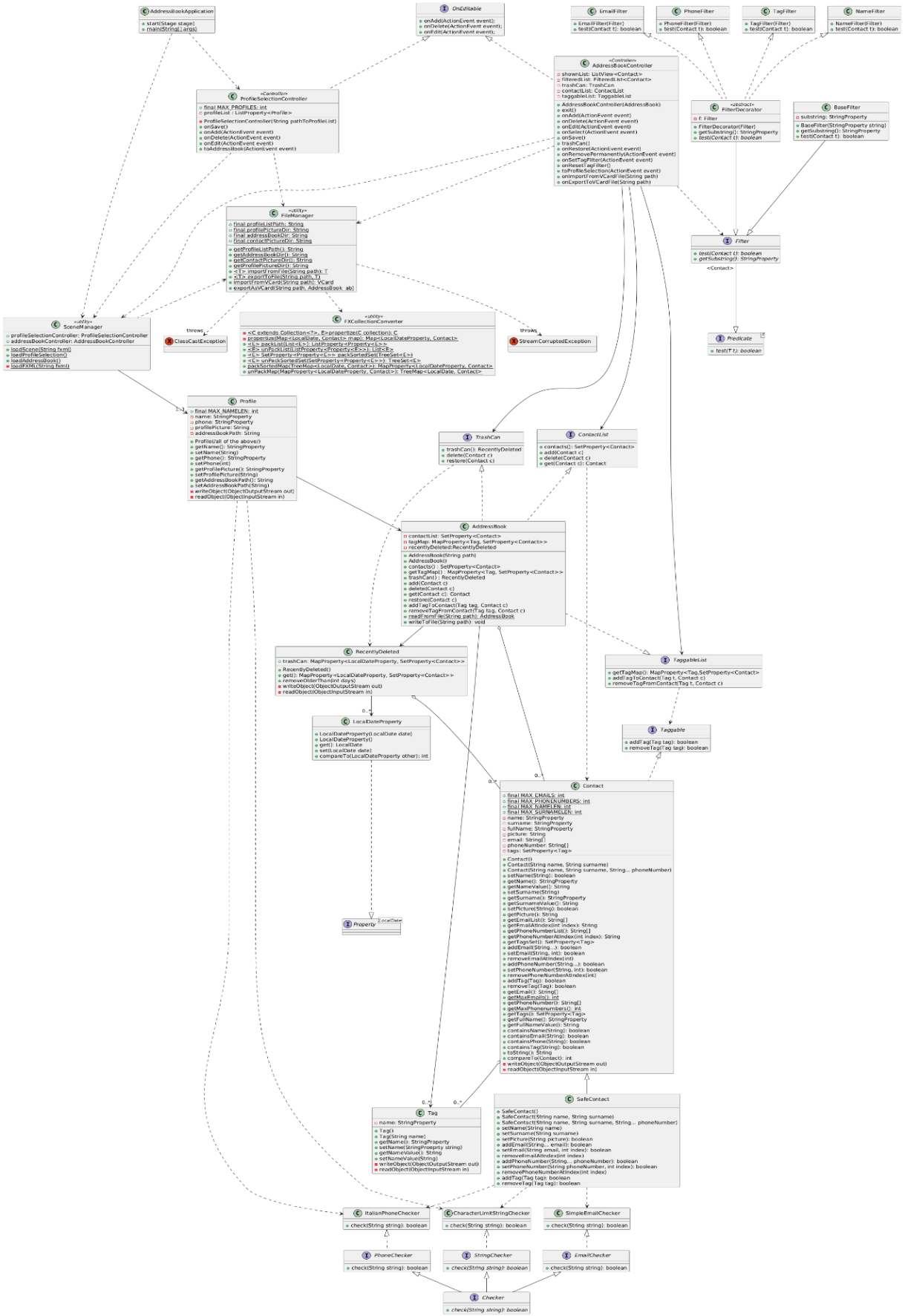
Design & Design di dettaglio

Diagramma delle Classi



Analisi del diagramma delle classi in termini di coesione ed accoppiamento:

