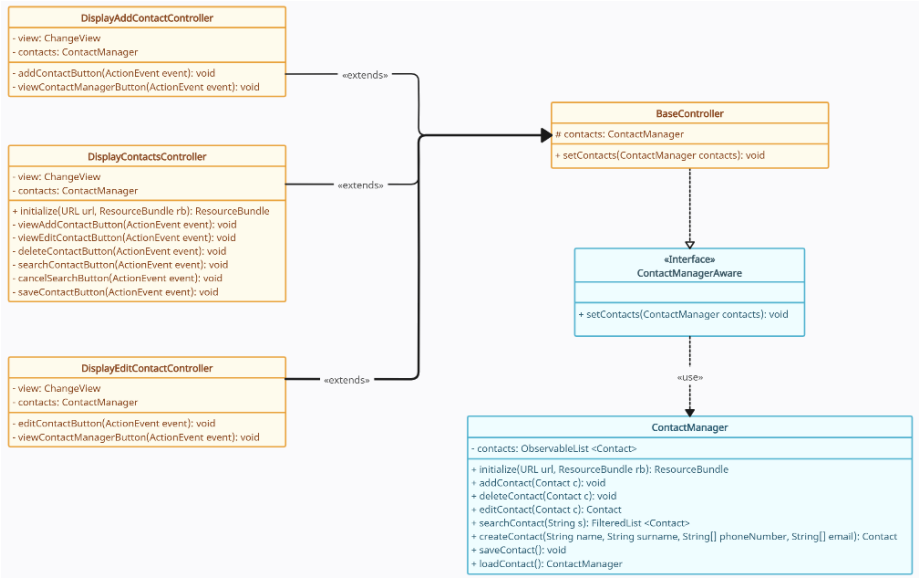

Figure 6: diagramma delle classi con attenzione alle relazioni tra i controller delle view FXML e la logica di gestione contatti. Il diagramma mette in evidenza il contratto pubblico tra la lista dei contatti contenuta in $ContactManager$ ed i controller delle sezioni di interfaccia che si occupano di: visualizzazione della lista dei contatti ($DisplayContactsController$), modifica di un contatto ($DisplayEditContactController$) e aggiunta di un contatto ($DisplayAddContactController$).

About Java Classes

Il sistema è composto da diverse classi e interfacce che cooperano per gestire una rubrica di contatti in modo modulare ed efficiente. Di seguito una breve descrizione delle componenti principali:

- **Contact**: Classe che rappresenta i dati di un contatto, come nome, cognome, numero di telefono ed email. Offre metodi getter e setter per accedere e modificare i dati.
- **ContactManager**: Classe che gestisce la logica principale della rubrica, inclusa la creazione, modifica, ricerca, cancellazione di contatti e salvataggio/caricamento della rubrica. Utilizza un $ObservableList < Contact >$ per mantenere la lista dei contatti.
- **NameAndSurnameComparator**: Classe che implementa la logica di confronto tra contatti basandosi sul nome o sul cognome, in accordo a quanto già definito nella fase di analisi dei requisiti, al fine di mantenere sempre ordinata la lista. Vale a dire che tale ordine deve essere rispettato durante l'esecuzione di qualsiasi azione che interessi la lista.

