

Corso di Laurea Triennale in Ingegneria Informatica

Progetto di Ingegneria del Software

Prof. Pasquale Foggia

Anno Accademico 2024/2025

Documentazione sulla Progettazione

**Sistema Software per la gestione di una Rubrica
Telefonica**

Versione 2 del 15/12/2024

Autrici

Gruppo 21:

Crisci Chiara

Del Sorbo Palma

Granturco Roberta

Graziuso Emanuela

INDICE

1 INTRODUZIONE	2
1.1 Descrizione del Documento	2
1.2 Obiettivi del Documento	2
2 DESIGN ARCHITETTURALE	2
2.1 Scelte Architettureali	2
2.1.1 Tecnologie e Strumenti	3
3 DESIGN DI DETTAGLIO	5
3.1 Diagramma dei Package	5
3.2 Diagramma delle Classi	6
3.2.1 Diagramma Concettuale	6
3.3 Interazioni dettagliate	7
3.3.1 Contatto	7
3.3.2 Controlli Validità	8
3.3.3 Rubrica - Contatto	9
3.3.4 Rubrica - GestorePersistenzaDati	12
3.3.5 Rubrica - Contatto - OperatoreFile	14
3.4 Diagrammi delle Sequenze	16
3.4.1 Esportazione Rubrica	16
3.4.2 Salvataggio dati	17
3.3.3 Caricamento dati	18

1 INTRODUZIONE

1.1 Descrizione del Documento

Il documento di design di un progetto software vuole offrire una prima panoramica delle scelte progettuali effettuate, discutendone innanzitutto ad un livello architetturale, per poi approfondire tali scelte ad un livello di maggiore dettaglio.

In secondo momento il documento vuole riportare l'attenzione alle conseguenze e alle motivazioni delle principali scelte progettuali, in termini di coesione dei componenti che costituiscono il sistema e le principali interazioni tra questi, mediante l'utilizzo dei diagramma di package, diagrammi di classe e dei diagrammi di sequenza.

1.2 Obiettivi del Documento

In questo documento sono illustrate le principali scelte progettuali con l'obiettivo di fornire una visione chiara e giustificata del design dell'applicazione, in modo da facilitare la comprensione e la manutenzione futura.

2 DESIGN ARCHITETTURALE

2.1 Scelte Architettureali

Con un breve focus sulle decisioni architettureali, si intende innanzitutto chiarire che l'applicazione di gestione della rubrica non presenta una complessità tale da richiedere scelte architettureali particolarmente sofisticate. Si tratta comunque di un sistema di dimensioni ridotte, con una richiesta di gestione dati relativamente semplice e di funzionalità che non richiedono interazioni o elaborazioni molto complesse, dunque un'architettura troppo complessa sarebbe poco giustificata.

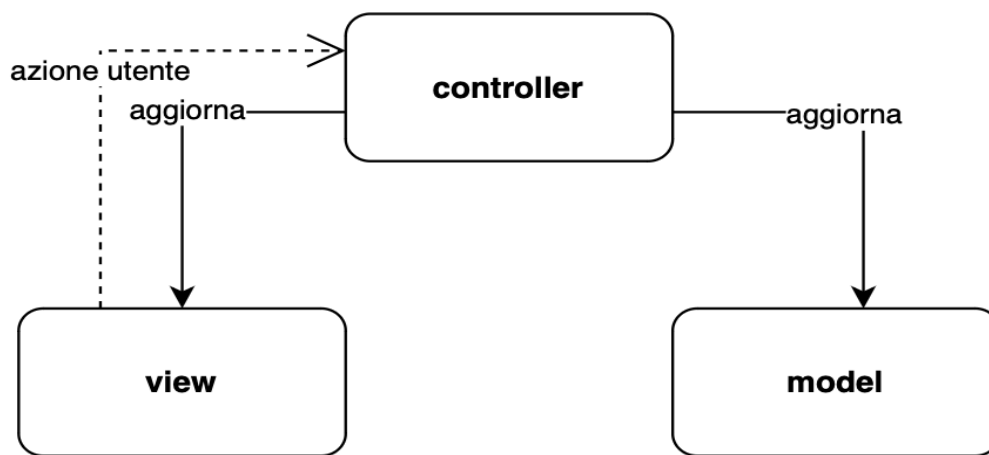
Tuttavia, per migliorare la modularità e la manutenibilità del codice, si è optato per il pattern architetturale Model-View-Controller (MVC).

