Sviluppo di un software di Rubrica telefonica ed e-mail: Design

Gruppo 03:
Iannone Ursula Giovanna, Pagano Maria Pia, Senatore Cory, Stanco Mario

---

1. Diagramma delle classi

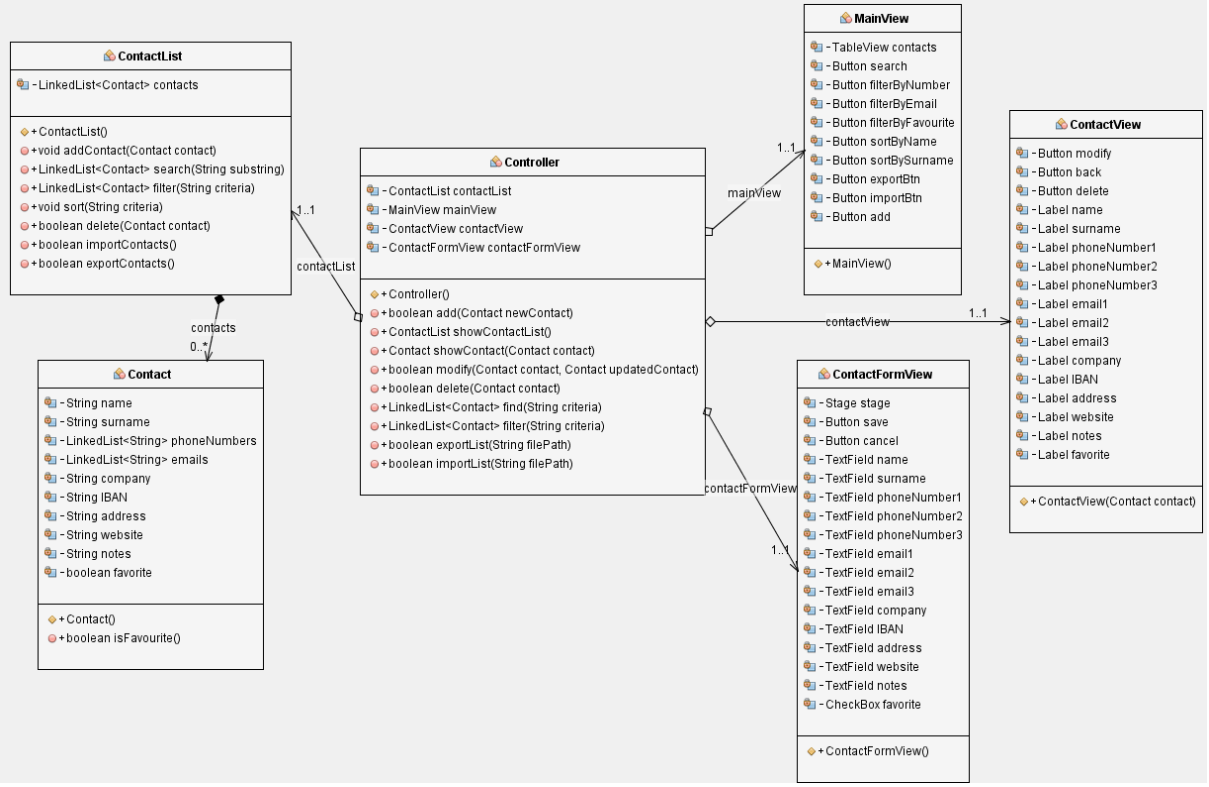
Nella progettazione del sistema è stato adottato il pattern architetturale *MVC* (*Model-View-Controller*), che prevede la suddivisione dell'applicazione in tre componenti: il *Model*, che si occupa dei dati; la *View*, che permette di visualizzare i dati grazie all'interfaccia; il *Controller*, che riceve input e modifica gli altri due componenti di conseguenza.

La scelta di questo stile architetturale risulta particolarmente coerente con l'utilizzo di un linguaggio di programmazione orientato agli oggetti quale *Java*, che mette a disposizione lo strumento *JavaFX* per la cura della *GUI*.

A ciascuno dei componenti sono state assegnate le classi appartenenti in maniera coerente con la risoluzione del problema. In particolare:

- al *Model* appartengono le classi:
    - *Contact*;
    - *ContactList*, che si occupa della gestione dell'insieme dei contatti.
- alla *View* appartengono le classi:
    - *MainView*;
    - *ContactView*;
    - *ContactFormView*;
- al *Controller* appartiene la sola classe
    - *Controller*.

1.1 Diagramma delle classi generato grazie allo strumento automatico EasyUML.



Note: Metodi standard previsti per le classi, come ad esempio i *getter* e *setter*, non sono inseriti intenzionalmente all'interno del diagramma per preservare la sua chiarezza e leggibilità, poiché non rappresentano elementi caratterizzanti per la progettazione.

1.2 Descrizione delle classi nel dettaglio.

Model

La classe *Contact* si occupa della rappresentazione del singolo contatto all'interno della rubrica. Gli attributi della classe corrispondono ai dati identificativi di un singolo contatto all'interno della rubrica quali nome, cognome, numero di telefono; i metodi della classe sono i getter e setter di ciascun attributo che permettono di prelevare o modificare i valori degli attributi stessi.

