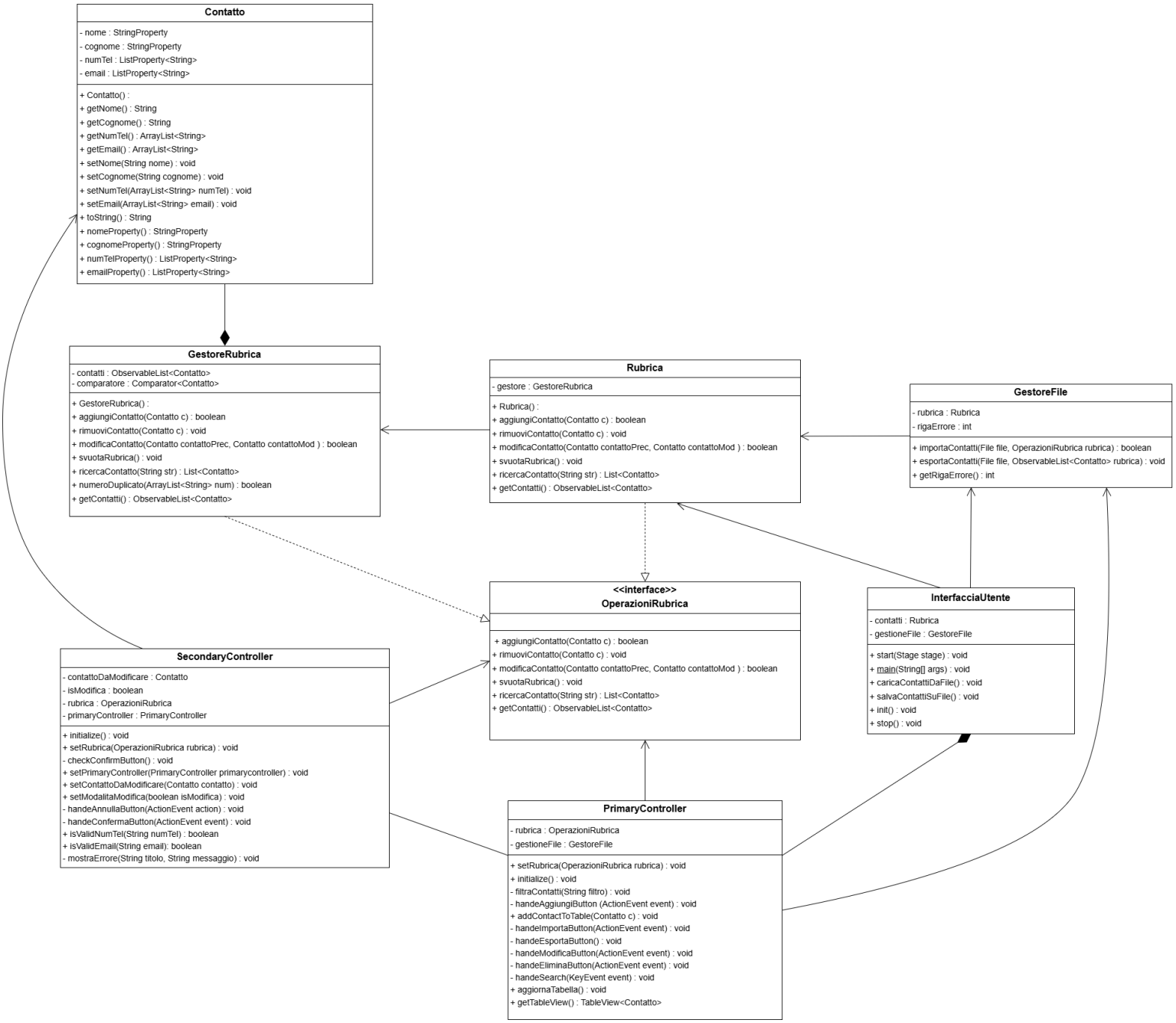
Documentazione del design

Introduzione

Il seguente documento ha lo scopo di spiegare quali sono state le scelte progettuali adottate dal gruppo nel descrivere la progettazione delle classi per il sistema di gestione della rubrica.

Diagramma delle classi



Il diagramma delle classi in figura illustra staticamente la struttura logica del sistema per la gestione di una rubrica.
Notiamo la presenza di 7 classi (Contatto, GestoreRubrica, GestoreFile, Rubrica, InterfacciaUtente, PrimaryController, SecondaryController) e un'interfaccia (OperazioniUtente).

Descrizione delle classi

Classe 'Contatto'

Questa classe rappresenta un singolo contatto nella rubrica, includendo tutti gli attributi ad esso associati, quali:

- **nome**;
- **cognome**;
- **numeri di telefono**;
- **e-mail**.

La classe implementa anche dei **metodi**:

- **getter** e **setter** per tutti gli attributi di un contatto. In questo modo è più semplice creare un nuovo contatto, assegnargli valori specifici a ciascun campo o recuperare i dati di un contatto già esistente. Questi metodi, insieme all'utilizzo degli attributi privati, permettono l'implementazione dell'incapsulamento;
- **toString()**, per visualizzare le informazioni relative al contatto.
- **nomeProperty(), cognomeProperty(), numTelProperty(), emailProperty(),** servono a fornire un accesso di tipo StringProperty e ListProperty, che sono delle proprietà di JavaFX; questi metodi forniscono un accesso diretto alle proprietà osservabili associate alla classe Contatto.