Questo approccio aiuta infatti a mantenere una separazione chiara tra le diverse componenti dell'applicazione, ciascuno ben coeso, grazie alla suddivisione delle responsabilità tra:

- **Model:** contiene la logica dell'applicazione e la gestione dei dati, centrando le funzionalità di manipolazione di questi.

- **View:** responsabile della presentazione dei dati a livello di interfaccia utente, invia i dati in input, in maniera appropriata, al controller.
- **Controller:** funge da intermediario tra la view e il model in quanto gestisce gli input dell'utente, comunica con il model per ottenere o modificare i dati e aggiorna la view di conseguenza; dunque trasforma le interazioni dell'utente in azioni sui dati.

La comunicazione tra tali componenti risulta inevitabile, ma si è pur sempre ottenuto un accoppiamento basso tra essi; d'altronde, l'approccio seguito facilita l'isolamento della logica dell'applicazione rispetto all'interfacciamento con l'utente, rendendo la prima completamente indipendente da come viene progettata la seconda.



2.1.1 Tecnologie e Strumenti

L'applicazione per la gestione della rubrica è stata progettata pensando a un'implementazione realizzata utilizzando il linguaggio di programmazione Java, scelta motivata dalla portabilità e dall'ampia libreria di supporto che offre il linguaggio: framework Collections, framework per applicazioni desktop (JavaFx) e I/O API. L'uso di Java Collections Framework per la gestione di strutture dati complesse ha contribuito a migliorare principalmente gli attributi di qualità, quali:

- **efficienza:** grazie alle strutture dati ottimizzate che esso fornisce, permette la riduzione dei tempi di ricerca, ordinamento e manipolazione dei contatti nella rubrica.

- **modularità:** delegando le responsabilità di gestione a componenti già ottimizzati e testati, ne permette una gestione centralizzata che si sofferma solo sulla logica applicativa.
- **manutenibilità:** sia consecutivamente alla modularità sia grazie ai metodi aggiuntivi che il framework offre.

La scelta di realizzare l'interfaccia utente con JavaFX è invece mirata ad incrementare l'usabilità dell'applicazione; questo è reso possibile creando interfacce utente visivamente piacevoli, che possono essere personalizzate attraverso CSS, permettendo di rendere l'applicazione più attraente all'utente, mantenendola sempre intuitiva.

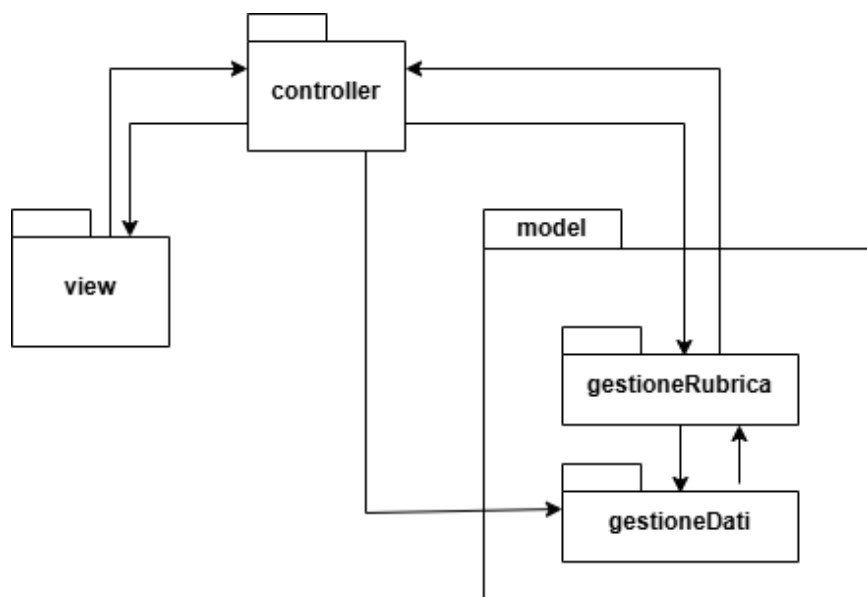
Inoltre con la gestione automatica della memoria tramite il Garbage Collector, che tale linguaggio di programmazione offre, è stata facilitata la gestione delle risorse al fine di ottenere una maggiore leggibilità del codice ed efficienza nell'uso delle risorse.