- **ContactManagerAware** (interfaccia): Definisce un contratto per le classi che necessitano di un'istanza di $ContactManager$. Contiene il metodo $setContacts(ContactManagercontacts)$ per l'iniezione delle dipendenze.
- **BaseController**: Classe base che implementa $ContactManagerAware$. Fornisce il metodo $setContacts$ per inizializzare il $ContactManager$ ed è estesa da tutti i controller.
- **DisplayContactsController**, **DisplayEditContactController**, **DisplayAddContactController**: Controller specifici per la gestione delle diverse funzionalità dell'interfaccia grafica (visualizzazione, modifica e aggiunta di contatti). Estendono $BaseController$.
- **ChangeView**: Classe responsabile della gestione delle transizioni tra le diverse schermate dell'interfaccia grafica.
- **PrimaryDisplayController**: Controller che gestisce la schermata principale e funge da punto di accesso per le funzionalità della rubrica.
- **DataManager**(interface): Rappresenta un interfaccia per la gestione dei contatti presenti all'interno dell'elenco, e per le operazioni che verranno eseguite su di essi, in particolar modo le operazioni gestite sono salvataggio e caricamento dell'elenco dei contatti su e da un file esterno. La classe che implementa questa interfaccia, tramite i metodi della stessa avrà quindi la possibilità di gestire il caricamento dell'elenco da un file esterno e il salvataggio di eventuali modifiche apportate sempre sullo stesso file esterno

Cohesion and Coupling

Coesione Il sistema raggiunge un livello di coesione **funzionale**, il più alto nella scala qualitativa. Le responsabilità delle classi sono ben definite e focalizzate:

Classe/Componente	Valutazione della Coesione
Contact	Alta coesione: si occupa esclusivamente della gestione dei dati di un singolo contatto.
ContactManager	Alta coesione: gestisce tutte le operazioni sui contatti e mantiene la lista coerente.
BaseController e derivati	Alta coesione: ogni controller è responsabile di un'area funzionale specifica della GUI.
ChangeView	Alta coesione: si occupa unicamente della navigazione tra schermate.

Table 1: Analisi della coesione per componente

Questa coesione funzionale rende il sistema più comprensibile, manutenibile e riutilizzabile.

Accoppiamento Il sistema presenta un livello di accoppiamento **per dati**, il migliore nella scala qualitativa. Le dipendenze tra le classi sono minime e ben gestite:

Classe/Componente	Valutazione dell'Accoppiamento
ContactManagerAware	Permette ai controller di interagire con il $ContactManager$ senza dipendenze rigide.
BaseController	Riduce l'accoppiamento centralizzando la gestione del $ContactManager$.
ChangeView	Isola la logica di navigazione tra schermate, limitando l'accoppiamento con i controller.
ObservableList	Consente un'interazione reattiva tra la GUI e i dati senza dipendenze complesse.

Table 2: Analisi dell'accoppiamento per componente

L'accoppiamento per dato garantisce che le classi scambino solo informazioni essenziali, migliorando la modularità e riducendo la probabilità di effetti collaterali indesiderati.

Principi di Buon Design Il sistema rispetta diversi principi fondamentali del design software:

- **Single Responsibility Principle (SRP)**: Ogni classe ha una responsabilità unica e ben definita.
- **Open/Closed Principle (OCP)**: Le classi sono aperte all'estensione (ad esempio, tramite ereditarietà o aggiunta di nuovi metodi), ma chiuse alle modifiche, garantendo stabilità del codice esistente.
- **Liskov Substitution Principle (LSP)**: I controller che estendono $BaseController$ possono essere utilizzati come istanze della classe base senza alterare il comportamento del sistema.
- **Dependency Inversion Principle (DIP)**: L'interfaccia $ContactManagerAware$ riduce le dipendenze rigide tra componenti di alto e basso livello, consentendo flessibilità.
- **Separation of Concerns**: La separazione tra modello ($Contact$, $ContactManager$), controller ($BaseController$, controller derivati) e vista ($ChangeView$) è ben definita.