Interfaccia 'OperazioniUtente'

L'interfaccia OperazioniUtente descrive i metodi utilizzati dalla rubrica:

- **aggiungiContatto()**;
- **rimuoviContatto()**;
- **modificaContatto()**;
- **ricercaContatto()**;
- **svuotaRubrica()**;
- **getContatti()**.

Classe 'GestoreRubrica'

La classe GestoreRubrica, che implementa l'interfaccia OperazioniRubrica, contiene una collezione di oggetti di tipo Contatto, organizzata in una struttura dati ObservableList<Contatto>. La classe ha i seguenti attributi:

- **ObservableList<Contatto> contatti** : è una lista osservabile che contiene tutti i contatti presenti nella rubrica;
- **Comparator<Contatto> comparatore** : è un comparatore utilizzato per ordinare i contatti in base al cognome e, in caso di parità, al nome.

La classe implementa i metodi:

- **aggiungiContatto()**, consente di aggiungere un contatto alla rubrica intesa come collezione;
- **rimuoviContatto()**, consente di rimuovere un contatto dalla rubrica come collezione;
- **modificaContatto()**, che permette di modificare gli attributi del contatto;
- **ricercaContatto()**, che effettua una ricerca sulla collezione di Contatti, in base alla sottostringa inserita nella barra di ricerca;
- **numeroDuplicato()**, che verifica se il numero inserito è già presente in un altro contatto o meno;
- **svuotaRubrica(),** svuota completamente la rubrica andando a rimuovere tutti i contatti;
- **getContatti(),** consente di ottenere la lista di contatti della rubrica.

Classe ‘Rubrica’

La classe Rubrica, che implementa l'interfaccia OperazioniRubrica, la quale sfrutta l’incapsulamento chiamando i metodi della classe GestoreRubrica. La classe ha come attributo un oggetto di tipo **GestoreRubrica**.

La classe implementa i metodi:

- **aggiungiContatto()**;
- **rimuoviContatto()**;
- **modificaContatto()**;
- **ricercaContatto()**;
- **svuotaRubrica()**;
- **getContatti()**.

Classe ‘GestoreFile’

La classe in esame gestisce le funzionalità di import ed export della rubrica. Questa classe è progettata per lavorare con la classe Rubrica. La classe ha i seguenti attributi:

- **Rubrica rubrica** : rappresenta la rubrica su cui vengono eseguite le operazioni di importazione ed esportazione;
- **int rigaErrore** : tiene traccia del numero di riga che ha causato un errore durante l’importazione di un file.

La classe implementa tre metodi:

- **importaContatti()**, per importare, da un file compatibile verso la rubrica, un insieme di contatti;

- **getRigaErrore()**, restituisce il numero di riga che ha causato l'errore durante l'importazione;
- **esportaContatti()**, per esportare su un file i contatti presenti nella rubrica.

Classe 'InterfacciaUtente'

La quinta classe è quella che provvede alla realizzazione dell'interfaccia utente. Questa classe si occupa di mostrare all'utente che interagisce con il sistema un'interfaccia realizzata in JavaFX, che permette di rendere l'utilizzo più semplice. La classe ha i seguenti attributi:

- **Rubrica contatti**: una rubrica che contiene i contatti;
- **GestoreFile gestioneFile** : Utilizzato per caricare i contatti all'avvio dell'applicazione e per salvarli quando l'applicazione viene chiusa.

La classe in questione possiede metodi:

- **init()**, per inizializzare la rubrica con l'ultimo salvataggio;
- **start()**, per effettuare le operazioni tramite l'interfaccia grafica;
- **stop()**, per salvare lo stato della rubrica prima di chiuderla;
- **main()**, per rendere eseguibile il progetto;
- **caricaContattiDaFile(),** per caricare i contatti salvati in precedenza da un file backup nella rubrica;
- **salvaContattiSuFile()**, per salvare i contatti attualmente presenti nella rubrica in un file di backup.

Classe 'PrimaryController'

La classe PrimaryController contiene il controller principale per l'interfaccia grafica della rubrica; è responsabile delle interazioni con l'utente come l'aggiunta, la modifica, la ricerca e la visualizzazione dei contatti. La classe in questione ha i seguenti attributi:

- **OperazioniRubrica** rubrica;
- **GestoreFile** gestioneFile;

Contiene, inoltre, dei campi annotati con FXML, che rappresentano e gestiscono vari elementi dell'interfaccia utente:

- **TableView**, tabella in cui viene visualizzata la rubrica;
- **TextField**, ovvero la barra di ricerca;
- **Button**, per i bottoni di aggiunta, modifica, rimozione contatto, oltre a quelli di importa ed esporta rubrica.

