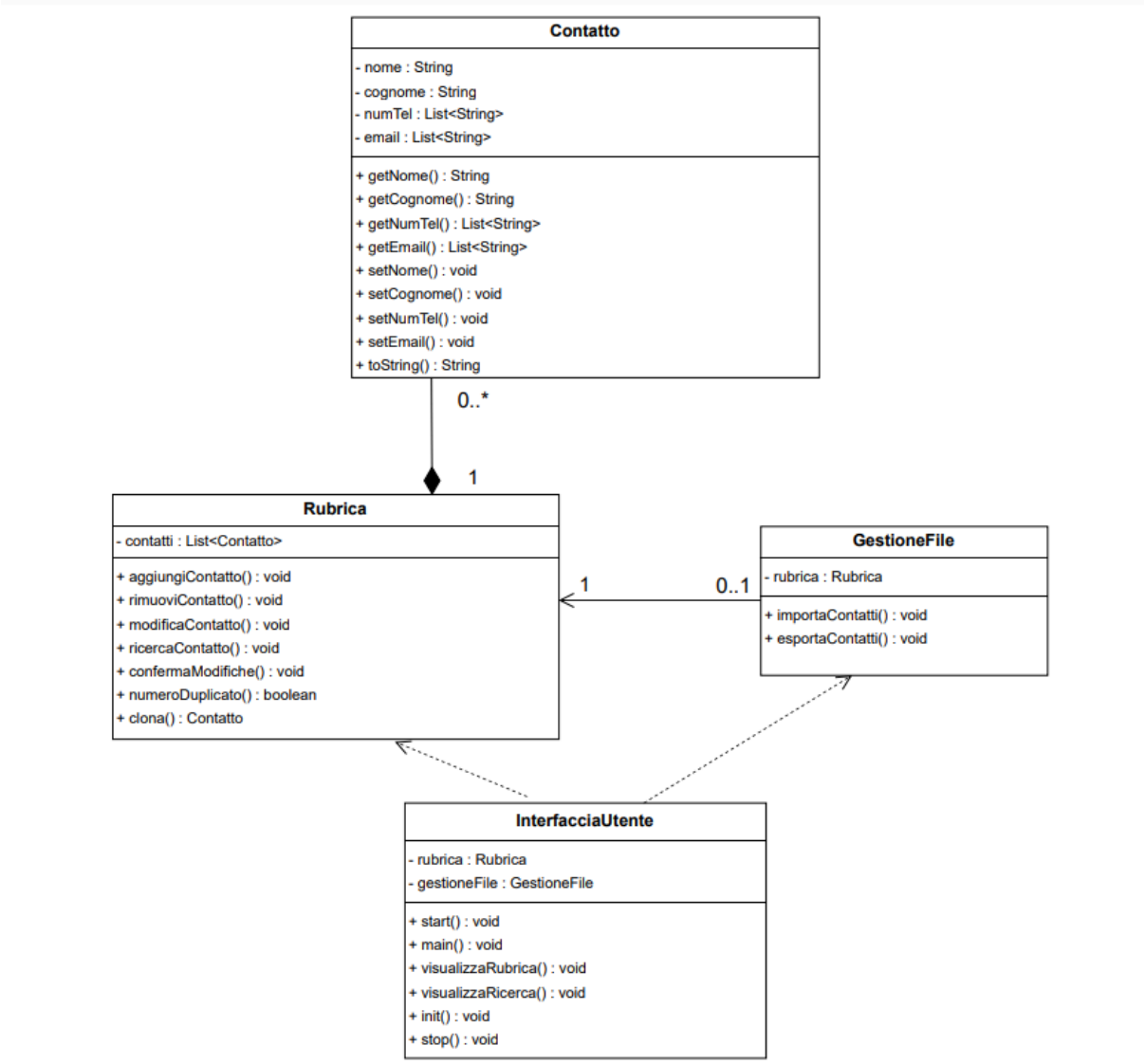
Documentazione del design

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 quattro classi:

1. Contatto
2. Rubrica
3. Gestione File
4. Interfaccia Utente

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.