3 Process View

3.1 Sequence Diagrams

Address Book - Visualization

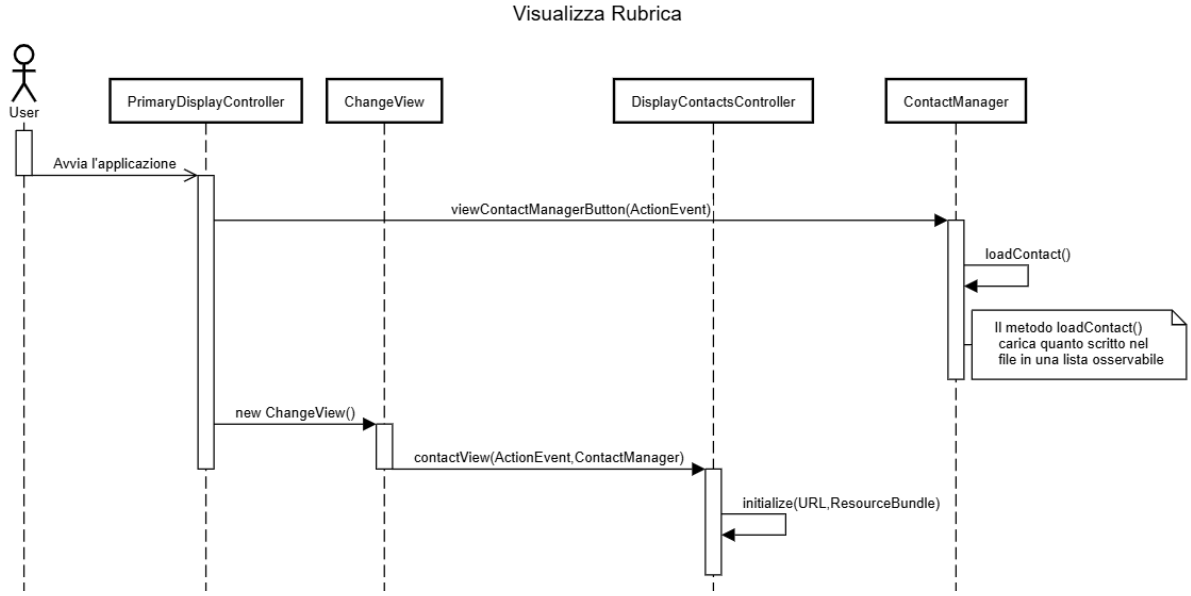

Figure 7: Il diagramma di sequenza in figura descrive come avviene la visualizzazione della rubrica a partire dall'avvio del sistema. La sequenza rappresentata non è l'unica per quanto concerne la visualizzazione della lista dei contatti, ma è sicuramente la principale. In qualunque altro istante l'utente voglia visualizzare la rubrica, le sequenze risulteranno quasi equivalenti a quella diagrammata, a meno dell'utilizzo di un oggetto *PrimayDisplayController* e del metodo *loadContact*() dell'oggetto *ContactManager*.

Contact - Creation

I diagrammi di sequenza nella sezione che segue trattano i due possibili scenari di creazione di un nuovo contatto. Il primo (vedi Figure 8) qualora l'utente scegliesse di creare un nuovo contatto senza visualizzare prima la rubrica e il secondo (vedi Figure 9) qualora l'utente scegliesse di creare un nuovo contatto dopo aver già utilizzato il software e quindi dopo aver avuto già accesso alla lista dei suoi contatti.