La classe *ContactList* si occupa della gestione della lista dei contatti, l'unico attributo è infatti la lista contenente i contatti presenti in rubrica. I metodi appartenenti alla classe invece si occupano di gestire tutte le operazioni effettuabili sull'intera rubrica:

- i metodi *add*, *modify* e *delete* restituiscono un valore booleano pari a 1 se l'operazione richiesta va a buon fine;
- il metodo *sort* opera sull'intera rubrica e permette di ordinare i contatti in base al criterio selezionato;
- il metodo *filter* opera sull'intera rubrica permettendo di filtrare i contatti in base al criterio selezionato e restituisce una lista contenente un sottoinsieme dei contatti;
- il metodo *search* prende in input una stringa e restituisce una sottolista dei contatti che contengono la stringa nei campi di nome/cognome.
- i metodi *import* ed *export* restituiscono un valore booleano pari a 1 se l'operazione di importazione ed esportazione richiesta va a buon fine;

View

La classe *MainView* si occupa della visualizzazione della homepage del software, tra gli attributi troviamo la tabella che permette la visualizzazione di tutti contatti in rubrica e i tasti presenti sulla schermata principale che rappresentano le funzionalità disponibili.

La classe *ContactView* si occupa della rappresentazione della schermata contenente dettagli del contatto: gli attributi della classe sono delle *Label* che rappresentano le info e che assumeranno il valore associato al contatto selezionato, sono presenti, inoltre, dei tasti che permettono la modifica o l'eliminazione del contatto e un tasto per tornare alla *MainView*.

La classe *ContactFormView* si occupa della rappresentazione grafica della schermata visualizzata alla pressione dei tasti modifica o aggiungi contatto: gli attributi della classe sono dei *TextFields* modificabili vuoti nel caso di nuovo contatto o che assumeranno il valore del campo del contatto nel caso di "modifica contatto".

Controller

Il *controller* è il componente che si occupa delle interazioni tra la *view* e il *model*: ad esempio interpreta gli input dell'utente sulla *View,* come la pressione dei tasti, quindi invoca i metodi appropriati del Model e successivamente si occupa del notificare la *MainView* della modifica al modello.

1.3 Coesione ed accoppiamento

La scelta del modello *MVC* riflette anche la necessità di scomporre il sistema in moduli caratterizzati da un'alta *coesione*, che misura il legame tra le parti interne ad uno stesso modulo, e da un basso *accoppiamento*, che valuta l'interdipendenza tra moduli diversi.

Coesione
La divisione in classi che hanno responsabilità differenti permette di raggruppare compiti coerenti fra loro. In particolare, la coesione si mantiene di tipo comunicazionale all'interno delle classi.

- Model:
  Le classi contenute all'interno del package mostrano un buon livello di coesione funzionale:
  la classe ContactList offre metodi che condividono lo scopo di manipolare la lista dei contatti stessa ad eccezione delle funzioni di importazione ed esportazione; la classe Contact, allo stesso modo, si focalizza esclusivamente sulla gestione delle proprietà individuali di un contatto non contenendo metodi utili a manipolare dati esterni.
- Controller:
  La classe Controller presenta un livello di coesione logica poiché al suo interno sono presenti funzionalità che, pur appartenendo allo stesso dominio applicativo, non sono collegate tra loro in modo sequenziale o comunicativo.
- View:
  Le classi ContactFormView e ContactView presenti all'interno del package mostrano un livello di coesione di tipo funzionale in quanto tutte le funzionalità pongono il focus su compiti specifici: la prima sulla creazione o modifica di un contatto, la seconda sulla visualizzazione dei dettagli di un singolo contatto.
  La classe MainView, invece, mostra un livello di coesione logica in quanto raggruppa operazioni diverse legate logicamente alla gestione della lista dei contatti, i vari metodi però non sono legati o dipendenti uno l'uno dall'altro.

Accoppiamento
L'accoppiamento tra le classi di un sistema ne descrive il livello di dipendenza e il modo in cui interagiscono, l'approccio di tipo MVC fa sì che, dividendo il sistema in tre componenti principali, Model e View non interagiscano direttamente. La classe Controller funge da intermediario garantendo così un livello basso di accoppiamento.

- Model:
  La classe Contact presenta l'accoppiamento per dati con altre classi in quanto interagisce con le altre classi solo per immagazzinare e fornire informazioni a componenti esterni senza avere dipendenze o interazioni complesse con altre classi.
  La classe ContactList presenta un accoppiamento per contenuti in quanto l'attributo lista è una raccolta di oggetti di tipo Contact ai quali accede per utilizzare i metodi implementati.
- Controller:
  La classe presenta un accoppiamento per controllo in quanto agisce da intermediario tra gli oggetti della classe *View e Model*, gestendo e passando

le informazioni di controllo che determinano la logica di funzionamento delle classi destinatarie.

- View
  Le classi del package View presentano un accoppiamento per dati in quanto le interazioni con le altre classi si limitano allo scambio di informazioni, come il passaggio o la restituzione di dati senza influire direttamente sul comportamento o sulla logica delle classi con cui "collaborano".