Classe ‘Rubrica’

La classe Rubrica contiene una collezione di oggetti di tipo Contatto, organizzata in una struttura dati List<Contatto>. In questo modo, la classe Rubrica si occupa solamente della gestione dei contatti, e non si occupa di nessun dettaglio implementativo di essi.
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 uno degli attributi del contatto;
- **cercaContatto()**, che effettua una ricerca sulla collezione di Contatti, in base alla sottostringa messa come ingresso;
- **salvaModifiche()**, che richiede all’utente di confermare o meno le modifiche apportate (inteso come aggiunta, modifica o rimozione di un contatto).
- **numeroDuplicato()**, che verifica se il numero inserito è già presente in un altro contatto o meno;
- **clona()**, che restituisce un clone del contatto passato come parametro.

Classe ‘GestioneFile’

La classe in esame gestisce le funzionalità di import ed export della rubrica.

La classe implementa tre metodi:

- **importaContatti()**, per importare, da un file verso la rubrica, un insieme di contatti;
- **esportaContatti()**, per esportare su un file i contatti presenti nella rubrica.

Classe ‘InterfacciaUtente’

La quarta ed ultima classe del progetto è quella che provvede alla realizzazione dell’interfaccia utente. Questa classe si occupa di mostrare all’utente che interagisce con il sistema un’interfaccia, che permette di rendere l’utilizzo più semplice.

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;
- **visualizzaRubrica(),** per poter mostrare all’utente la rubrica;
- **visualizzaRicerca(),** per poter mostrare all’utente il risultato della ricerca.

Le relazioni tra classi

La classe **Rubrica** 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 classe **GestioneFile** 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 dipendenza dalle classi **Rubrica** e **GestioneFile**, in quanto utilizza i metodi delle due classi da cui dipende.

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.

Classe 'Rubrica'

La classe Rubrica, invece, è stata pensata in modo tale che svolga 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 'GestioneFile'

La classe GestioneFIle 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'

Come quarta ed ultima classe, si menziona la classe InterfacciaUtente, centrale per la gestione dell'interazione tra l'utente e la rubrica, che funge da punto di comunicazione tra l'utente e le funzionalità offerte dalle altre classi.
La classe svolge funzionalità che ne deteriorano leggermente la coesione: in questa classe, infatti, si può parlare di **coesione temporale/coesione procedurale.** Si può dire che sia questo il livello di coesione perché tutti i metodi di questa classe devono essere eseguiti con un certo grado di parallelismo.
Per di più, il metodo main() esegue successivamente il metodo start() che a sua volta esegue la visualizzazione della rubrica, garantendo di fatto una vera e propria sequenzialità.

Accoppiamento tra classi

Accoppiamento tra Rubrica e Contatto

La prima relazione che si può notare nel diagramma delle classi è quella tra **Rubrica** e **Contatto**. Abbiamo ottenuto un basso accoppiamento grazie all'utilizzo della collection List <Contatto> , che gestisce i contatti in modo generico tramite i metodi get/set. Si può parlare, in questo caso, di un **accoppiamento per dati,** in quanto la rubrica gestisce una collezione

di oggetti di tipo **Contatto**, ma lo scambio di informazioni avviene solo attraverso i metodi setter e getter di **Contatto**.

Accoppiamento tra GestioneFile e Rubrica

La classe **GestioneFile** ha una dipendenza diretta da **Rubrica** per poter svolgere le sue funzionalità, quali l'importazione e l'esportazione.
L'accoppiamento in questa classe, rispetto alla precedente, è leggermente peggiore: in questo specifico caso possiamo parlare di **accoppiamento per timbro**. Quando i metodi della classe devono svolgere l'importazione o l'esportazione, devono passare (o prendere) l'intera collezione di contatti. Anche se non accedono ai dettagli interni dei contatti, il fatto che dipendono dalla collezione crea una dipendenza più forte rispetto all'accoppiamento tra **Rubrica** e **Contatto**.