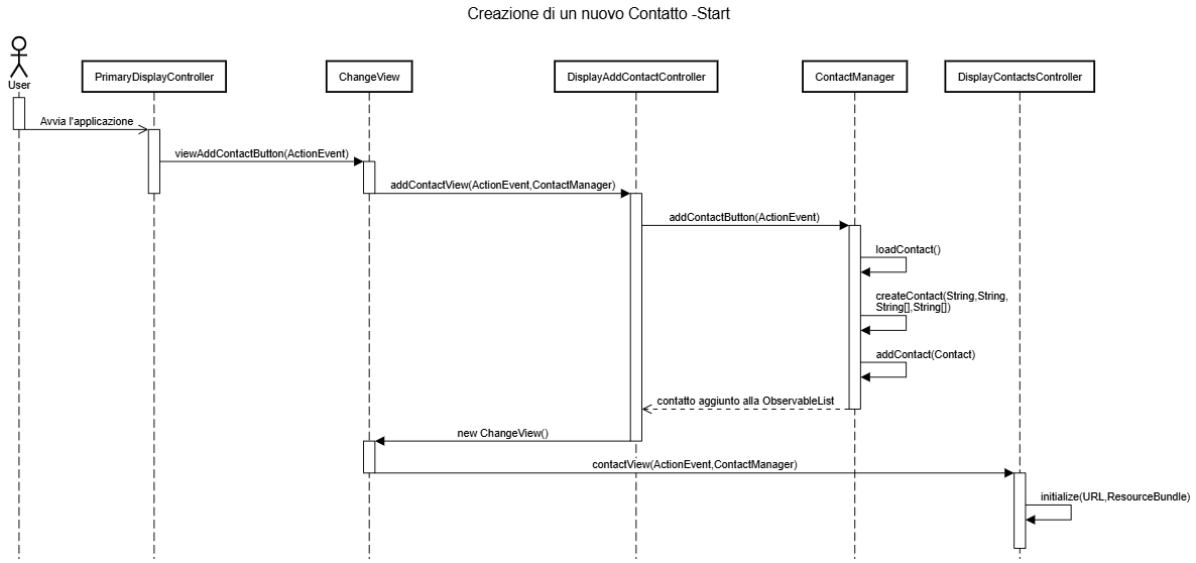

Figure 8: diagramma di sequenza che descrive i messaggi scambiati tra gli oggetti interessati nel flusso di creazione di un nuovo contatto, data la scelta esplicita dell'utente, ad avvio sistema.

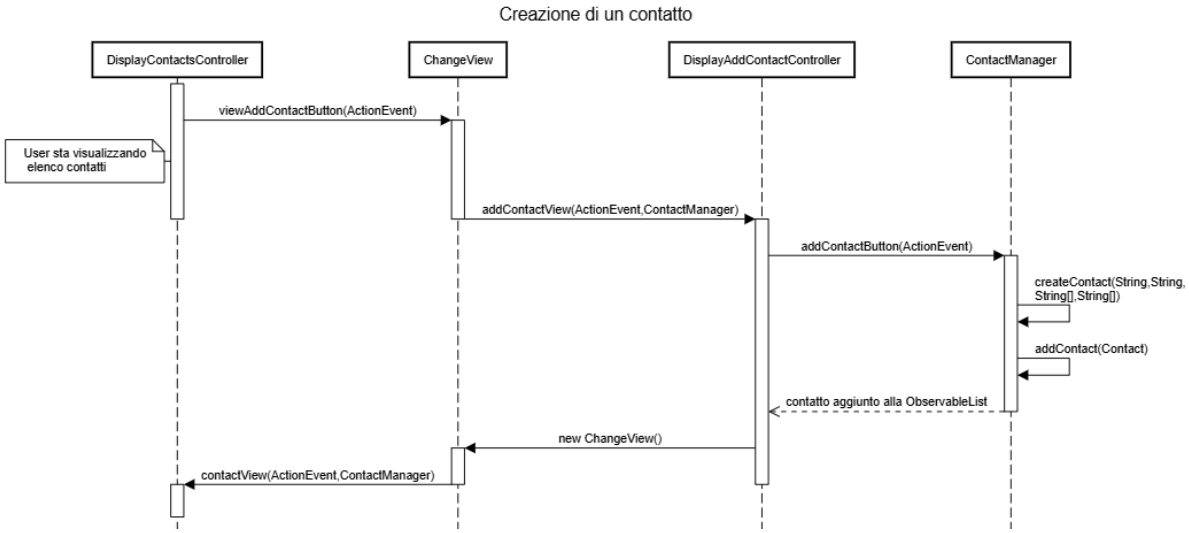

Figure 9: diagramma di sequenza che descrive i messaggi scambiati tra gli oggetti interessati nel flusso di creazione di un nuovo contatto durante una sessione sul software già avviata. L'utente potrebbe aver già svolto una qualsiasi azione con il sistema, ad esempio una di quelle previste dai casi d'uso, compresa la creazione di un contatto ad avvio sistema.