Infine, Java supporta pienamente la programmazione orientata agli oggetti (OOP), che permette di modellare in modo naturale l'applicazione, giungendo ad un sistema progettato con una buona separazione delle responsabilità e permettendo di ottenere una progettazione di qualità, visti i guadagni qualitativi che una buona separazione delle responsabilità comporta.

3 DESIGN DI DETTAGLIO

Nel processo progettuale di dettaglio, in seguito a una prima, più generica, decomposizione modulare secondo un approccio funzionale, è stato seguito principalmente un approccio Object-Oriented, supportato naturalmente da Java.

3.1 Diagramma dei Package



Il diagramma dei package è l'esito della realizzazione di un processo di decomposizione funzionale del sistema; è osservabile infatti come l'applicazione sia strutturata in diversi package, ciascuno responsabile di un aspetto specifico del sistema: interfaccia utente, logica di controllo, logica applicativa, gestione dei dati. Questa suddivisione consente non solo di ridurre la complessità, ma anche di migliorare la manutenibilità del sistema. Quest'ultimo risulterà facile da mantenere e facilmente estendibile, poiché ogni package può essere modificato indipendentemente, rispettando i principi di separazione delle preoccupazioni:

- L'interfaccia utente è progettata separatamente, consentendo modifiche al layout o aggiunta di nuove funzionalità, senza influire sulla logica applicativa.
- La logica di controllo separa la gestione degli eventi dall'interfaccia visiva e dalla logica applicativa, permettendo un'alta modularità e una più facile gestione delle operazioni. In caso di necessità di modificare la logica di interazione, questa può essere fatta senza impattare direttamente altre parti dell'applicazione.
- La logica applicativa (contatti e rubrica) è separata dalla gestione dei dati e dall'interfaccia utente, facilitando la manutenibilità, inclusa la gestione di

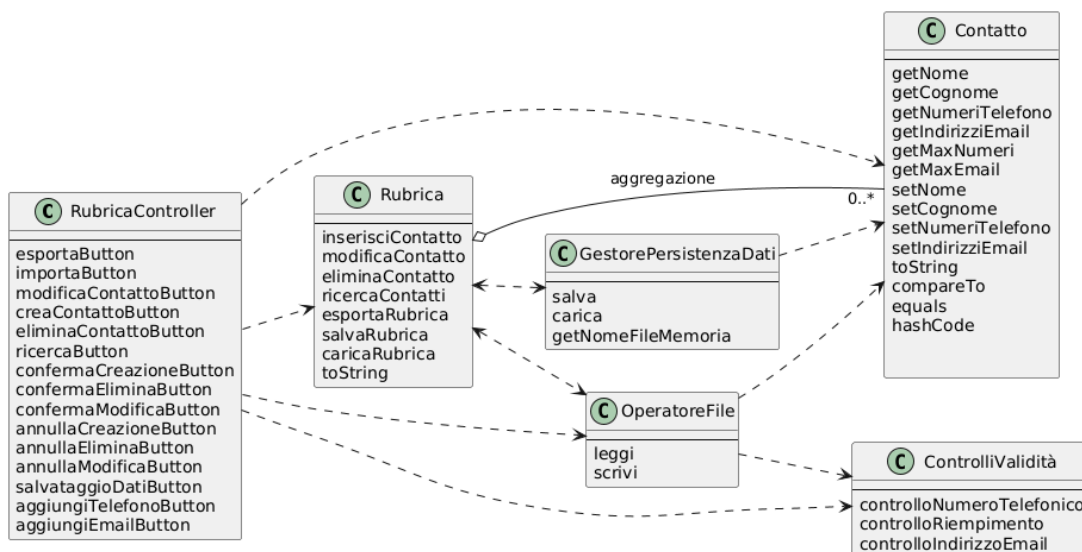
modifiche (ad esempio se si desidera aggiungere nuovi campi al contatto) e la testabilità.