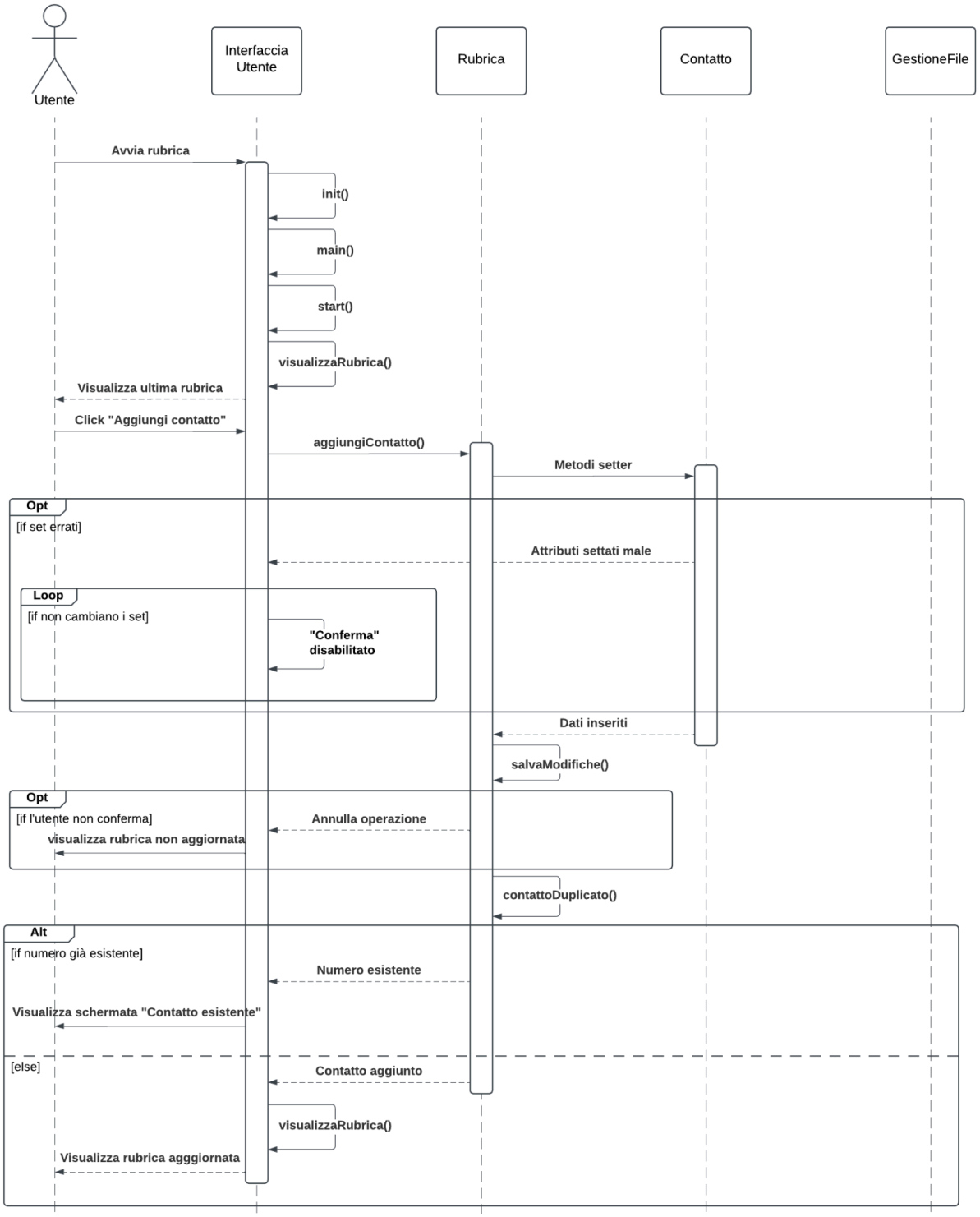
Accoppiamento tra InterfacciaUtente e Rubrica/Gestione File

La classe **InterfacciaUtente** ha dipendenze dirette con **Rubrica** e **GestioneFile**. Questo è inevitabile perché deve gestire l'interazione tra l'utente e le funzionalità del sistema. Per la relazione tra InterfacciaUtente e Rubrica possiamo parlare di **accoppiamento per timbro**, e le motivazioni sono alquanto simili a quelle discusse tra **GestoreFile** e **Rubrica**. Un discorso diverso si può fare per la relazione tra InterfacciaUtente e GestoreFile: in questo caso si può parlare di un **accoppiamento per dati**, siccome il modulo InterfacciaUtente si limita a richiamare metodi del gestore file e quindi si scambia solo dei dati con il modulo di GestioneFile.