Contact - Editing

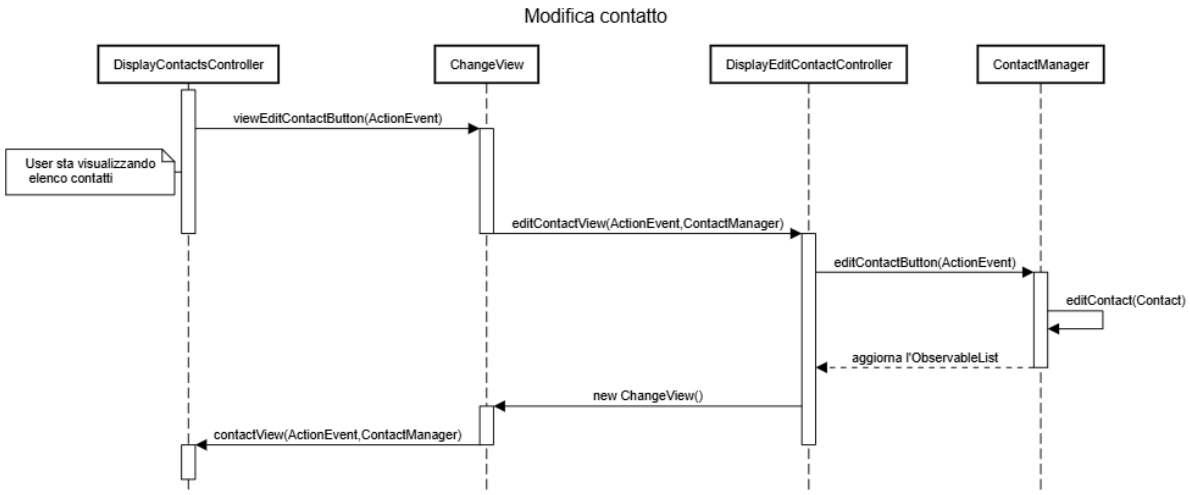

Figure 10: il diagramma rappresenta la sequenza di messaggi necessari per permettere all'utente la modifica di un contatto, tenendo a mente che l'user sta già visualizzando la lista dei contatti. Inoltre, come già chiarito, l'utente deve selezionare uno ed un solo contatto affinché il button che permette la modifica si abiliti; per essere precisi la selezione abilita $viewEditContactButton(ActionEvent)$ nella classe $DisplayContactsController$

