

Design Engineering

Maxim Makhovskyy, Nicola Liguori, Luca Lanzetta, Marco Orefice

December 16, 2024

Contents

1	Introduzione	3
1.1	Descrizione generale	3
2	Design architetturale	4
3	Diagramma dei package	5
4	Diagrammi delle classi	6
4.1	Model diagram	6
4.2	Commento sui principi di buona progettazione	7
4.2.1	Interfaccia AddressBook	7
4.2.2	Classe AddressBookModel	8
4.2.3	Interfaccia ExportFileStrategy	10
4.2.4	Interfaccia ImportFileStrategy	11
4.3	Controller Diagram	12
4.3.1	Panoramica generale e sezioni del diagramma	13
4.3.2	Coesione	13
4.3.3	Accoppiamento	13
4.3.4	Pattern Chain of Responsibility	14
4.3.5	Classe Validator	15
4.3.6	Classe AddContactController	15
4.3.7	Classe ModifyContactController	18

5	Diagrammi di sequenza	21
5.1	Salvataggio contatto	21
5.2	Rimozione contatto	23
5.3	Ricerca contatto	24
5.4	Importa contatto	25
5.5	Esporta contatto	26
6	Matrice di tracciabilità dei requisiti	27

1 Introduzione

Il seguente documento illustra l'architettura e il design del progetto di Ingegneria del Software proposto.

1.1 Descrizione generale

Il documento rappresenta un punto di riferimento per lo sviluppo e la comprensione del sistema, descrivendo le componenti chiave, le interazioni e le decisioni progettuali adottate. In particolare, vengono forniti: Diagrammi delle classi, che descrivono la struttura statica del sistema, evidenziando le relazioni tra le entità e la loro organizzazione. Diagrammi di sequenza, che illustrano le interazioni dinamiche tra gli oggetti nelle operazioni più significative, evidenziando i flussi di controllo e di dati. Diagramma dei package, che aiuta a comprendere rapidamente la struttura modulare del sistema. Commenti sui diagrammi con una discussione delle scelte progettuali effettuate, valutate in termini di: coesione e accoppiamento.

La documentazione ha l'obiettivo di rendere chiaro il disegno del sistema garantendo un'implementazione che sia allineata ai requisiti funzionali e non funzionali stabiliti in precedenza.

2 Design architetturale

L'architettura di riferimento del progetto è il Model-View-Controller, un pattern di progettazione utilizzato in molte applicazioni web e desktop. La nostra scelta sull'architettura ha le seguenti motivazioni:

- JavaFX, l'applicazione si basa su JavaFX che è progettata su MVC. Continuare a seguire la stessa filosofia rende più semplice la coesione tra i vari componenti software.
- Separation of concerns, uno dei principi di buona progettazione è la separazione delle preoccupazioni, ossia aspetti diversi del sistema devono essere gestiti da moduli distinti e non sovrapposti; il Model-View-Controller è fondato su questo principio.
- Testing semplificato, l'architettura Model-View-Controller facilita il testing unitario. I modelli possono essere testati indipendentemente dai controller e dalle viste, migliorando la qualità del codice.

3 Diagramma dei package

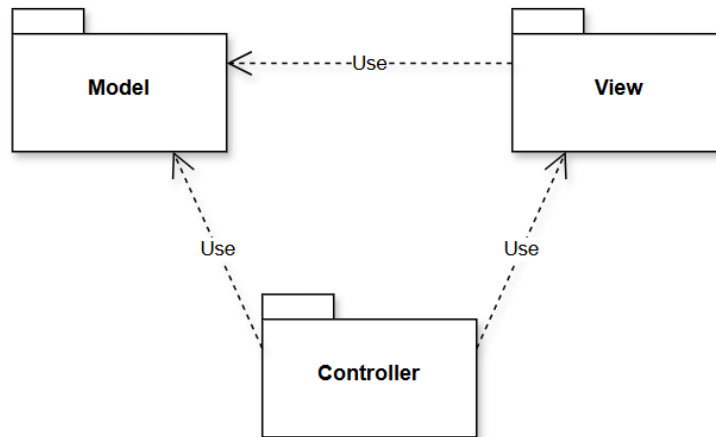


Figure 1: Package diagram

Essendo che il nostro design architetturale si basa su MVC, i package principali saranno proprio Model, View e Controller.