Questa classe contiene i metodi:

- **initialize()**, in cui inizializza gli attributi e crea la GUI, aggiunge un listener al campo di ricerca per filtrare i contatti visualizzati;
- **filtraContatti()**, filtra i contatti in base al testo di ricerca inserito;
- **handleAggiungiButton()**, gestisce l'evento di clic sul pulsante "Aggiungi";
- **addContactToTable()**, aggiunge un nuovo contatto alla rubrica;
- **aggiornaTabella()**, aggiorna la TableView;
- **handleModificaButton()**, gestisce l'evento di clic sul pulsante "Modifica";
- **setRubrica()**, imposta la rubrica corrente;
- **handleEliminaButton()**, gestisce l'evento di clic sul pulsante "Elimina";
- **handleImportaButton()**, gestisce l'evento di clic sul pulsante "Importa";
- **handleEsportaButton()**, gestisce l'evento di clic sul pulsante "Esporta";

- **handleSearchButton()**, gestisce l’evento di pressione di un tasto nel campo di ricerca;
- **getTableView()**, restituiscela TableView corrente.

Classe ‘SecondaryController’

La classe SecondaryController gestisce la logica per la finestra di dialogo secondaria dell'applicazione, che viene utilizzata per aggiungere o modificare contatti. La classe in questione ha i seguenti attributi:

- **OperazioniRubrica** rubrica;
- **PrimaryController** primaryController;
- **Contatto** contattoDaModificare;
- **boolean** isModifica;

Contiene, inoltre, dei campi annotati con FXML, che rappresentano e gestiscono vari elementi dell’interfaccia utente:

- **TextField**, per i campi nome, cognome, 3 numeri di telefono e 3 e-mail;
- **Button**, per i bottoni di aggiunta, modifica, rimozione contatto, oltre a quelli di importa ed esporta rubrica.

Questa classe contiene i metodi:

- **initialize()**, disabilita il pulsante conferma finché non sono inseriti nome o cognome. Aggiunge un listener per abilitare il pulsante conferma quando i campi sono completati;
- **setRubrica()**, imposta la rubrica;
- **setPrimaryController()**, imposta il controller principale;
- **setContattoDaModificare()**, imposta il contatto da modificare e popola i campi di testo con i dati del contatto;
- **setModalitaModifica()**, imposta la modalità di modifica o di aggiunta;
- **handleAnnullaButton()**, chiude la schermata secondaria senza salvare le modifiche;
- **handleConfermaButton()**, ottiene i dati dai campi di testo, valida i numeri di telefono e le email, crea o modifica un contatto e aggiorna la TableView nel controller principale.
- **checkConfirmButton()**, abilita o disabilita il pulsante conferma in base alla presenza di almeno uno tra nome o cognome;
- **isValidNumTel()**, verifica se il numero di telefono inserito è valido;
- **isValidEmail()**, verifica se l'email inserita è valida;
- **mostraErrore()**, mostra un messaggio di errore all'utente.

Le relazioni tra classi

La classe GestoreRubrica ha una relazione di **composizione** con la classe Contatto. In una rubrica ci possono essere da 0 a * contatti, mentre i contatti possono essere in 1 rubrica. La Rubrica è in relazione di **associazione unidirezionale** con la classe GestoreRubrica perchè sfrutta i metodi del gestore per implementare i propri.

Entrambe le classi GestoreRubrica e Rubrica **implementano** l'interfaccia OperazioniUtente.

La classe GestoreFile ha una relazione di **associazione unidirezionale** con la classe Rubrica, necessaria per accedere ai dati durante le operazioni di importazione ed esportazione. La gestione file può avvenire su 1 rubrica, mentre la rubrica può dover gestire 0 o 1 file alla volta.

La classe InterfacciaUtente ha una relazione di **associazione unidirezionale** con le classi Rubrica e GestoreFile, in quanto mantiene un riferimento a un oggetto di entrambe le classi, mentre le due classi prima citate non mantengono un riferimento all'InterfacciaUtente**.**

La classe PrimaryController è in una relazione di **composizione** con l'InterfacciaUtente, in quanto è quest'ultima è responsabile della creazione e della distruzione del PrimaryController. Inoltre, PrimaryController e SecondaryController sono in relazione di **associazione bidirezionale**: il PrimaryController è dall'interfaccia, ma a sua volta il SecondaryController deve essere utilizzato dal primary. Le due classi devono interagire tra loro, condividendo i dati che vengono modificati, al fine di aggiornare la rubrica correttamente.

Il PrimaryController mantiene un riferimento alla classe GestoreFile, il SecondaryController alla classe Contatto. Per questo motivo, fra le due coppie vi è una relazione di **associazione unidirezionale**, in quanto Contatto e GestoreFile non sanno dell'esistenza dei controller.

Infine, entrambi i controller mantengono un riferimento all'interfaccia OperazioniRubrica, dunque risiede fra le classi e l'interfaccia una relazione di **associazione unidirezionale**.

Scelte progettuali

Nella sezione dedicata alle scelte progettuali, si analizzano le motivazioni alla base della suddivisione delle classi e dei relativi metodi, mettendo in evidenza tali decisioni in relazione ai principi di accoppiamento e coesione.

Coesione delle classi

Classe ‘Contatto’

La classe Contatto è stata progettata per gestire solamente gli attributi relativi al contatto come entità a sé stante.
Questa scelta è stata dettata dalla necessità di rendere ogni classe il più indipendente possibile: in virtù di ciò, il risultato prodotto è una classe con un livello di coesione molto alto. Si può dire che questa classe possegga una **coesione funzionale**, in quanto le operazioni all'interno del modulo lavorano insieme per gestire un contatto, compiendo operazioni sullo stesso oggetto.