- La gestione dei dati (come l'importazione e l'esportazione dei contatti dalla rubrica, il salvataggio dei dati in formato persistente e il caricamento dei dati salvati) è isolata dalla logica applicativa.

Ciò facilita soprattutto la manutenibilità del sistema in merito a misure scelte per la persistenza e la gestione dei dati (ad esempio file di testo o database), e future espansioni di funzionalità di importazione/esportazione dati (ad esempio l'aggiunta di opzioni di scelta del formato del file di importazione/esportazione). La gestione centralizzata dei dati semplifica anche il debug, poiché il codice responsabile del caricamento e salvataggio dei dati è contenuto in una singola area.

3.2 Diagramma delle Classi

3.2.1 Diagramma Concettuale



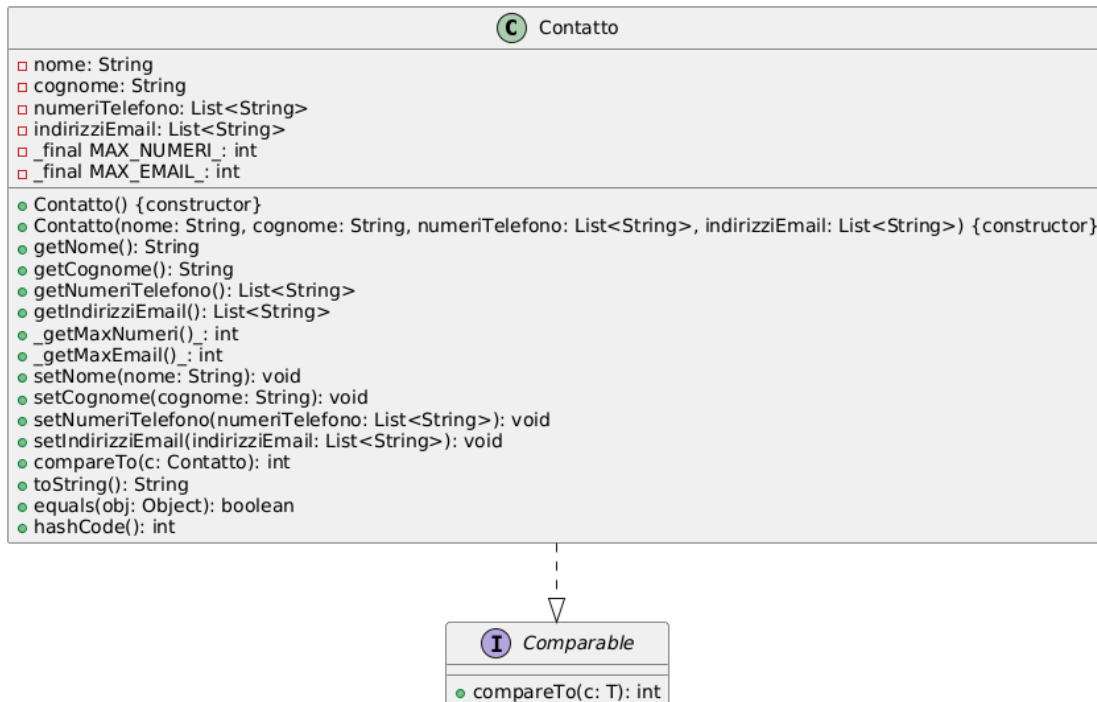
Il presente diagramma delle classi riflette l'approccio Object-Oriented su cui ci si è soffermati secondariamente in fase di progettazione, evidenziando le relazioni tra le classi principali del sistema.

Nei paragrafi successivi, è possibile analizzare in maniera dettagliata le interazioni di tali classi, analizzando i livelli di coesione e accoppiamento che sono stati raggiunti e, in particolar modo, quali principi di progettazione sono stati rispettati in fase di design di dettaglio.

3.3 Interazioni dettagliate

3.3.1 Contatto

Focalizziamo innanzitutto la nostra attenzione sull'unità minima dell'applicazione di gestione rubrica: la classe Contatto.