4 Diagrammi delle classi

4.1 Model diagram

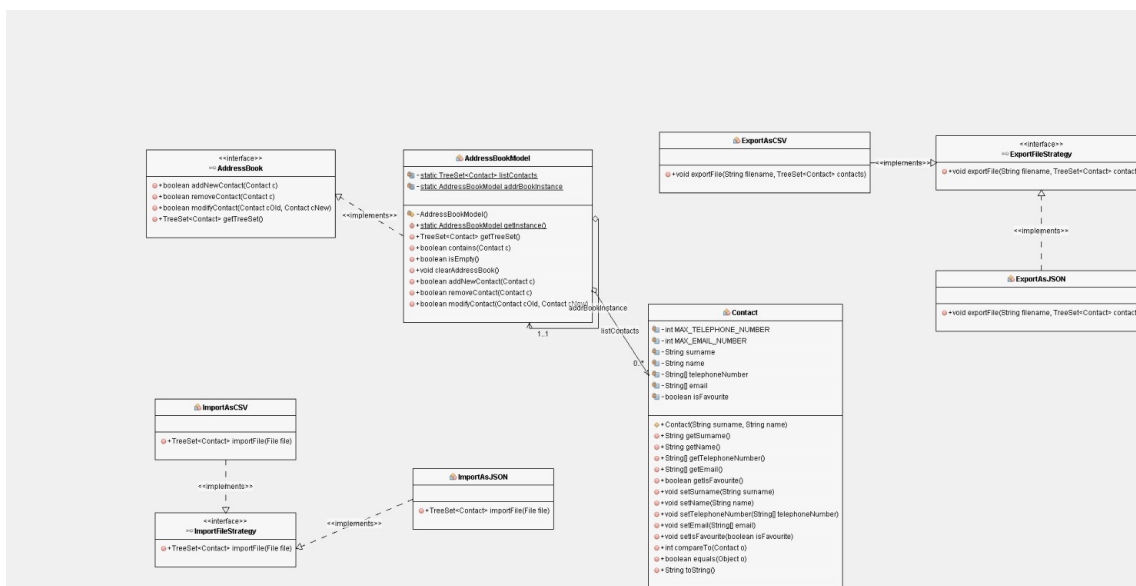


Figure 2: Model diagram

4.2 Commento sui principi di buona progettazione

Il nostro obiettivo primario per la fase di design è stato quello di seguire sin dall'inizio i principi di buona progettazione per lo sviluppo della nostra applicazione. Per semplificare il processo, abbiamo cercato di implementare alcuni design pattern comuni. In questo diagramma si può notare già il primo, ossia il design pattern "Strategy" per la selezione del tipo di file da poter importare ed esportare. Vantaggi del pattern strategy;

- Isolamento delle operazioni di importazione ed esportazione dal resto del sistema
- Open/Close principle, ossia un componente software deve essere sempre disponibile all'estensione, ma chiuso alle modifiche. Il pattern strategy utilizzato rispecchia questo principio.

Per quanto riguarda la classe Contact non c'è nulla da aggiungere, definisce e implementa i metodi necessari per la gestione dei contatti. La classe AddressBookModel è responsabile di tutte le operazioni base per la gestione dei contatti. L'AddressBookModel utilizza una TreeSet per poter effettuare le operazioni di aggiunta, rimozione e ricerca di un contatto con una complessità computazionale di tipo logaritmica. La maggior parte dei metodi da implementare riutilizza i metodi già messi a disposizione dalla collezione TreeSet. AddressBookModel utilizza il pattern Singleton, ovvero una classe che può avere una sola istanza. Nel nostro caso, questo pattern semplifica la gestione della rubrica.