1. **AddressBookApplication:**
    - Coesione: Temporale - Le funzionalità (start e main) sono utilizzate all'inizio del programma per avviare l'applicazione.
    - Accoppiamento: Per dati - Passa dati necessari tra i moduli, ad esempio Stage per inizializzare l'app.
2. **SceneManager:**
    - Coesione: Procedurale - Include operazioni correlate per la gestione delle scene (loadScene, loadProfileSelection ecc.).
    - Accoppiamento: Per controllo - Coordina la visualizzazione delle scene, ricevendo ordini dai Controller e comandando le View.
3. **FileManager:**
    - Coesione: Logica - Contiene operazioni simili (gestione file) su dati non necessariamente correlati (profili, address book, contatti).
    - Accoppiamento: Per dati - Riceve percorsi di file e oggetti strettamente necessari per l'import/export.
4. **FXCollectionConverter:**
    - Coesione: Logica - Implementa operazioni simili per convertire tipi di collezioni.
    - Accoppiamento: Per dati - Si occupa esclusivamente di manipolare e restituire collezioni di dati.
5. **RecentlyDeleted:**
    - Coesione: Comunicazionale - Tutte le operazioni gestiscono dati relativi ai contatti eliminati (aggiunta, rimozione, salvataggio).
    - Accoppiamento: Per dati - Usa contatti come input/output per le operazioni.
6. **Profile:**
    - Coesione: Comunicazionale - Gestisce dati di profilo come nome, telefono e immagine.
    - Accoppiamento: Per dati - Utilizza solo i dati strettamente necessari per rappresentare e manipolare un profilo.
7. **AddressBook:**
    - Coesione: Comunicazionale - Le operazioni gestiscono un insieme di contatti e metadati associati (tag, ricerca, eliminazioni).
    - Accoppiamento: Per dati - Scambia solo le informazioni necessarie con RecentlyDeleted, Contact, e altri moduli.
8. **AddressBookController:**
    - Coesione: Comunicazionale - Implementa funzionalità che vengono usate per controllare le interazioni dell'utente con l'address book.
    - Accoppiamento: Per controllo - Passa eventi per controllare la logica di altre classi.
9. **Tag:**
    - Coesione: Comunicazionale - Tutte le operazioni manipolano lo stesso dato (il nome del tag).

- Accoppiamento: Per dati - Richiede solo stringhe per rappresentare il nome di un tag.

10. **Contact:**
    - Coesione: Comunicazionale - Le funzionalità lavorano sui dati di un contatto (nome, email, telefono, ecc.).
    - Accoppiamento: Per dati - Utilizza proprietà strettamente necessarie per rappresentare un contatto.
11. **SafeContact:**
    - Coesione: Comunicazionale - Le funzionalità lavorano sui dati di un contatto (nome, email, telefono, ecc.).
    - Accoppiamento: Per dati - Utilizza proprietà strettamente necessarie per rappresentare un contatto.
12. **ProfileSelectionController:**
    - Coesione: Comunicazionale - Le operazioni (onSelect, onAdd, ecc.) sono usate per controllare le interazioni con la lista profili.
    - Accoppiamento: Per controllo - Passa eventi per attivare la logica di altre classi.
13. **Checker ed estensioni:**
    - Coesione: Funzionale. Le interfacce e le implementazioni si occupano esclusivamente della validazione.
    - Accoppiamento: Per dati - Ricevono solo stringhe o numeri come input per il controllo.
14. **Filters (ConcreteFilter, FilterDecorator ed estensioni):**
    - Coesione: Funzionale. Ogni classe si occupa di un tipo specifico di filtro, rispettando il **Single Responsibility Principle**.
    - Accoppiamento: Per dati - Ricevono e restituiscono liste di contatti filtrati.