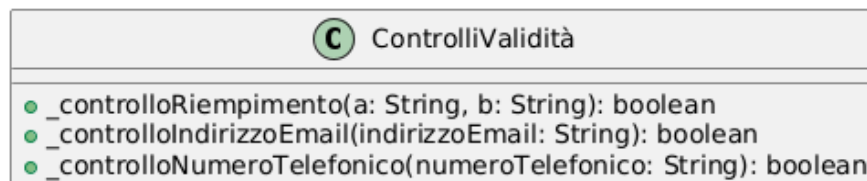
- **Contatto:** rappresenta un'entità di base, che possiede tutti gli attributi che un contatto di rubrica può prevedere (come nome, cognome, etc). La sua responsabilità è limitata alla creazione e inizializzazione del contatto e alla gestione dei propri dati (metodi getter/setter).
 - **Coesione** → tale classe ha una coesione alta e di tipo funzionale, poiché tutte le sue operazioni si concentrano sul trattamento dei dati di un singolo contatto. Ciò è stato ottenuto cercando di evitare di realizzare in tale classe le logiche di validazione dei dati del contatto, che sono state delegate ad un'altra classe (la classe ControlliValidità).

La classe **Contatto** implementa l'interfaccia **Comparable**, offerta dalle librerie standard di Java, permettendo di definire un criterio naturale di ordinamento. Questa rappresenta una scelta progettuale fondamentale, soprattutto in un'applicazione che gestisce una rubrica e che richiede funzionalità di ordinamento dei contatti (eventualmente utile per la visualizzazione ordinata della rubrica, desiderata sia per motivi di usabilità dell'applicazione che in quanto un requisito esplicitamente richiesto dal cliente). In generale ciò permette il riutilizzo

della classe Contatto in una collezione ordinata, in cui l'ordinamento può essere gestito automaticamente quando i contatti vengono inseriti.

Ciò riduce, inoltre, l'accoppiamento tra il modello e l'interfaccia utente, poiché l'ordinamento può essere effettuato senza modificare il codice di visualizzazione.

3.3.2 Controlli Validità



➤ **ControlliValidità**: si occupa di fornire semplicemente i servizi di validazione dei dati utili alle altre classi per la creazione di oggetti contatto, utili affinché ogni contatto risulti sempre in uno stato coerente a ciò che l'oggetto stesso vuole modellare (utili, cioè, al rispetto delle invarianti di classe Contatto). La classe ControlliValidità prevede, dunque, metodi che permettono di verificare che i dati siano correttamente formattati o che rispettino determinati criteri.