4.2.1 Interfaccia AddressBook

L'interfaccia AddressBook definisce il contratto per la gestione dei contatti. Definisce

- addNewContact(): boolean
- removeContact(): boolean
- modifyContact(): boolean

4.2.2 Classe AddressBookModel

Class	AddressBookModel
Descrizione	La classe "AddressBookModel" implementa l'interfaccia "AddressBook", gestendo i contatti con una struttura dati di tipo "TreeSet". È anche singleton, in quanto permette di ottenere una sola istanza della rubrica tramite il metodo "getInstance()".
Attributi	<ul style="list-style-type: none">- listContacts: TreeSet < Contact > Collezione che memorizza i contatti nella rubrica, utilizzando una "TreeSet" che assicura l'ordinamento dei contatti.- addrBookInstance: AddressBookModel Rappresenta l'istanza singleton della classe "AddressBookModel".

Class	AddressBookModel
Metodi	<ul style="list-style-type: none"> - getInstance(): AddressBookModel Metodo statico che restituisce l'istanza della rubrica. Se l'istanza non esiste, viene creata. - getTreeSet(): TreeSet <code>< Contact ></code> Restituisce la lista dei contatti sotto forma di "TreeSet". - isEmpty(): boolean Restituisce "true" se la rubrica è vuota, "false" altrimenti. - clearAddressBook(): void Svuota la rubrica, rimuovendo tutti i contatti. - addNewContact(Contact c): boolean Aggiunge un nuovo contatto nella rubrica. Restituisce "true" se il contatto è stato aggiunto, "false" se il contatto esiste già. - removeContact(Contact c): boolean Rimuove un contatto dalla rubrica. Restituisce "true" se il contatto è stato eliminato con successo. - modifyContact(Contact cOld, Contact cNew): boolean Modifica un contatto esistente nella rubrica. Restituisce "true" se la modifica è andata a buon fine.

4.2.3 Interfaccia ExportFileStrategy

Interface	ExportFileStrategy
Descrizione	L'interfaccia ExportFileStrategy definisce il metodo di esportazione di un file con contatti.
Metodi	<ul style="list-style-type: none">- exportFile(filename: String, contacts: TreeSet<Contact>): void Questo metodo esporta i contatti in un file specificato dal percorso filename.
Implementazioni	<ul style="list-style-type: none">- ExportCSVStrategy Esporta i contatti in un file CSV.- ExportJSONStrategy Esporta i contatti in un file JSON.

4.2.4 Interfaccia ImportFileStrategy

Interface	ImportFileStrategy
Descrizione	L'interfaccia ImportFileStrategy definisce il metodo di importazione di un file contenente contatti.
Metodi	<ul style="list-style-type: none">- importFile(filename: String): TreeSet<Contact> Questo metodo importa i contatti da un file specificato dal percorso filename e restituisce un TreeSet contenente i contatti importati.
Implementazioni	<ul style="list-style-type: none">- ImportCSVStrategy Importa i contatti da un file CSV.- ImportJSONStrategy Importa i contatti da un file JSON.