Descrizione Generale delle classi:

Abbiamo deciso di implementare le classi della nostra AddressBookApplication seguendo il pattern MVC. Per rappresentare i concetti principali dell'applicazione, abbiamo pensato diverse classi, tra cui:

- **Profile:**
  La classe Profile rappresenta l'entità di un profilo all'interno dell'applicazione. E' utilizzata per gestire la selezione e l'accesso ad una specifica rubrica. Include gli attributi già definiti in fase di analisi dei requisiti e permette l'accesso alla rubrica associata tramite un AddressBookPath, ovvero il nome del path che fa riferimento al file contenente l'effettiva rubrica.
  Infine la classe implementa le interfacce di Checker per garantire la validazione dei campi inseriti dall'utente.

- **AddressBook:**
  La classe AddressBook rappresenta il concetto principale dell'applicazione, come suggerisce il suo nome. La lista dei contatti è implementata utilizzando un TreeSet di oggetti Contact, ordinata per cognome e per nome tramite il metodo compareTo predefinito.

La classe mantiene un riferimento alla classe RecentlyDeleted associata a quella rubrica e contiene una tagMap che associa ad ogni Tag esistente (chiavi) un Set di contatti (valori) che hanno come attributo quel Tag.
In aggiunta alle operazioni di base, come l'aggiunta, la rimozione, la modifica ed il recupero dei contatti, la classe include metodi di lettura e scrittura per la persistenza su file. Questi metodi trasformano la collezione in un formato non-Observable prima di salvarla su file, poiché i tipi Observable non sono serializzabili e quindi non adatti per la memorizzazione su file.

- **Contact:**
  La classe Contact gestisce tutti gli attributi definiti durante l'analisi dei requisiti, tra cui gli attributi mail e phone implementati come array di String per permettere l'inserimento di più valori. L'attributo tags è invece rappresentato come un Set di oggetti Tag, consentendo di associare un numero "illimitato" di tag ad un contatto.
  La classe include metodi add e remove per gestire dinamicamente gli attributi non obbligatori, permettendo agli utenti di aggiungere o rimuovere dettagli a seconda delle necessità. Questa flessibilità assicura che il concetto centrale di un contatto venga mantenuto, consentendo al contempo la personalizzazione per realizzare casi d'uso specifici.

- **SafeContact:**
  La classe SafeContact estende la classe Contact, aggiungendo controlli di validazione che garantiscono la correttezza dei dati inseriti sulla base dei requisiti precedentemente descritti. Questa progettazione permette di mantenere la flessibilità per future implementazioni alternative e promuove il riutilizzo del codice esistente, aderendo così al **Open - Closed Principle**, garantendo così una progettazione flessibile e facilmente estendibile.

- **RecentlyDeleted:**
  La classe gestisce i contatti eliminati di recente utilizzando una TreeMap osservabile, in cui la chiave è una LocalDate (che rappresenta la data in cui un contatto è stato eliminato) e il valore è una ArrayList di contatti eliminati in quella data specifica. Questa struttura consente un confronto efficiente tra la data di eliminazione e la data odierna. Per mantenere l'integrità dei dati, la classe include un metodo removeOlderThan(int days), che rimuove automaticamente le voci più vecchie di un determinato numero di giorni. Alternativamente al suo utilizzo abbiamo pensato di implementare anche un metodo removePermanently(Contact contact) che permette all'utente di rimuovere manualmente e definitivamente il contatto anche dagli eliminati di recente.

- **Tag:**
  La classe Tag è stata implementata per garantire la coerenza concettuale, rappresentando ogni entità centrale con una classe dedicata. I tag rappresentano descrittori per i contatti, consentendo una migliore categorizzazione e offrendo una ricerca efficace all'interno dell'applicazione. Avere una classe indipendente per i tag aderisce al **principio di modularità**, rendendo più facile estendere e mantenere questa funzionalità.

Per ogni classe ci siamo occupati di includere metodi

- Getter e Setter per accedere correttamente agli attributi di ognuna, garantire l'**incapsulamento** e rispettare l'**Open - Closed Principle**.
- writeObject e readObject per la serializzazione e deserializzazione dei dati, in modo da convertire l'oggetto in un formato adatto alla scrittura e lettura su file.