Classi ‘Rubrica’ e ‘GestoreRubrica’

Le classi Rubrica e GestoreRubrica, invece, sono state pensate in modo tale che svolgono unicamente le operazioni che riguardano le interazioni con la collezione di contatti, come l’aggiunta, la rimozione e anche la modifica di un contatto. Per queste operazioni, infatti, non sarebbe una buona idea pensare di agire direttamente sull’entità “contatto”, piuttosto, risulta conveniente agire sulla collezione che manipola gli stessi.
Da questa scelta ne consegue un livello di **coesione comunicazionale**: tutte le operazioni, anche essendo diverse tra loro, agiscono sulla stessa struttura dati (ossia la collezione che accoglie i contatti).

Classe ‘GestoreFile’

La classe GestoreFile è stata progettata, come si può facilmente intuire dal nome, per gestire le operazioni di import/export della Rubrica.
Questo sta a significare che anche questo modulo possiede un livello alto di coesione, dal momento che svolge solo degli specifici task che hanno un alto livello di coerenza tra di loro. Anche per questa classe possiamo parlare di **coesione comunicazionale:** entrambi i metodi lavorano sulla stessa struttura (ossia la rubrica e il file), ma svolgendo compiti differenti.

Classe ‘InterfacciaUtente’

Tutte le sue operazioni della classe InterfacciaUtente sono strettamente legate alla gestione dell'interfaccia grafica e all'interazione con la rubrica. Per questo motivo possiamo affermare che nella classe InterfacciaUtente è presente una **coesione** di tipo **funzionale.**

Classe ‘PrimaryController’

La classe PrimaryController gestisce la TableView, i bottoni e la barra di ricerca, offrendo all’utente un’interazione con l’interfaccia grafica grazie ai suoi metodi. Questi ultimi lavorano

sugli stessi dati, ovvero i contatti della rubrica, e su una stessa struttura dati, ovvero la TableView. Per questo motivo, definiamo la **coesione** di questa classe **comunicazionale**.

Classe ‘SecondaryController’

Anche in questa classe, così come per la classe PrimaryController, consideriamo il fatto che i metodi lavorano sull’interazione utente-GUI. I dati manipolati sono ancora una volta i contatti della rubrica, tramite la schermata di aggiunta o modifica, e la struttura dati da aggiornare è sempre la TableView in base alle modifiche apportate. Di conseguenza, anche per questa classe definiamo una **coesione comunicazionale**.

Accoppiamento tra classi

Accoppiamenti della classe “Rubrica”

Dall’analisi del diagramma delle classi, emerge chiaramente che la classe **Rubrica** svolge un ruolo centrale nelle relazioni con le altre classi dell’applicazione.

Una delle relazioni più evidenti è quella tra **Rubrica** e **Contatto**. Si può parlare, in questo caso, di un **accoppiamento per dati,** in quanto la classe **Rubrica** gestisce una collezione di oggetti di tipo **Contatto**. Tuttavia, a differenza di un semplice scambio di metodi getter e setter tra le due classi, la Rubrica ottiene i valori dei contatti attraverso l'istanziazione di oggetti di tipo **GestoreRubrica**.La relazione tra **Rubrica** e **GestoreRubrica** è più complessa rispetto a quella con Contatto. Si può parlare di **accoppiamento per timbro**, poiché la classe **Rubrica** possiede un attributo di tipo **GestoreRubrica**, una struttura dati complessa che gestisce l’interazione con la collezione di contatti.

Inoltre, la classe **Rubrica** deve anche interagire con la classe **GestoreFile,** con la quale ha un **accoppiamento per timbro**. Quando la **Rubrica** esegue operazioni di importazione ed esportazione, la collezione di contatti deve essere passata o ricevuta dalla classe **GestoreFile**. Nonostante GestoreFile non acceda mai ai contenuti interni, si instaura egualmente una forte dipendenza.

Un altro legame importante della classe **Rubrica** è quello con la classe **InterfacciaUtente.** Questo rapporto è inevitabile, poiché l'interfaccia gestisce l'interazione tra l'utente e le funzionalità del sistema. Anche in questo caso si può parlare di un **accoppiamento per timbro**, per le stesse motivazioni già discusse per **GestoreRubrica.**

Infine, la classe Rubrica è legata anche all’interfaccia **OperazioniRubrica**, la quale definisce una serie di metodi che la classe **Rubrica** implementa.
Dato che **Rubrica** è legata solo ai metodi definiti nell'interfaccia e non alle implementazioni concrete, comporta una diminuzione del grado di accoppiamento, portando ad un **accoppiamento per dati**.

Accoppiamenti della classe “PrimaryController”

La classe **PrimaryController** riveste un ruolo centrale nel funzionamento dell’applicazione, poiché intrattiene relazioni fondamentali con diverse altre classi.

Uno dei legami principali è quello con la classe **InterfacciaUtente**, con la quale comunica in maniera diretta, come evidenziato dalla relazione di aggregazione tra le due. Il **PrimaryController** è responsabile dell’avvio e della gestione dell’interfaccia, garantendo il corretto funzionamento delle interazioni con l’utente. Questa relazione è caratterizzata da un **accoppiamento per timbro**, poiché l'**InterfacciaUtente** passa al **PrimaryController** una struttura complessa, nello specifico un’istanza della classe **Rubrica**, che viene utilizzata per gestire i dati dell’applicazione.