4.3 Controller Diagram

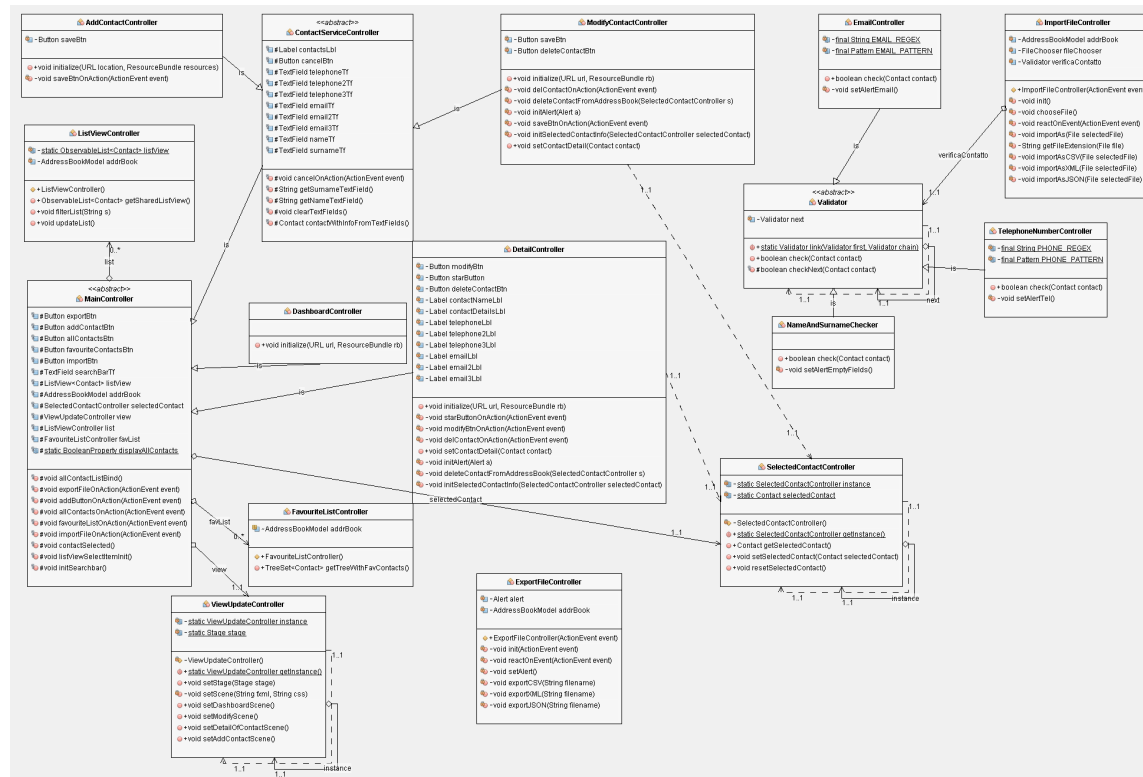


Figure 3: Controller diagram

4.3.1 Panoramica generale e sezioni del diagramma

Il diagramma rappresenta l'architettura che gestisce le funzionalità legate ai contatti (come aggiungere, modificare, eliminare e visualizzare).

4.3.2 Coesione

- Alta Coesione nelle Classi di Controllo

Le classi AddController, DashboardController, DetailController e ModifyController mostrano alta coesione poiché ognuna è responsabile di un compito specifico legato alla gestione dei contatti. Ad esempio, AddController si occupa esclusivamente dell'aggiunta dei contatti, DetailController gestisce la visualizzazione dei dettagli e ModifyController gestisce la modifica dei contatti.

- Alta Coesione nei Validatori

Le classi EmailController, TelephoneNumberController e NameAndSurnameChecker hanno alta coesione poiché ciascuna esegue un tipo specifico di validazione.

4.3.3 Accoppiamento

- Ridotto Accoppiamento tramite Chain of Responsibility

L'uso del Pattern Chain of Responsibility nelle classi Validator, EmailController, TelephoneNumberController e NameAndSurnameChecker riduce l'accoppiamento tra i componenti del sistema. Questo pattern permette di collegare una serie di oggetti validatori in una catena, dove ciascuno può gestire una richiesta o passarla al successivo nella catena. Le classi di controllo (AddandModifyController, DashboardController, DetailController) non devono conoscere i dettagli di implementazione dei validatori, riducendo così l'accoppiamento.

4.3.4 Pattern Chain of Responsibility

Il Pattern Chain of Responsibility è stato scelto per favorire la coesione e ridurre l'accoppiamento.

- Implementazione del Pattern

La classe astratta Validator definisce un'interfaccia per la validazione dei contatti. Le classi EmailController, TelephoneNumberController e Name-AndSurnameChecker estendono Validator e implementano il metodo check per eseguire specifiche validazioni sui contatti.