Abbiamo immaginato due **VIEW** principali per l'applicazione :

- La prima view è dedicata alla selezione del profilo, gestita dalla classe **ProfileSelectionController**.
- La seconda view è visualizzabile dopo aver scelto o creato un profilo, permettendo all'utente di interagire con la vera e propria rubrica. Questa view viene ottenuta tramite la classe **AddressBookController**.

Il passaggio da una view all'altra, è gestito tramite una classe di utilità dedicata, chiamata **SceneManager**, che permette di caricare in modo indipendente sia la view di selezione dei profili, sia la view della rubrica associata, a seconda delle necessità.

Il nostro principale obiettivo è di mantenere una chiara separazione tra il profilo e le rubriche, consentendo alla rubrica di funzionare come un'applicazione autonoma, indipendentemente dalla selezione del profilo precedente. Una volta selezionato il profilo, il programma potrà funzionare come una rubrica totalmente indipendente dall'implementazione e dalla view precedente.

Questa scelta è stata fatta per massimizzare il **principio di separazione delle preoccupazioni**, assicurando che questi aspetti diversi del sistema siano gestiti da moduli distinti e non sovrapposti. Così facendo, vorremmo puntare a semplificare sia lo sviluppo che la manutenzione dell'applicazione, riducendo le dipendenze e consentendo una gestione e un aggiornamento più semplici di ogni parte del sistema.

Abbiamo poi introdotto diverse **INTERFACCE** all'interno della nostra applicazione con l'obiettivo di applicare il **principio di segregazione delle interfacce** e realizzare un'applicazione **modulare**, preoccupandoci di progettare interfacce piccole e altamente specifiche. Questa decisione è stata presa in risposta alla necessità di consentire a più oggetti complessi di implementare metodi concettualmente simili e di uso generale. Le interfacce pensate sono le seguenti:

- **Checker:** Interfaccia funzionale utilizzata per controllare il corretto inserimento dei campi in fase di creazione di un contatto. L'unico metodo presente nell'interfaccia è il metodo check che riceve in input una stringa e restituisce un booleano. L'interfaccia viene implementata da tre classi che verificano la validità di campi specifici:
- **PhoneChecker, StringChecker, EmailChecker** le cui realizzazioni complete sono **ItalianPhoneChecker** (in vista di un futuro supporto multilingua per l'interfaccia utente)**, CharacterLimitStringChecker, SimpleEmailChecker.**
- **Filter:** Inizialmente, avevamo implementato un pattern Decorator per il filtraggio personalizzato della ricerca, basato su tutti i campi associati a un contatto. Questo approccio ci permetteva di comporre più filtri in modo flessibile. Successivamente, abbiamo scoperto l'esistenza della classe FilteredList (nel package javafx.collections.transformation), che consente il filtraggio diretto tramite predicati booleani. Abbiamo quindi adattato il nostro design, trasformando i filtri in interfacce funzionali che agiscono come predicati. La nostra interfaccia Filter ora estende

direttamente Predicate<Contact>. In questo modo, ogni filtro può essere applicato come un predicato booleano, mantenendo la compatibilità con FilteredList. Abbiamo comunque mantenuto il pattern Decorator per consentire la combinazione di più filtri. Questo approccio ci permette di creare filtri composti che effettuano verifiche logiche, ad esempio tramite operazioni "OR". Ogni filtro riceve in input un Set di contatti e una stringa di ricerca e restituisce un valore booleano che indica se il contatto soddisfa o meno il criterio.

- **Taggable:** utilizzata per realizzare oggetti con la caratteristica di essere “taggabili”, ovvero associati ad un oggetto di classe Tag, è implementata dalla classe Contact.
- **TaggableList:** estensione dell'interfaccia Taggable, utilizzata per gestire una collezione di oggetti che supportano i Tag.
- **OnEditable:** utilizzata per definire operazioni di base implementate da un controller per poter gestire oggetti modificabili.
- **TrashCan:** utilizzata per definire le operazioni implementate da una classe che contiene un “cestino”, è infatti implementata da RecentlyDeleted.
- **ContactList:** utilizzata per definire i metodi relativi ad una lista di contatti, è infatti implementata da AddressBook.