---

2. Diagrammi di sequenza

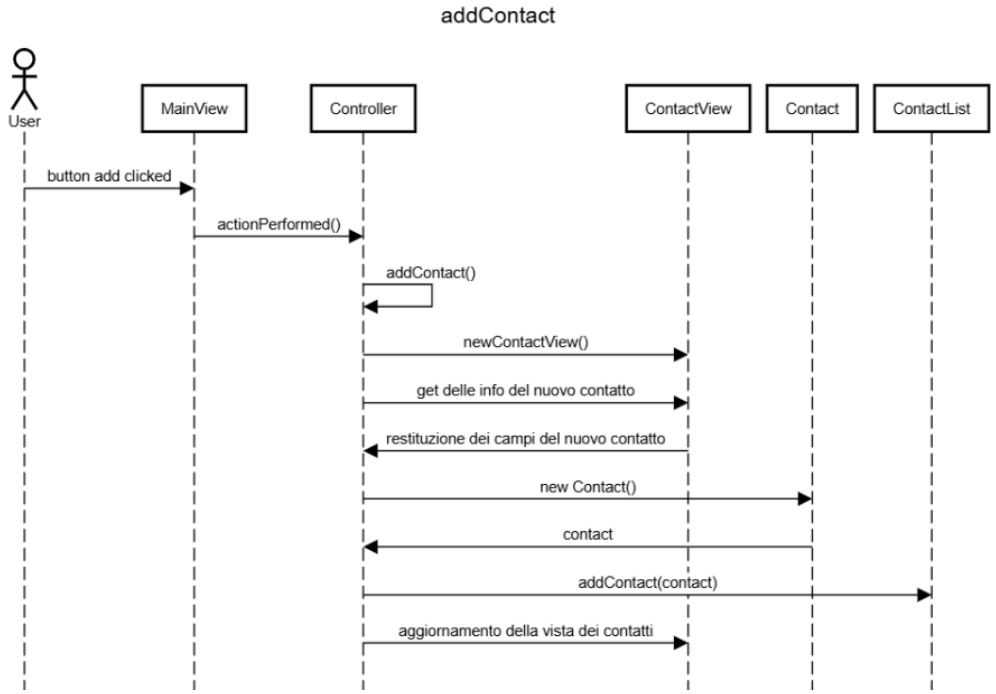
I diagrammi di sequenza sono un particolare tipo di diagrammi di interazione. Sono di particolare utilità perché permettono di schematizzare il comportamento dei componenti del sistema in un singolo scenario.

All'interno di questi diagrammi, i partecipanti coinvolti sono disposti in alto, da sinistra a destra, mentre il tempo avanza verticalmente verso il basso.

Le interazioni rappresentate nei diagrammi derivano dalla precedente fase di analisi dei requisiti, durante la quale i casi d'uso hanno evidenziato le principali funzionalità messe a disposizione dell'utente.

1. Inserisci contatto

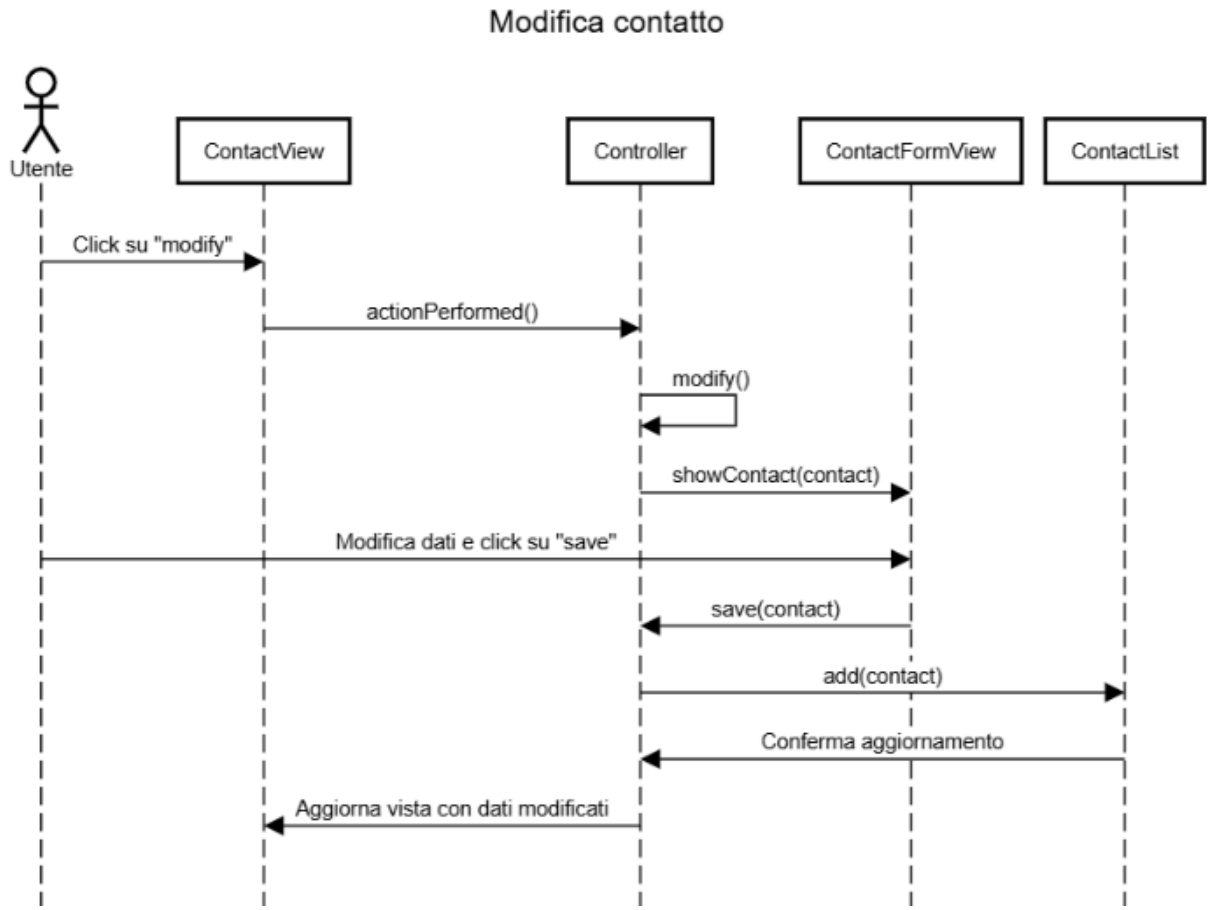
L'attore "Utente" interagisce con la MainView della rubrica cliccando su "aggiungi".
Controller richiama il metodo *addContact()* che a sua volta chiama *newContactView()* della classe *ContactVi*ew che restituisce la schermata di inserimento contatto. Vengono così restituiti i campi compilati con i dati del nuovo contatto. *Controller* procederà implementando *contact()* nella classe *Contact* e *addContact(contact)* in *ContactList* che aggiunge il nuovo contatto all'interno della rubrica, viene quindi aggiornata la vista dei contatti.




```
title addContact
User->MainView: button add clicked
MainView->Controller: actionPerformed()
Controller->Controller: addContact()
Controller->ContactView: newContactView()
Controller->ContactView: get delle info del nuovo contatto
ContactView->Controller: restituzione dei campi del nuovo contatto
Controller->Contact:new Contact()
Contact->Controller: contact
Controller->ContactList:addContact(contact)
Controller->ContactView: aggiornamento della vista dei contatti
```