- Funzionamento del Pattern

La classe Validator contiene un riferimento a un altro Validator, permettendo di creare una catena di validatori. Quando viene richiesta una validazione, ogni Validator nella catena può processare la richiesta o passarla al successivo. Questo permette di aggiungere nuovi tipi di validazione senza modificare le classi di controllo, migliorando la coesione e riducendo l'accoppiamento.

4.3.5 Classe Validator

Class	Validator
Descrizione	La classe astratta Validator implementa il pattern Chain of Responsibility per la validazione dei contatti da inserire in una rubrica.
Metodi	<ul style="list-style-type: none"> - link(first: Validator, chain: Validator...): Validator Questo metodo crea la catena di validatori, restituendo il primo validatore della sequenza. - check(contact: Contact): boolean Questo metodo è astratto e deve essere implementato dalle classi derivate per validare un contatto. - checkNext(contact: Contact): boolean Questo metodo permette di invocare il prossimo validatore della catena per continuare la validazione del contatto.
Attributi	<ul style="list-style-type: none"> - next : Validator Riferimento al prossimo validatore nella catena.

4.3.6 Classe AddContactController

Class	AddContactController
Descrizione	<p>La classe "AddContactController" estende "ContactServiceController" e implementa "Initializable". Si occupa della gestione dell'interfaccia utente per l'aggiunta di un nuovo contatto nella rubrica. Utilizza un validatore per verificare i dati immessi prima di salvarli nella rubrica.</p>
Attributi	<ul style="list-style-type: none"> - saveBtn: Button Bottone per il salvataggio di un nuovo contatto. - addrBook: AddressBookModel Rappresenta l'istanza della rubrica dove vengono salvati i contatti. - view: ViewUpdateController Rappresenta il controller per l'aggiornamento della vista. - selectedContact: SelectedContactController Rappresenta il controller per la gestione del contatto selezionato. - list: ListViewController Gestisce l'elenco dei contatti visualizzati. - listView: ListView Visualizza la lista dei contatti.

Class	AddContactController
Metodi	<ul style="list-style-type: none">- initialize(URL location, ResourceBundle resources): void Metodo che inizializza i vari controller e lega la lista dei contatti alla vista.- saveBtnOnAction(ActionEvent event): void Metodo che raccoglie i dati dalle textfield, li verifica tramite il "Validator" e, se validi, aggiunge il contatto alla rubrica.

4.3.7 Classe ModifyContactController

Class	ModifyContactController
Descrizione	La classe "ModifyContactController" estende "ContactServiceController" e implementa "Initializable". Gestisce l'interfaccia utente per la modifica e l'eliminazione di un contatto esistente nella rubrica. Permette di confermare l'eliminazione del contatto e di salvare le modifiche dopo aver validato i dati.

Class	ModifyContactController
Attributi	<ul style="list-style-type: none"> - saveBtn: Button Bottone per salvare le modifiche apportate a un contatto esistente. - deleteContactBtn: Button Bottone per eliminare il contatto selezionato. - addrBook: AddressBookModel Rappresenta l'istanza della rubrica dove vengono memorizzati i contatti. - view: ViewUpdateController Rappresenta il controller per l'aggiornamento della vista. - selectedContact: SelectedContactController Rappresenta il controller per la gestione del contatto selezionato. - list: ListViewController Gestisce la lista dei contatti visualizzati. - listView: ListView Visualizza la lista dei contatti. - surnameTf, nameTf, telephoneTf, telephone2Tf, telephone3Tf, emailTf, email2Tf, email3Tf: TextField Campi di testo per la modifica delle informazioni del contatto selezionato.