Contact - Research

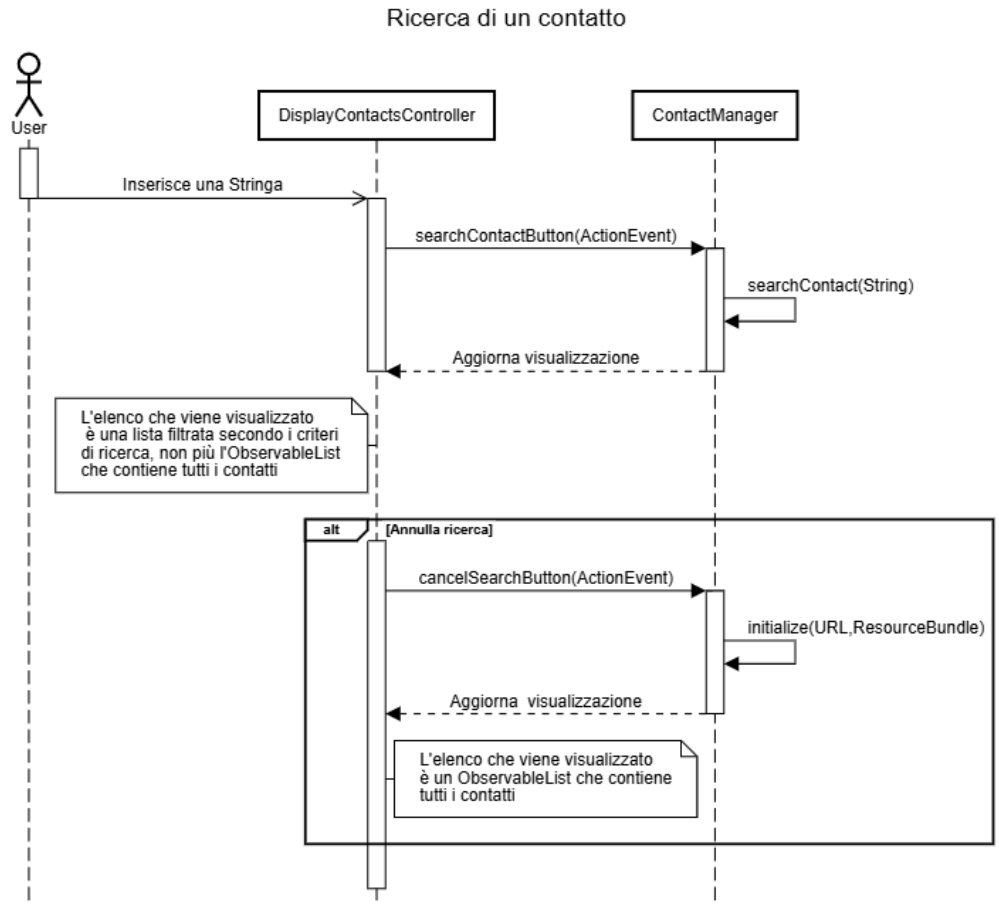

Figure 11: diagramma di sequenza che mappa l'interazione tra gli oggetti *DisplayContactsController* e *ContactManager* per far sì che l'utente, a conferma di ricerca mediante un bottone associato, visualizzi la lista filtrata dei contatti secondo i parametri di ricerca inseriti; in alternativa, come si può vedere dalla sezione *alt* in poi, la visualizzazione della lista di contatti ritorna al suo stato inizale