Diagrammi sequenziali

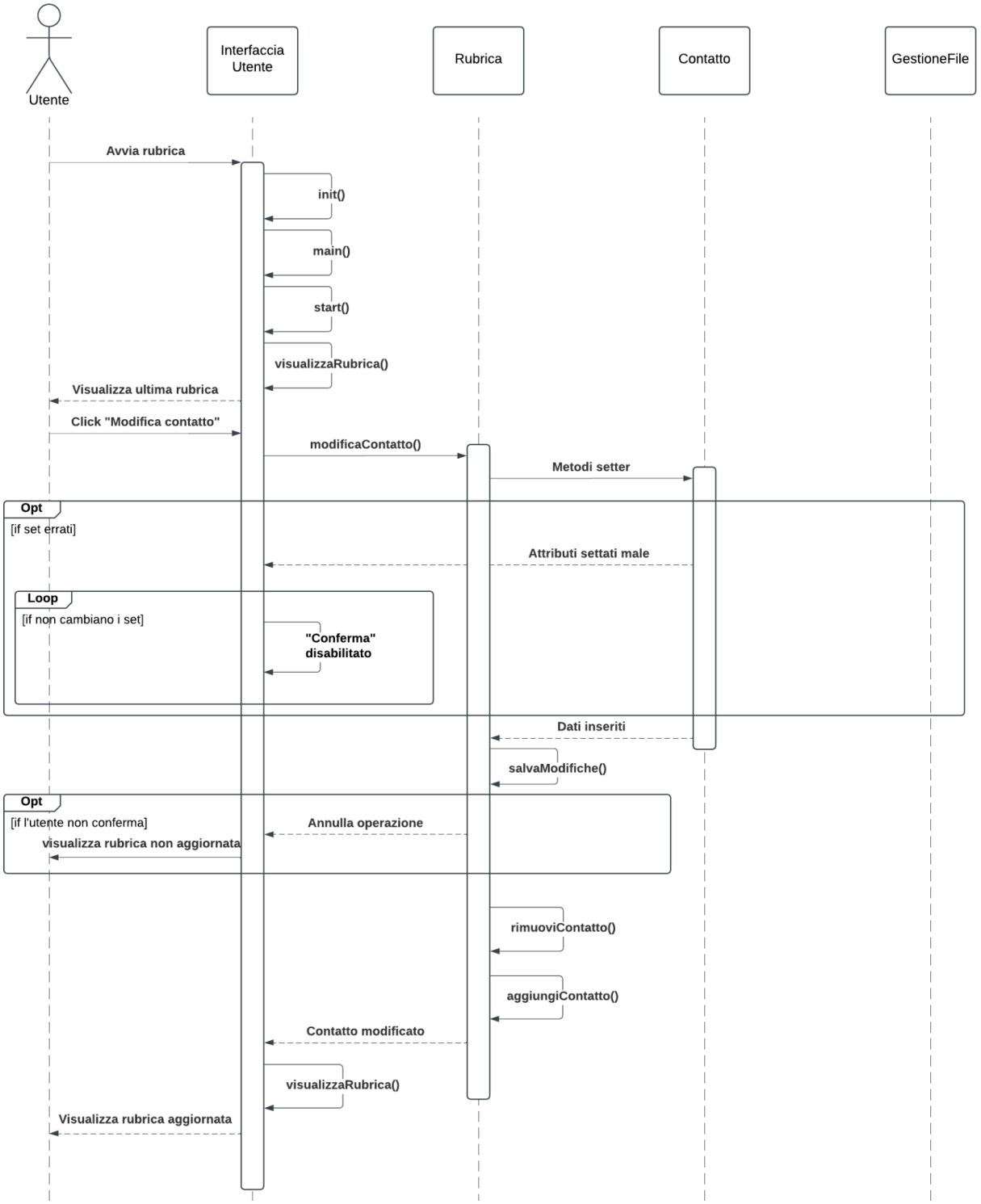
Interazione “Aggiungi Contatto”

Diagramma sequenziale per l'interazione “Aggiungi contatto”