Class	ModifyContactController
Metodi	<ul style="list-style-type: none"> - initialize(URL url, ResourceBundle rb): void Inizializza i vari controller e i dati relativi al contatto selezionato per la modifica. - delContactOnAction(ActionEvent event): void Elimina il contatto selezionato previa conferma dell'utente tramite un alert. - deleteContactFromAddressBook: void Rimuove il contatto selezionato dalla rubrica e aggiorna la lista. - initAlert(Alert a): void Inizializza l'alert per la conferma dell'eliminazione di un contatto. - saveBtnOnAction(ActionEvent event): void Salva le modifiche al contatto selezionato, se i dati sono validi. - initSelectedContactInfo: void Inizializza le informazioni del contatto selezionato sulla pagina di modifica. - setContactDetail(Contact contact): void Imposta i dettagli del contatto selezionato nei campi di testo per la modifica.

5 Diagrammi di sequenza

5.1 Salvataggio contatto

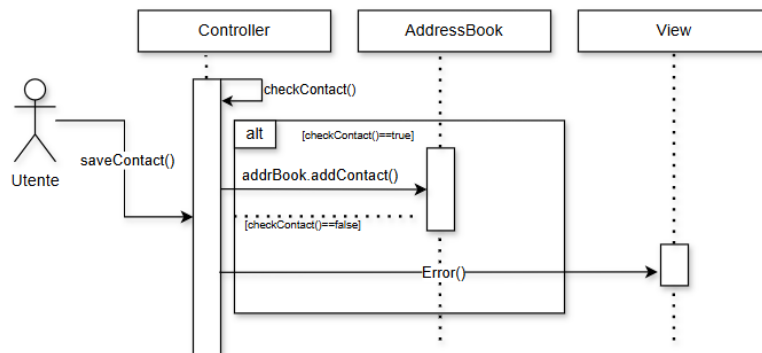


Figure 4: Sequence diagram "saveContact()"

Il diagramma di sequenza mostra l'interazione tra Utente, Controller, Address-Book View durante il processo di salvataggio di un contatto.

- Utente

L'utente inizia l'interazione inviando il messaggio `saveContact()` al Controller.

- Controller

Il Controller riceve il messaggio `saveContact()` e invia il messaggio `checkContact()` al Validator per verificare se il contatto rispetta determinati requisiti.

- Controller

Il Controller utilizza un blocco alternativo (`alt`) per gestire due possibili scenari:

1. Se `checkContact()` restituisce `false`, il Controller invia il messaggio `Error()` alla View per notificare un errore.

2. Se `checkContact()` restituisce `true`, il Controller invia il messaggio `AddressBook.addContact()` all'`AddressBook` per aggiungere il nuovo contatto.

- `AddressBook`

L'`AddressBook` aggiunge il contatto se il controllo è positivo.

- `View`

La `View` riceve il messaggio `Error()` e gestisce l'errore visualizzando un messaggio appropriato all'utente.

5.2 Rimozione contatto

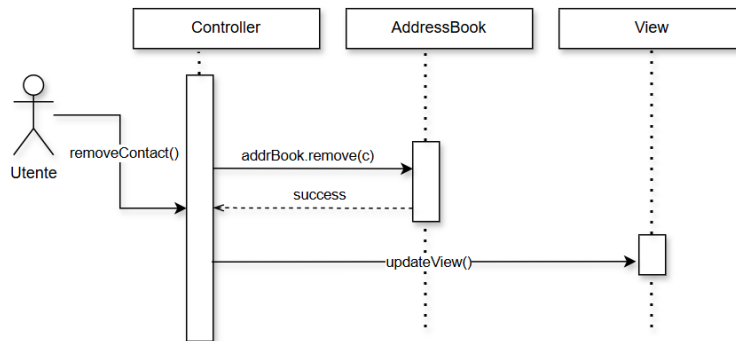


Figure 5: Sequence diagram "removeContact("

Il seguente diagramma di sequenza rappresenta l'interazione tra Utente, Controller, AddressBook View durante il processo di rimozione di un contatto.

- Utente

L'utente invia una richiesta `removeContact()` al Controller.

- Controller

Il Controller riceve la richiesta e invia `addrBook.remove(c)` all'AddressBook per rimuovere il contatto.