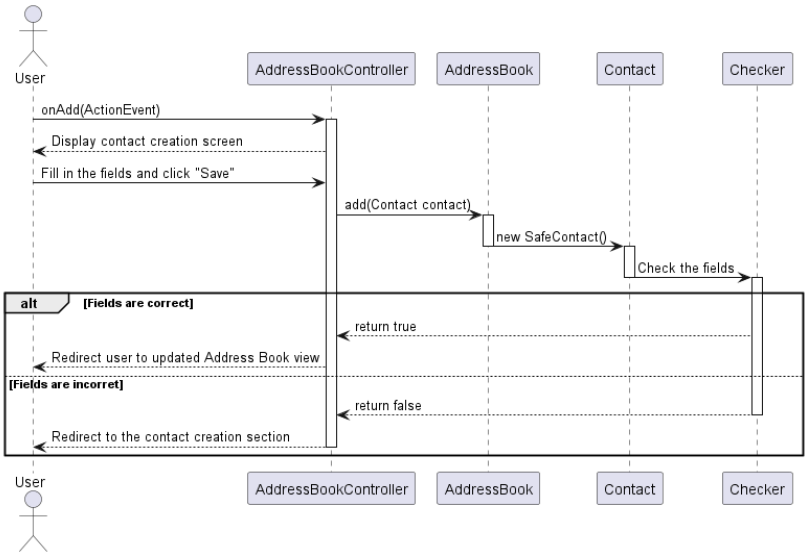
Abbiamo pensato di introdurre una classe di utilità dedicata alla **GESTIONE DEI FILE**, chiamata appunto **FileManager** che memorizza i path relativi alla lista di profili esistenti, alle rubriche associate e alle foto profilo utilizzate sia dai profili che dai contatti. Grazie ai metodi della classe è possibile ottenere queste informazioni per permettere alla classe **SceneManager** sia di ottenere le due view sopra descritte, sia di importare ed esportare su File le rubriche quando richiesto dall’utente (permettendo una versione dei due metodi in formato VCard).

Infine abbiamo pensato ad implementare una classe apposita che si occupi della **COMPATIBILITA’ CON JAVAFX**: **FXCollectionConverter** si occupa di convertire le collezioni standard di Java in strutture dati compatibili con JavaFX (ad esempio, ListProperty, SetProperty, e MapProperty). Questo è utile per integrare i dati dell'applicazione con i componenti grafici JavaFX, che utilizzano proprietà osservabili.
La classe contiene metodi pack e unpack, per i vari tipi di collezione, sono progettati per convertire strutture dati standard Java in proprietà osservabili compatibili con JavaFX (e viceversa). Vi è un ulteriore metodo propertize privato che prende una collezione in input per trasformarla in una proprietà osservabili.