Un’interazione simile si osserva tra il **PrimaryController** e la classe **GestoreFile**. Anche in questo caso, la relazione è di **accoppiamento per timbro**, dato che il **PrimaryController** mantiene un attributo di tipo **GestoreFile**. Questo permette di delegare al **GestoreFile** le operazioni relative alla lettura e alla scrittura dei dati su file, semplificando la gestione dei dati persistenti.

Un’altra relazione di rilievo coinvolge il **SecondaryController**. Il **PrimaryController** passa al **SecondaryController** una struttura dati complessa, implicando anch’essa un **accoppiamento per timbro**.

Infine, il **PrimaryController** è legato anche all’interfaccia **OperazioniRubrica**, da cui dipende per utilizzare i metodi che ne derivano. In questo caso, la relazione è caratterizzata da un **accoppiamento per dati**, poiché l’interazione è limitata allo scambio dei soli dati essenziali.

Accoppiamenti Restanti

Nella trattazione precedente, ci si è concentrati principalmente sulle relazioni tra le classi più centrali del sistema. Tuttavia, come evidenziato dal diagramma delle classi, esistono ulteriori relazioni significative che meritano attenzione.

La classe **InterfacciaUtente** non interagisce esclusivamente con **Rubrica** e **PrimaryController**, ma è collegata anche a **GestoreFile**. L’accoppiamento tra queste due classi è di tipo **per timbro**, poiché InterfacciaUtente include un attributo di tipo GestoreRubrica. Quest’ultimo è una struttura dati complessa responsabile della gestione della collezione di contatti e delle relative operazioni.

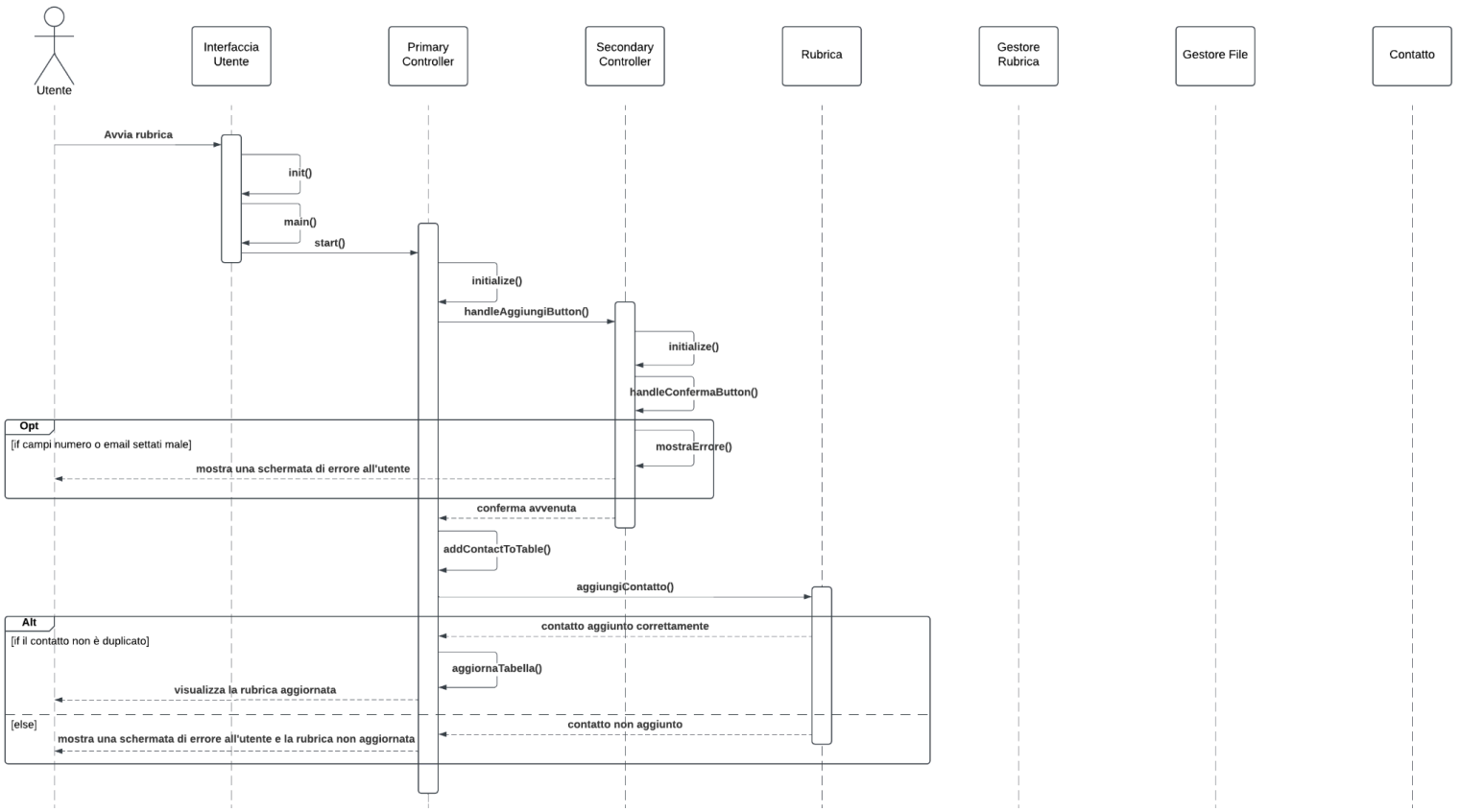
Un’altra relazione significativa è quella tra **SecondaryController** e **OperazioniRubrica**. Anche in questo caso, l’accoppiamento è di tipo **per timbro**, poiché la classe SecondaryController include un attributo di tipo **GestoreRubrica**.