- AddressBook

L'AddressBook esegue la rimozione e risponde con un messaggio di successo (`success`) al Controller.

- Controller

Il Controller invia una richiesta `updateView()` alla View per aggiornare la visualizzazione.

5.3 Ricerca contatto

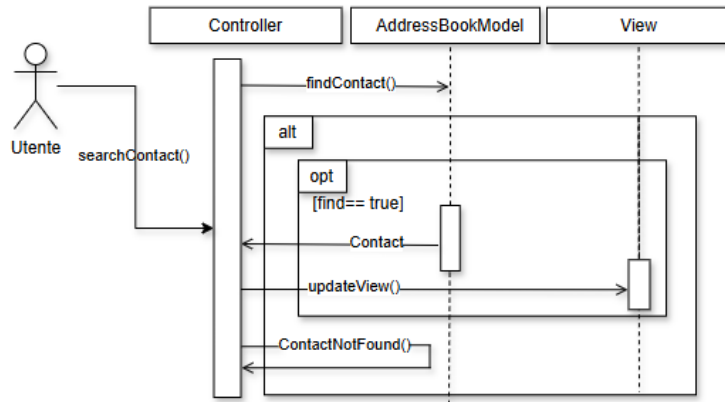


Figure 6: Sequence diagram "searchContact()"

Il diagramma di sequenza mostra l'interazione tra `Utente`, `AddressBookController`, `AddressBookModel` e `AddressBookView` durante la ricerca di un contatto.

- `Utente`

L'utente inizia l'interazione inviando la richiesta `searchContact()` all'`AddressBookController`.

- `Controller`

Il `Controller` inoltra una richiesta di ricerca tramite `findContact()` all'`AddressBookModel`.

- `AddressBookModel`

Nel diagramma è presente un blocco alternativo (`alt`) con due opzioni:

1. Se il contatto viene trovato (`[find==true]`) l'`AddressBookModel` restituisce il `Contact` all'`AddressBookController`.
2. Se il contatto non viene trovato l'`AddressBookModel` invia `ContactNotFound()` all'`AddressBookController`.

- `AddressBookView`

L'`AddressBookView` viene aggiornata attraverso il metodo `updateView()` per visualizzare le informazioni del contatto all'`Utente`.

5.4 Importa contatto

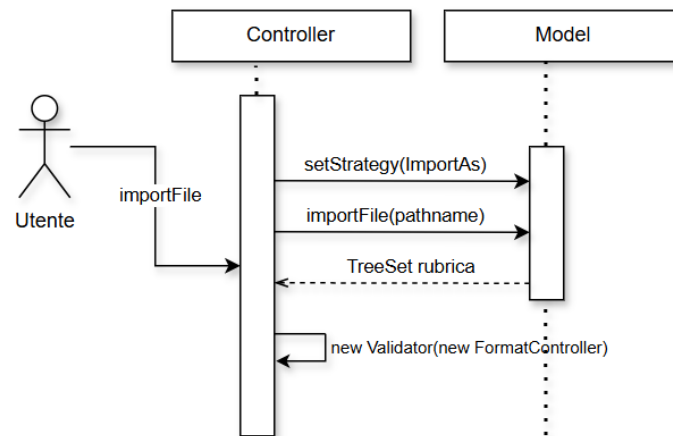


Figure 7: Sequence diagram "importFile()"

Il diagramma di sequenza mostra l'interazione tra Utente, Controller e Model durante il processo di importazione di un file.

- Utente

L'Utente inizia l'interazione inviando un messaggio importFile al Controller.

- Controller

Il Controller riceve il messaggio di importFile e invia un messaggio setStrategy(ImportAs) al Model per impostare la strategia di importazione. Dopodiché il Controller invia un messaggio importFile(pathname) al Model in modo da importare il file specificato dal percorso.

- Model

Il Model risponde con una TreeSet rubrica al Controller.