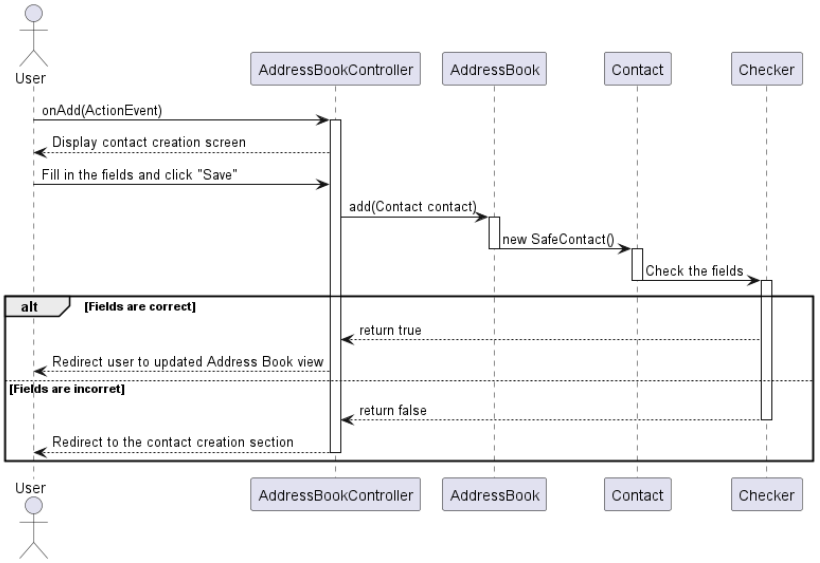
Diagrammi di Sequenza

Rappresentazione delle interazioni più significative tra gli oggetti della nostra applicazione