2. Modifica contatto

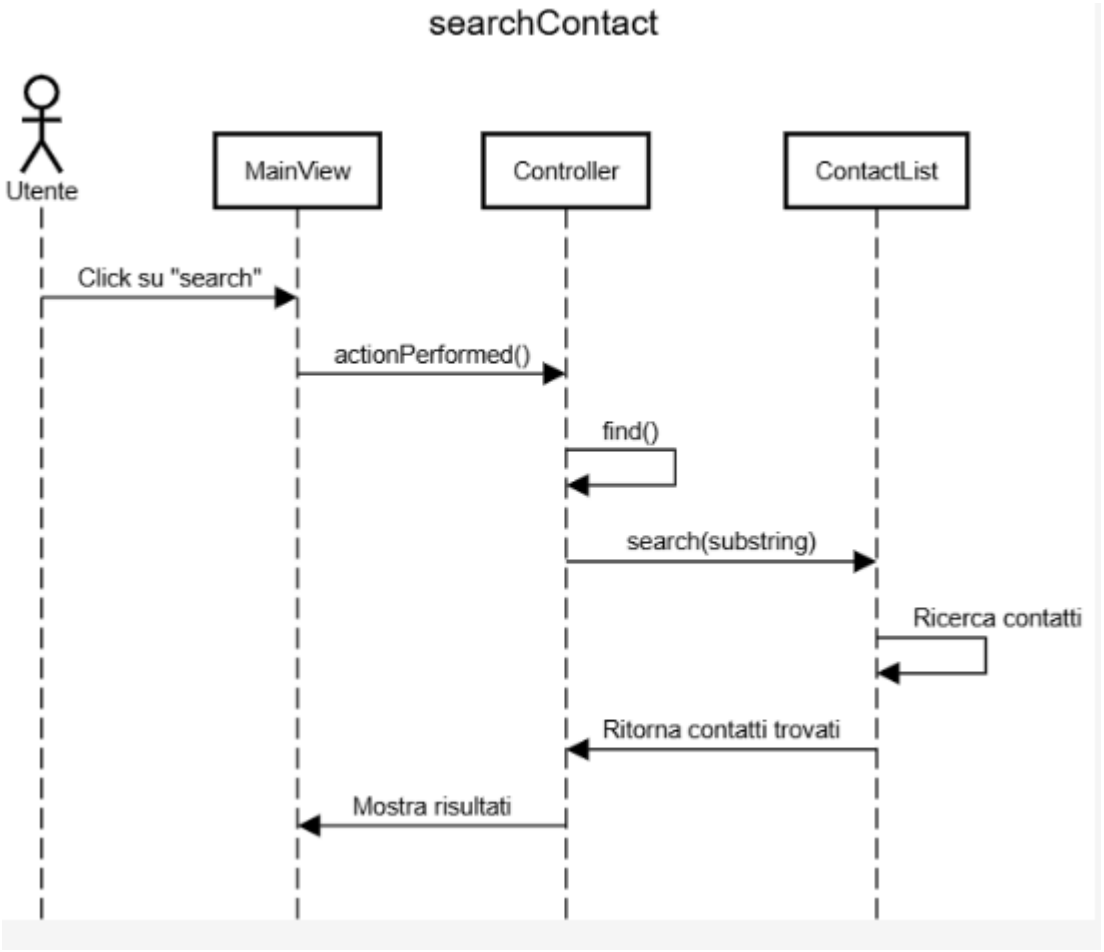
L'attore "Utente" interagisce con la *ContactView* del contatto cliccando su "modifica".
Controller richiama il metodo *showContact(contact)* di *ContactFormView* che mostra all'utente il contatto da modificare.
L'attore "Utente" interagisce con la *ContactFormView* cliccando su "salva".
Controller procederà implementando *addContact(contact)* in *ContactList* che aggiunge il nuovo contatto all'interno della rubrica e conferma l'aggiornamento al *Controller*, viene quindi aggiornata la vista del contatto modificato in *ContactView.*




```
title Modifica contatto
actor Utente
Utente -> ContactView: Click su "modify"
ContactView->Controller: actionPerformed()
Controller->Controller: modify()
Controller -> ContactFormView: showContact(contact)
Utente -> ContactFormView: Modifica dati e click su "save"
ContactFormView -> Controller: save(contact)
Controller -> ContactList: add(contact)
ContactList -> Controller: Conferma aggiornamento
Controller -> ContactView: Aggiorna vista con dati modificati
```