- Controller

Il Controller crea un oggetto Validator passando come parametro un nuovo oggetto FormatController. Validator controlla ogni contatto della TreeSet. Per ogni contatto che non supera la fase di controllo viene rimosso dalla TreeSet. La TreeSet viene settata nel Model.

5.5 Esporta contatto

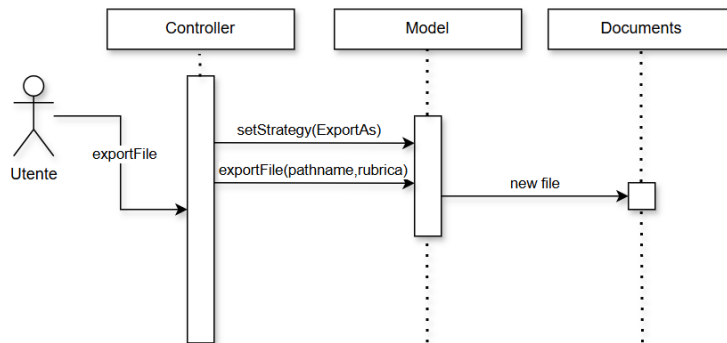


Figure 8: Sequence diagram "exportFile()"

Il diagramma di sequenza mostra l'interazione tra Utente, Controller e Model durante il processo di esportazione di un file.

- Utente

L'Utente inizia l'interazione inviando un messaggio exortFile al Controller.

- Controller

Il Controller riceve il messaggio di exportFile e invia un messaggio set-Strategy(ExportAs) al Model per impostare la strategia di esportazione. Dopodiché il Controller invia un messaggio exportFile(pathname, rubrica) al Model in modo da importare il file specificato dal percorso con i dati della rubrica.

- Model

Il Model dopo aver elaborato i messaggi del Controller crea un nuovo file con i dati della rubrica.

- Documents

Una volta creato il file viene inviato a Documents per la memorizzazione.

6 Matrice di tracciabilità dei requisiti

ID	Design	Implementazione	Testing	Stato
IF-1.1	X	X	X	Design, Implementazione e Testing terminati
IF-1.2	X	X	X	Design, Implementazione e Testing terminati
IF-1.3	X	X	X	Design, Implementazione e Testing terminati
IF-1.4	X	X	X	Design, Implementazione e Testing terminati
IF-1.5	X	X	X	Design, Implementazione e Testing terminati

ID	Design	Implementazione	Testing	Stato
IF-1.6	X	X	X	Design, Implementazione e Testing terminati
IF-1.7	X	X	X	Design, Implementazione e Testing terminati
IF-1.8	X	X	X	Design, Implementazione e Testing terminati
IF-1.9	X	X	X	Design, Implementazione e Testing terminati
IF-1.10	X	X	X	Design, Implementazione e Testing terminati
UI-1.1	X	X	X	Design, Implementazione e Testing terminati
UI-1.2	X	X	X	IDesign, Implementazione e Testing terminati

ID	Design	Implementazione	Testing	Stato
UI-1.3	X	X	X	Design, Implementazione e Testing terminati
UI-1.4	X	X	X	Design, Implementazione e Testing terminati
UI-1.5	X	X	X	Design, Implementazione e Testing terminati
UI-1.6	X	X	X	Design, Implementazione e Testing terminati
UI-1.7	X	X	X	Design, Implementazione e Testing terminati
DF-1.1	X	X	X	Design, Implementazione e Testing terminati
DF-1.2	X	X	X	Design, Implementazione e Testing terminati

ID	Design	Implementazione	Testing	Stato
DF-1.3	X	X	X	Design, Implementazione e Testing terminati
DF-1.4	X	X	X	Design, Implementazione e Testing terminati
IF-2.1	X	X	X	Design, Implementazione e Testing terminati
IF-2.2	X	X	X	Design, Implementazione e Testing terminati
UI-2.1	X	X	X	Design, Implementazione e Testing terminati
UI-2.2	X	X	X	Design, Implementazione e Testing terminati