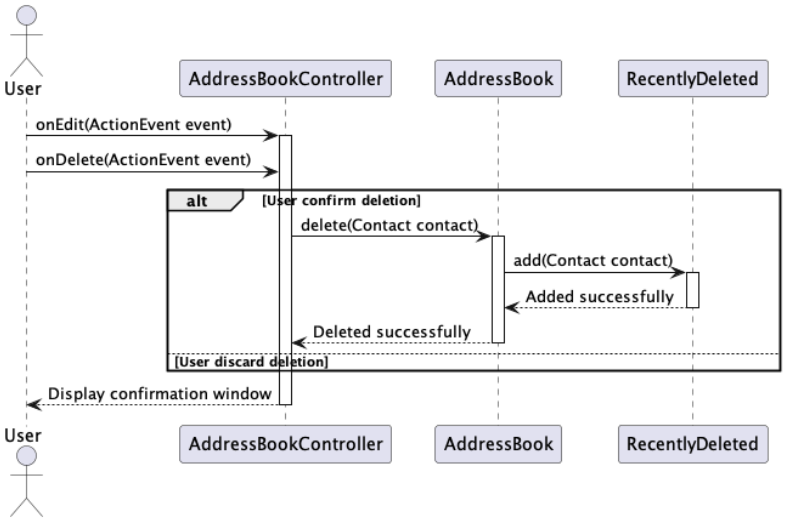
1. Create Profile



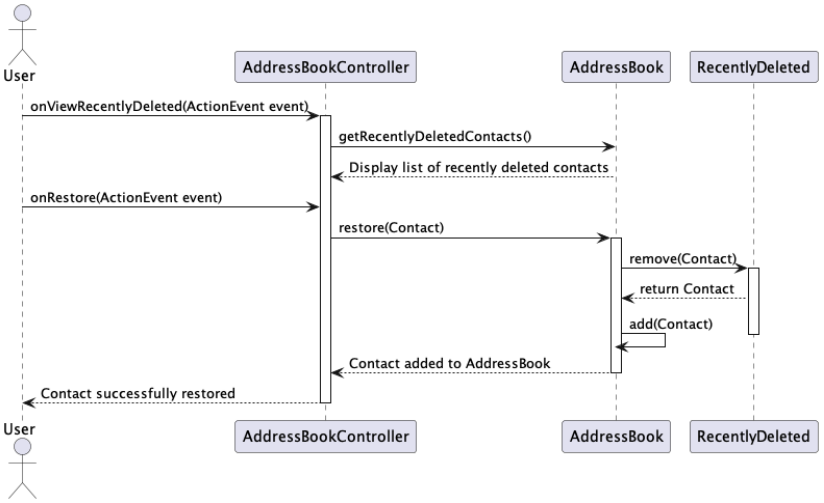
2. Create Contact



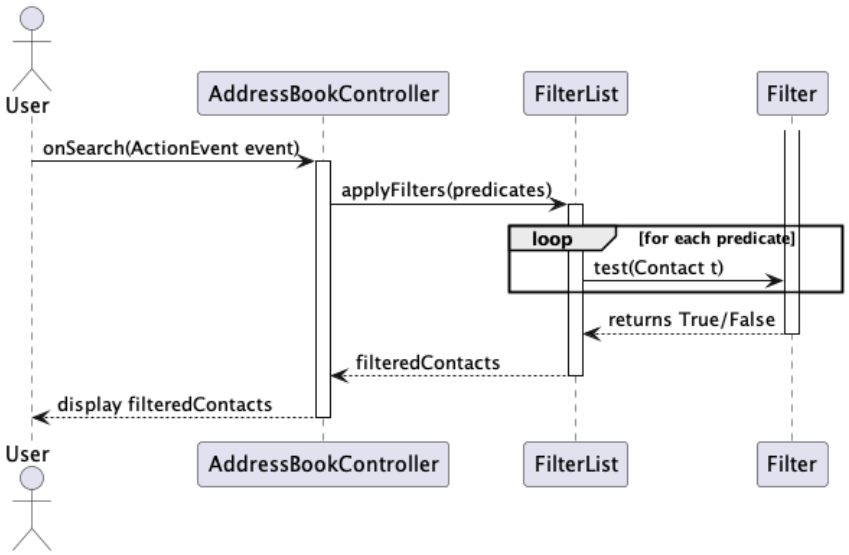
3. Delete Contact



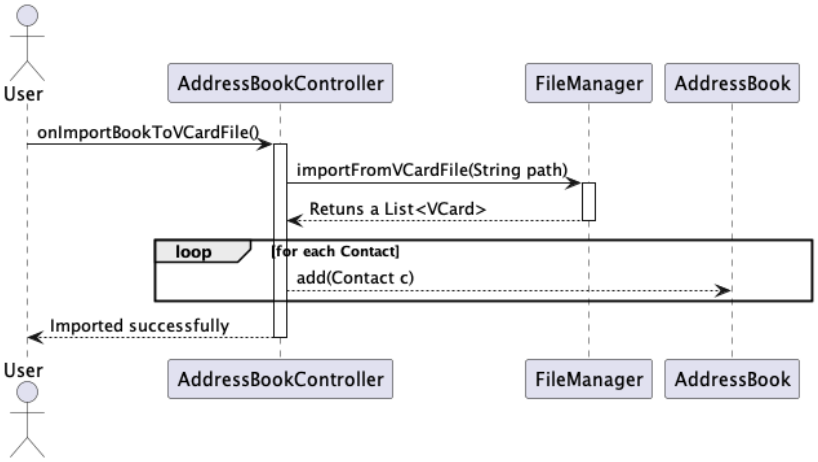
4. Restore Contact



5. Search Contact



6. Import Address Book from File



7. Export Address Book to File

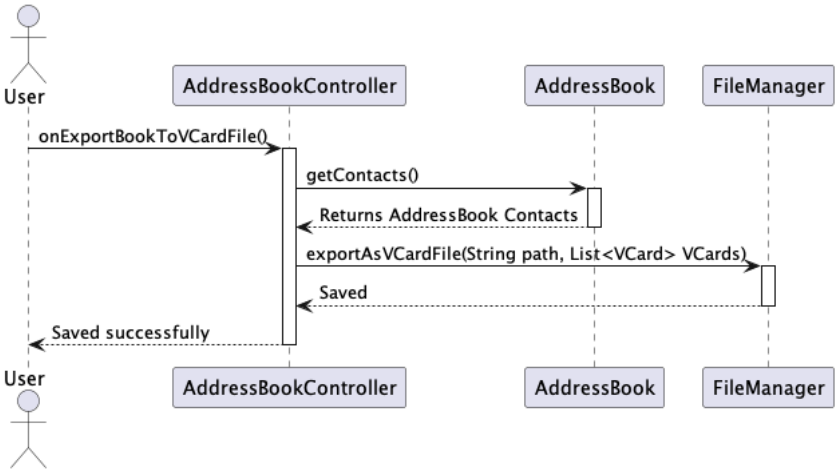


Diagramma dei package