Contact - Delete

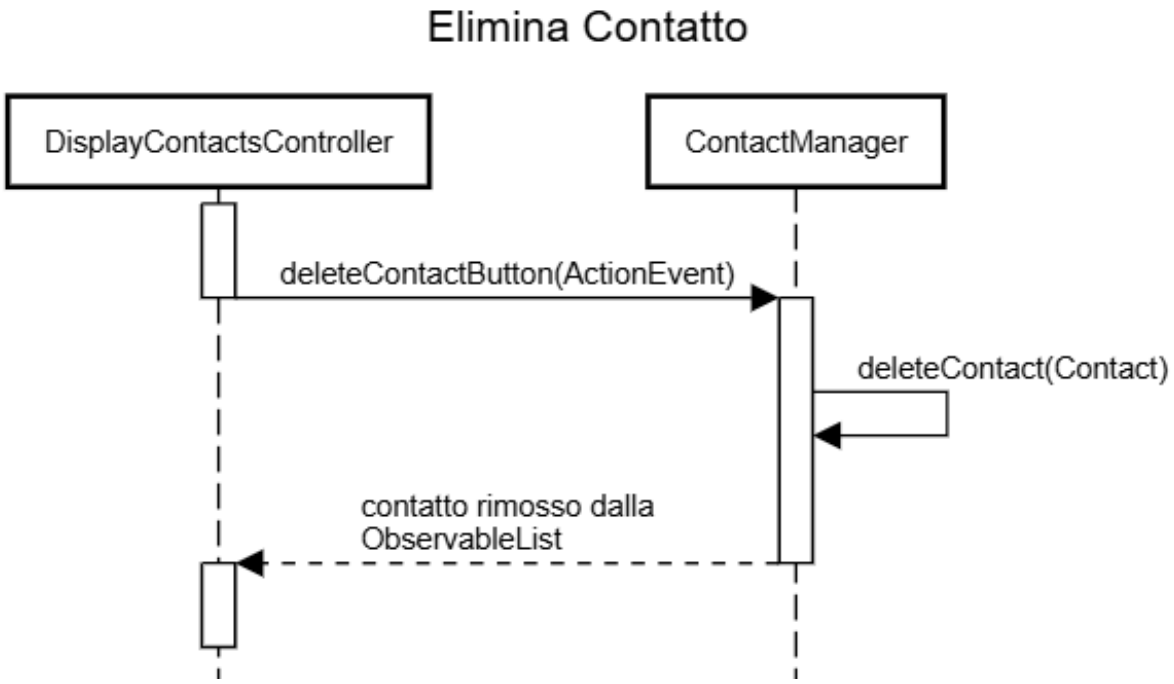

Figure 12: il diagramma rispetta quanto già detto nella sezione edit contact in Figure 10. L'eliminazione infatti, come la modifica, necessita della selezione, prevista e gestita proprio come già descritto

Address Book - Save

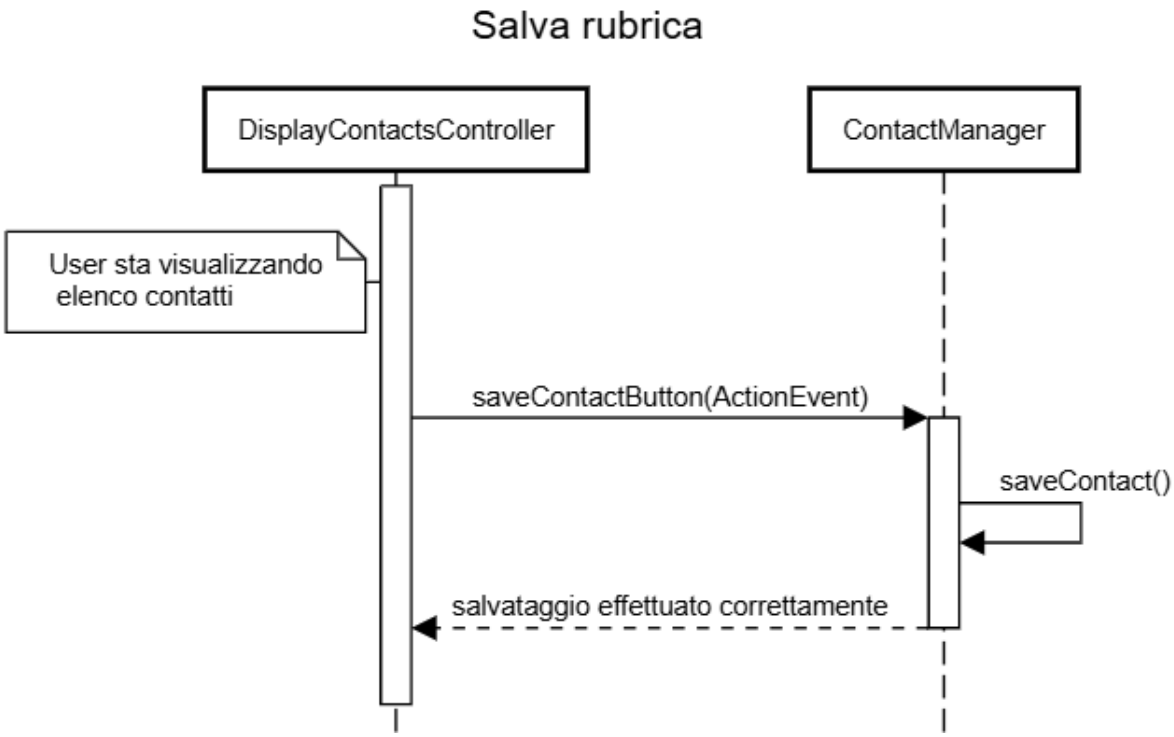

Figure 13: diagramma delle sequenze che interessano il flusso di salvataggio delle modifiche apportate alla rubrica.