3. Ricerca contatto

L'attore "Utente" interagisce con la *MainView* della rubrica cliccando su "cerca" e specificando una sottostringa iniziale del nome o del cognome.
Controller richiama il metodo *find()* che a sua volta chiama *search(substring)* della classe *ContactList* che seleziona i contatti contenenti la sottostringa inserita all'interno del campo cognome/nome e restituisce una lista che li contiene e che verrà visualizzata da ContactListView all'interno della MainView.

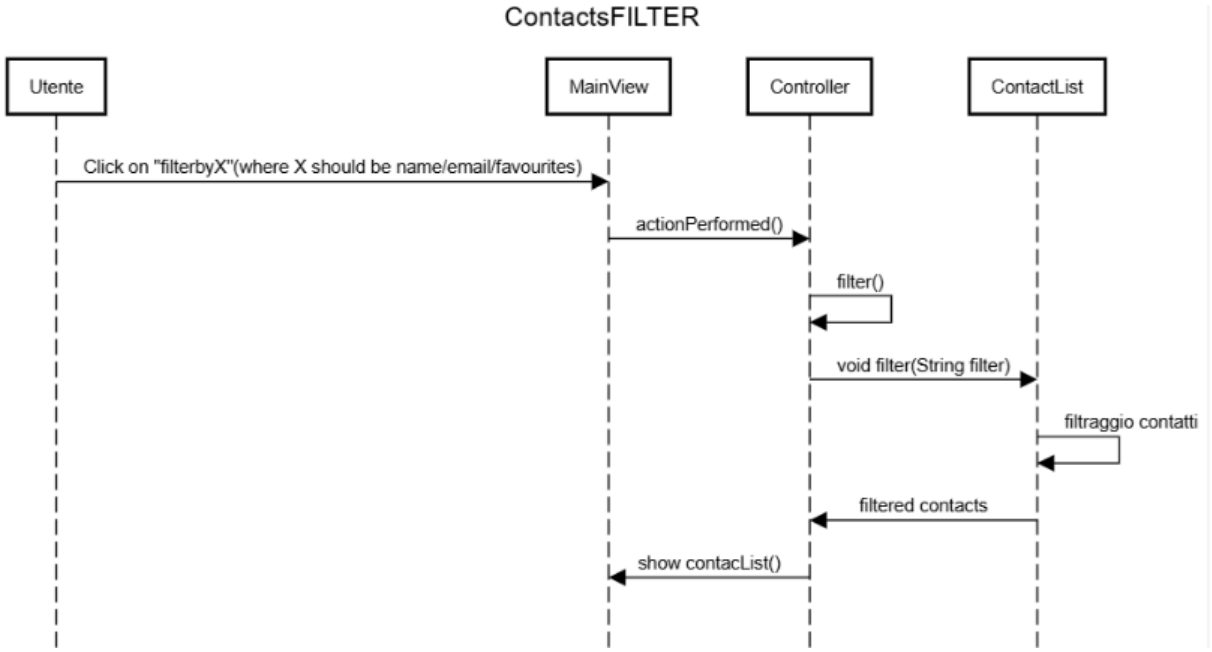



```
title searchContact

actor Utente
Utente-> MainView: Click su "search"
MainView->Controller: actionPerformed()
Controller->Controller: find()
Controller -> ContactList: search(substring)
ContactList -> ContactList: Ricerca contatti
ContactList -> Controller: Ritorna contatti trovati
Controller ->MainView: Mostra risultati
```