Infine, **SecondaryController** è in relazione diretta con la classe **Contatto**. L’accoppiamento è nuovamente di tipo **per timbro**, poiché **SecondaryController** possiede un attributo di tipo **Contatto**. Questa relazione consente alla classe del controller secondario di comunicare direttamente con l’entità Contatto, garantendo un accesso efficiente ai dati..

Diagrammi sequenziali

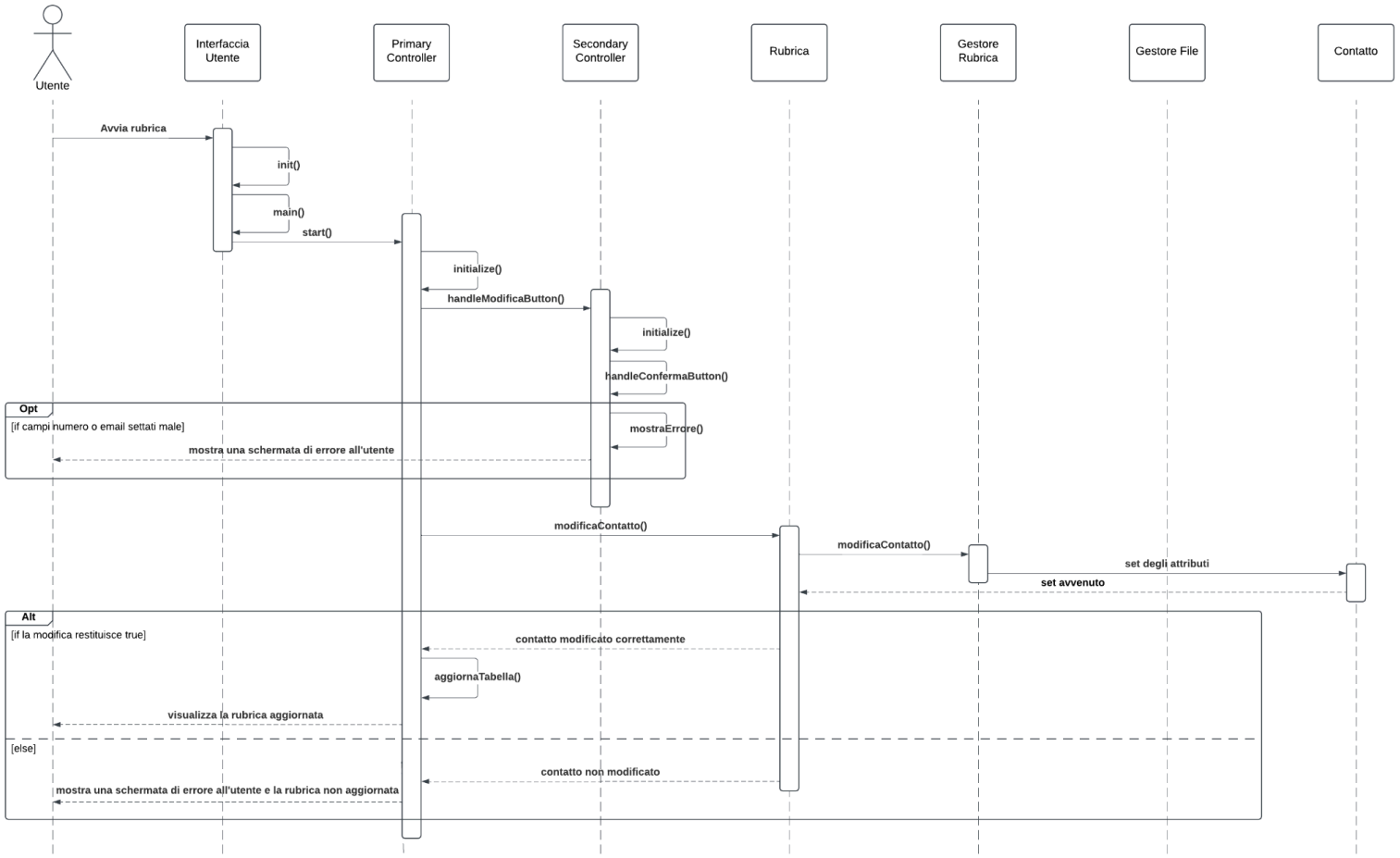
Interazione “Aggiungi Contatto”

Diagramma sequenziale per l'interazione “Aggiungi contatto”