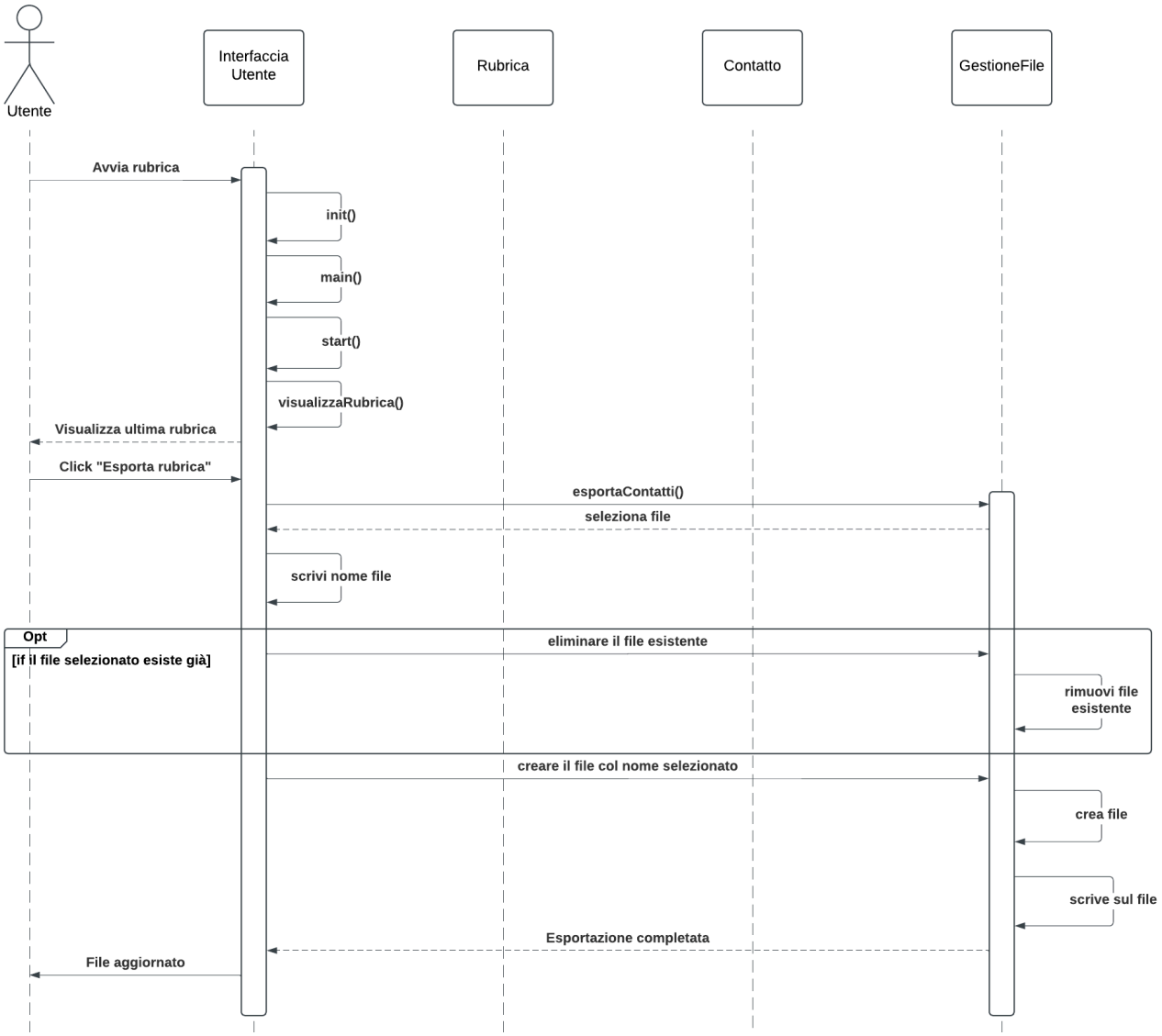
Interazione “Modifica Contatto”

Diagramma sequenziale per l’interazione “Modifica contatto”



Interazione “Esporta rubrica”

Diagramma sequenziale per l’interazione “Esporta rubrica”



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.

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 la classe **Rubrica,** che si occupa della gestione di una collezione di contatti.
   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.
   Quindi, 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.

4. **I** (Interface Segregation Principle)
   Il principio in questione non è stato esplicitamente violato poiché non sono state utilizzate interfacce per specifiche funzionalità durante la progettazione.

5. **D** (Dependency Inversion Principle)
   Quest'ultimo è l'unico principio del **SOLID** che non è stato rispettato, in quanto le dipendenze principali sono tra classi concrete e non astratte. Le scelte progettuali che hanno portato a questa conseguenza sono legate al tentativo di mantenere le relazioni tra le classi dirette, per offrire un livello di semplicità nella struttura generale del sistema.