4. Filtra

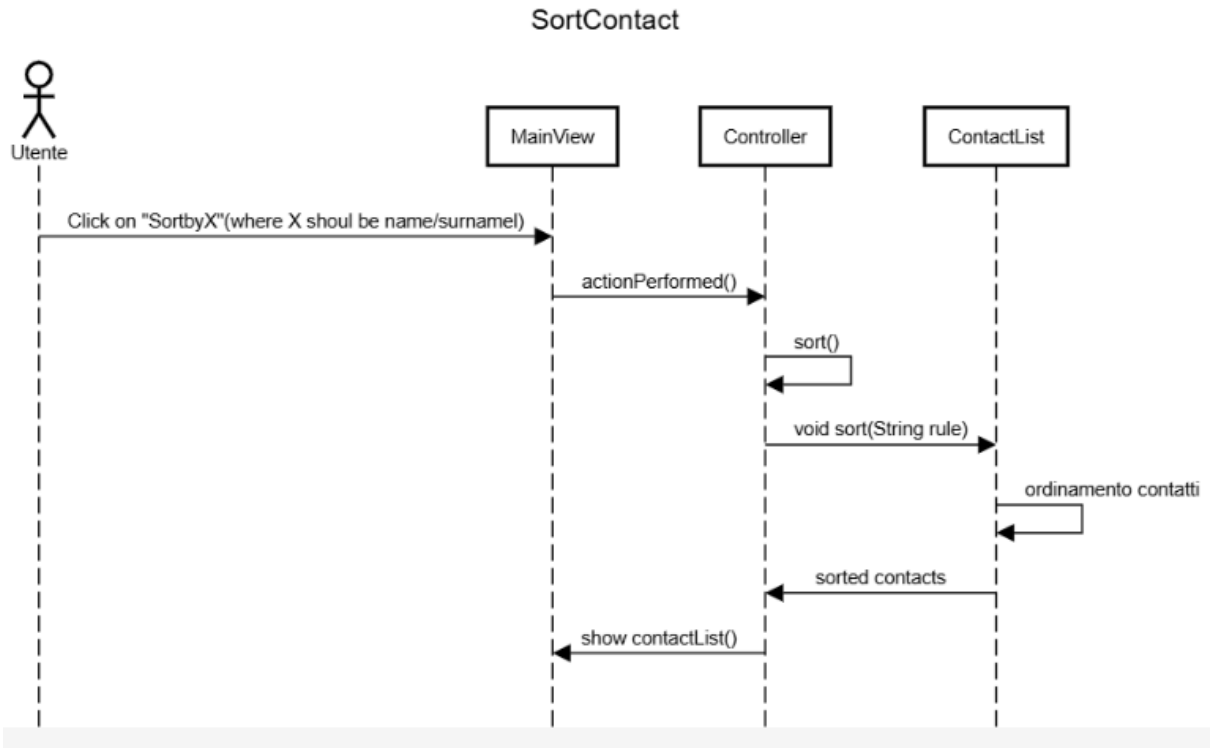
L'attore "Utente" interagisce con la *MainView* della rubrica cliccando su "filtra" ed inserendo la propria preferenza fra nome,email,preferiti .
Controller richiama il metodo *filter()* che a sua volta chiama f*ilter(String filter)* della classe *ContactList* che restituisce i contatti filtrati, viene quindi mostrata la schermata con i contatti filtrati.




```
title ContactsFILTER
Utente->MainView:Click on "filterbyX"(where X should be name/email/favourites)
MainView->Controller: actionPerformed()
Controller->Controller:filter()
Controller->ContactList: void filter(String filter)
ContactList->ContactList:filtraggio contatti
ContactList->Controller:filtered contacts
Controller->MainView: show contacList()
```