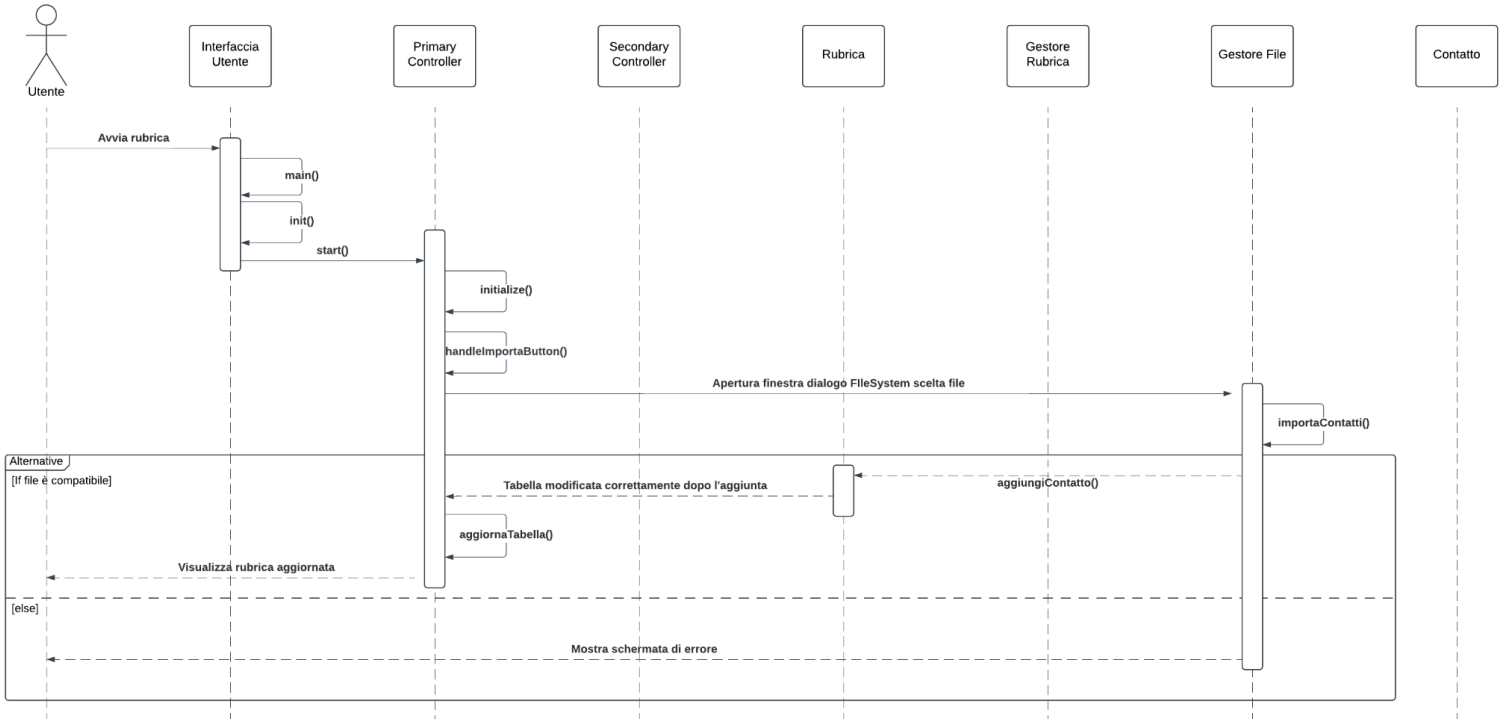
Interazione “Modifica Contatto”

Diagramma sequenziale per l’interazione “Modifica contatto”



Interazione “Importa Contatto”

Diagramma sequenziale per l’interazione “Importa contatto”



I principi di progettazione adottati

Nella progettazione del software per la gestione della rubrica telefonica si è dato grande rilievo ai principi di buona progettazione, con lo scopo di garantire un prodotto manutenibile, riutilizzabile e scalabile.
Questo approccio ha reso il codice, e quindi l'applicazione, semplice ed efficiente.

Un aspetto prioritario è stato il criterio di **semplicità**: ogni classe è stata progettata con metodi brevi e semplici in termini di implementazione.
In aggiunta a quanto detto, la semplicità è stata perseguita anche nella scelta degli algoritmi per risolvere i task: sono stati evitati algoritmi difficili da implementare e complessi da comprendere. Questo ha permesso anche di soddisfare il **principio della minima sorpresa**, offrendo una struttura coerente e chiara che semplifica il lavoro di chi dovrà leggere o manutenere il codice.

Un altro criterio fondamentale utilizzato è stato il principio **DRY** (*Don't Repeat Yourself*)**,** ossia il principio che mira ad evitare la ripetizione delle funzionalità. Ogni classe è stata progettata per implementare metodi dedicati a precisi task, evitando così sovrapposizioni e ripetizioni con le funzionalità delle altre classi.
Tutto ciò ha permesso di organizzare il codice in moduli separati e questo, oltre a favorire un basso accoppiamento, ha rispettato anche il **principio di separazione delle preoccupazioni**.

Infine, è stato applicato il principio **YAGNI** (*You Aren't Gonna Need It*), concretizzatosi con l'eliminazione delle funzionalità non essenziali che avrebbero potuto introdurre un certo grado di complessità, oltre che una scarsa chiarezza. Concretamente, sono stati tolti dei controlli molto specifici, come il riconoscimento di duplicati tra numeri che differiscono per uno spazio o per il + del prefisso, oppure delle schermate di errore o informazioni in alcuni casi, che avrebbero solamente reso meno fluido il programma.