5. Ordina

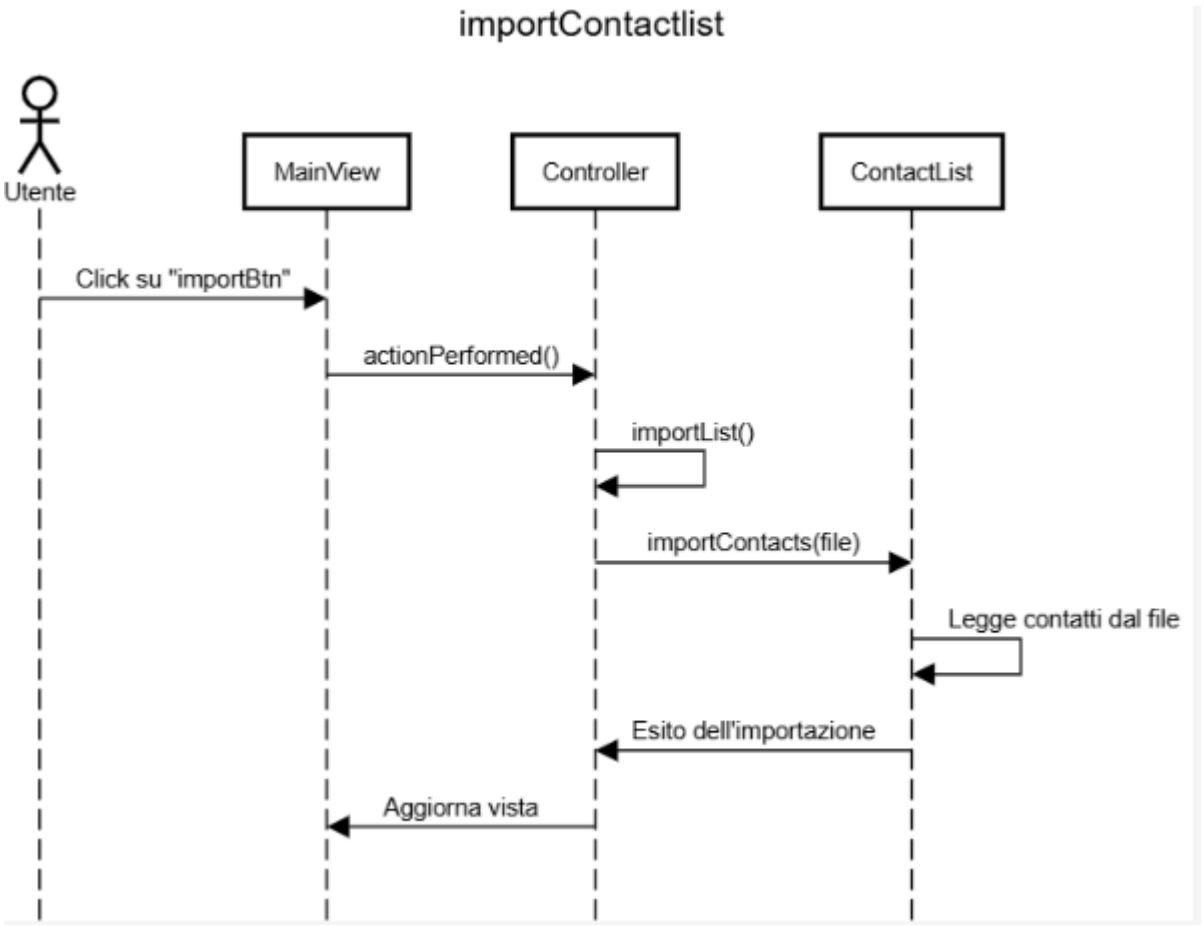
L'attore "Utente" interagisce con la *MainView* della rubrica cliccando su "ordina" ed inserendo la propria preferenza fra nome e cognome.
Controller richiama il metodo *sort()* che a sua volta chiama sort*(String rule)* della classe *ContactList* che restituisce i contatti ordinati, viene quindi aggiornata la schermata con i contatti ordinati.




```
title SortContact
actor Utente
Utente->MainView:Click on "SortbyX"(where X shoul be name/surnamel)
MainView->Controller: actionPerformed()
Controller->Controller:sort()
Controller->ContactList: void sort(String rule)
ContactList->ContactList:ordinamento contatti
ContactList->Controller: sorted contacts
Controller->MainView: show contactList()
```

6. Importa

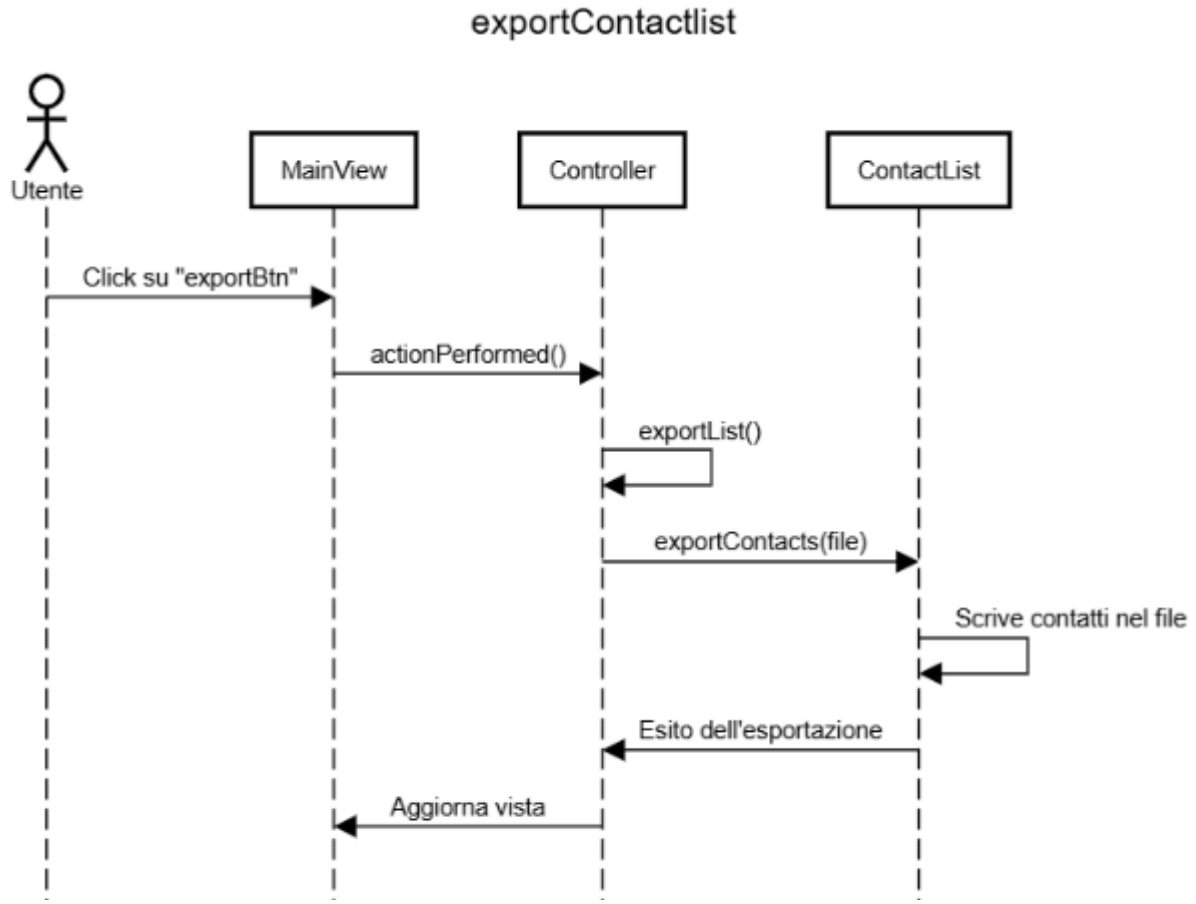
L'attore "Utente" interagisce con la *MainView* della rubrica cliccando su "importa". *Controller* richiama il metodo *importList()* che a sua volta chiama *importContacts(file)* della classe *ContactList* che legge i contatti e restituisce l'esito dell'importazione. Viene quindi aggiornata la vista dei contatti.




```
title importContactlist
actor Utente
Utente -> MainView: Click su "importBtn"
MainView->Controller: actionPerformed()
Controller-> Controller: importList()
Controller -> ContactList: importContacts(file)
ContactList -> ContactList: Legge contatti dal file
ContactList -> Controller: Esito dell'importazione
Controller -> MainView: Aggiorna vista
```