Nella progettazione sono stati seguiti diversi principi del **SOLID**, che costituiscono un insieme di linee guida per una buona programmazione ad oggetti. Tuttavia, non tutti i principi sono stati completamente rispettati.

1. **S** (Single Responsibility Principle)
   Ogni classe è stata progettata per avere una singola responsabilità, anche in linea con il principio **DRY:** questo lo si può notare sia con la classe **Contatto,** che si occupa esclusivamente della gestione delle informazioni di un contatto, sia con le classi **GestoreRubrica** e **Rubrica,** che si occupano della gestione di una collezione di contatti. Lo stesso vale per il **GestoreFile**, le classi dei **Controller** e dell'**interfacciaUtente**.
   Alla luce di quanto detto, questo criterio è da ritenersi soddisfatto.

2. **O** (Open-Closed Principle)
   Il codice è stato strutturato per essere **aperto all'estensione** ma **chiuso alla modifica**: sostanzialmente, nuove funzionalità possono essere aggiunte senza alterare il codice esistente. Esempi lampanti dell'aderenza a questo principio sono le

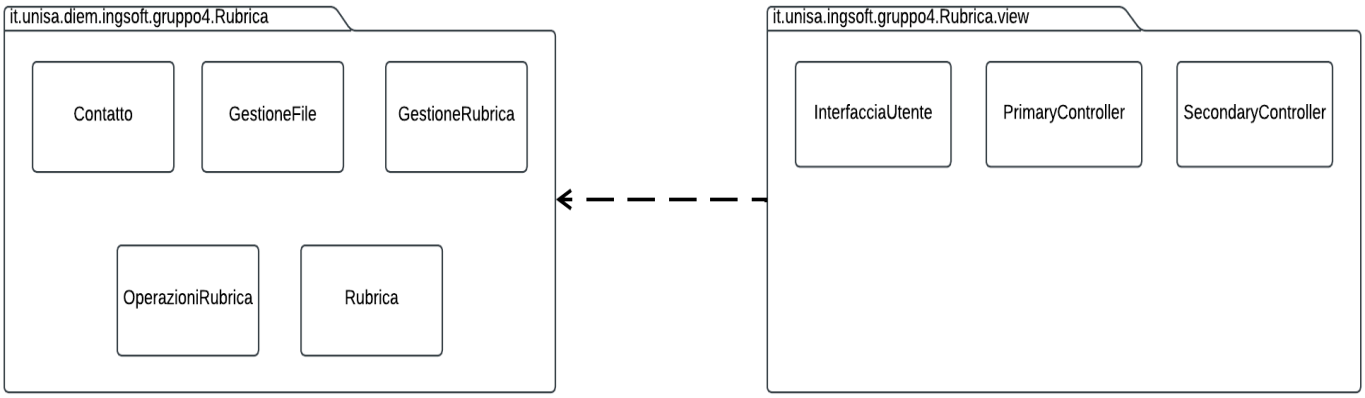
classi **GestoreFile** e **GestoreRubrica**, oltre all'utilizzo dei metodi dell'interfaccia **OperazioniRubrica**.
Dunque, anche questo principio si può ritenere soddisfatto.

3. **L** (Liskov Substitution Principle)
   Questo principio non è direttamente rilevante nella progettazione, non essendoci relazioni di **ereditarietà** tra le classi.
   Pertanto, potenzialmente non esistono problemi di sostituzione di sottoclassi che possano compromettere il comportamento del programma.
   Si è preferito privilegiare l'associazione rispetto all'ereditarietà, in modo da ridurre l'accoppiamento fra le classi.

4. **I** (Interface Segregation Principle)
   Il progetto si avvicina al rispetto del principio di segregazione delle interfacce, grazie all'uso di **OperazioniRubrica**, che separa chiaramente la gestione dei contatti dalle implementazioni concrete.

5. **D** (Dependency Inversion Principle)
   Il progetto aderisce parzialmente al DIP. Questo perché le operazioni sulla rubrica seguono il principio grazie all'interfaccia **OperazioniRubrica**, ma le operazioni come l'importazione e l'esportazione violano il DIP perché dipendono da implementazioni concrete (GestoreFile).

DIAGRAMMA DEI PACKAGE



Il progetto è organizzato in due package principali:

1. **it.unisa.diem.ingsoft.gruppo4.Rubrica**: Contiene le classi per la gestione dei contatti e delle operazioni relative alla rubrica.
2. **it.unisa.diem.ingsoft.gruppo4.view**: Contiene le classi per la gestione dell'interfaccia grafica e il controllo delle interazioni utente.

L'organizzazione del codice in package favorisce la separazione delle responsabilità, migliorando la leggibilità e la manutenibilità del progetto.

Relazioni tra i Package

Dipendenze:

Il package **View** dipende dal package **Rubrica** per gestire i dati dei contatti e utilizzare le funzionalità di persistenza e gestione della rubrica.

Mentre il package **Rubrica** non dipende dal package **View**, rendendolo un modulo riutilizzabile, indipendente dall'interfaccia grafica.