7. Esporta

L’attore “Utente” interagisce con la *MainView* della rubrica cliccando su “esporta”. *Controller* richiama il metodo *exportList()* che a sua volta chiama *exportContacts()* della classe *ContactList* che scrive i contatti nel file e restituisce l'esito dell'esportazione. Viene quindi aggiornata la vista dei contatti.

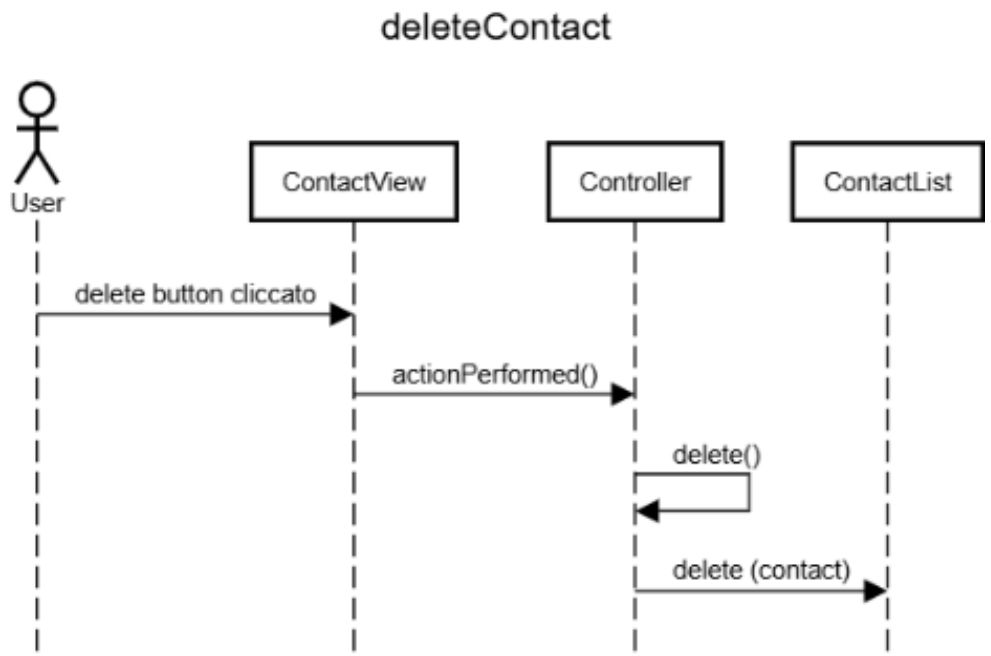



```
title  exportContactlist
actor Utente
Utente -> MainView: Click su "exportBtn"
MainView->Controller: actionPerformed()
Controller -> Controller: exportList()
Controller -> ContactList: exportContacts(file)
ContactList -> ContactList: Scrive contatti nel file
ContactList -> Controller: Esito dell'esportazione
Controller -> MainView: Aggiorna vista
```

8. Elimina contatto

L'attore utente interagisce con la *ContactView* cliccando su "elimina"per procedere con l'eliminazione del contatto.
Il controller cattura l'evento di pressione del tasto quindi viene invocato il metodo delete() della classe *Controller* che a sua volta chiama il metodo delete della classe

ContactList. Il contatto viene quindi rimosso dalla rubrica e, dopo aver notificato il cambiamento al Controller quest'ultimo restituisce la ContactList aggiornata alla ContactView.




```
title deleteContact
User->ContactView: delete button cliccato
ContactView->Controller: actionPerformed()
Controller->Controller: delete()
Controller->ContactList: delete (contact)
```