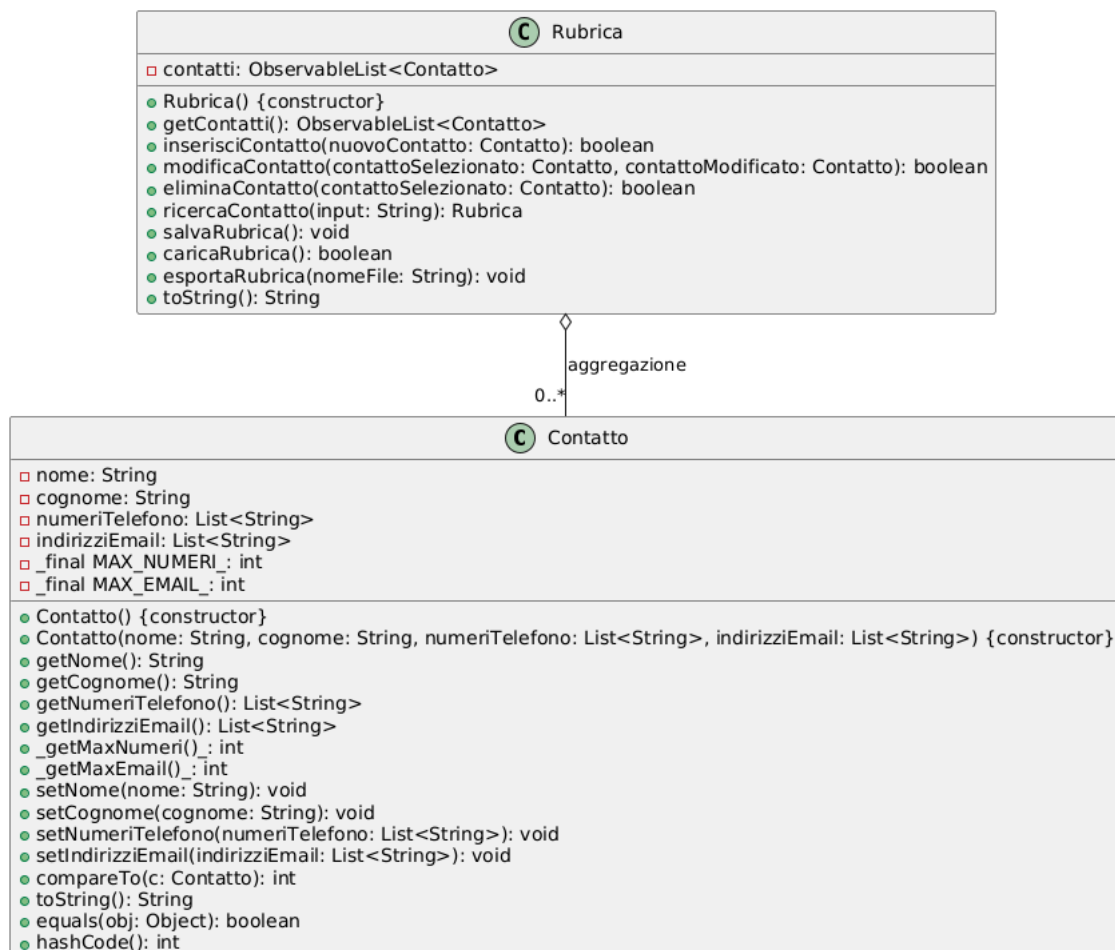
- **Coesione** → tale classe presenta quindi, chiaramente, una coesione funzionale. I suoi metodi sono stati pensati come servizi forniti dalla classe, che implementano la logica di controllo sui dati passati, senza doverli prelevare da un oggetto di tipo Contatto. Ciò rende, quindi, la classe ControlliValidità indipendente dalla classe Contatto dal punto di vista implementativo.

La scelta di progetto mira alla riusabilità del codice di controllo dati per la creazione di un contatto, e allo stesso tempo a ridurre i costi computazionali dei metodi della classe contatto stessa: i metodi di contatto non si preoccupano di valutare che i dati forniti dal chiamante per la modifica/creazione di un contatto possano portarlo a uno stato incoerente; sarebbe molto più efficiente, invece, che il chiamante di tali metodi si preoccupi di fornire dati validi secondo le aspettative di coerenza dei dati definite dalla classe contatto. Al chiamante è data la possibilità di fare ciò invocando i metodi di controlli validità, al fine di riutilizzare il codice il più possibile.

La realizzazione di ciò è stata poi basata fortemente sul principio di separazione delle responsabilità, tale che la classe Contatto non dipende dalle logiche di validazione dei suoi dati.

La scelta aiuta la riusabilità e la manutenibilità delle classi: laddove si voglia realizzare un controllo più restrittivo sui dati del contatto, sarà sufficiente implementare la logica di controllo nella classe ControlliValidità.

3.3.3 Rubrica - Contatto



Proseguiamo soffermandoci sul gruppo di classi utili alla realizzazione di una rubrica: la classe `Rubrica` e la classe `Contatto`.

➤ **Classe Rubrica:** rappresenta una collezione di contatti.

La scelta di implementare tale collezione con una lista, sfruttando dunque il framework `Collection` di Java, ha evidenti vantaggi in termini di efficienza, leggibilità e manutenibilità del codice. Renderla una lista osservabile è stata una scelta motivata dall'intento di voler facilitare e automatizzare le operazioni di aggiornamento della rubrica che viene visualizzata in interfaccia, riducendo così l'accoppiamento tra il modello e la view dell'architettura.

La classe fornisce i metodi per la gestione della rubrica, tramite operazioni come l'aggiunta, la rimozione, la modifica e la ricerca dei contatti; ciò rende la classe

Rubrica molto coesa. L'interazione tra la classe Contatto e la classe Rubrica offre un'ottima separazione delle responsabilità, rispettando principi fondamentali di coesione e accoppiamento. Tale separazione verte sui seguenti attributi di qualità: la manutenibilità, la modularità e la testabilità.

- **Coesione** → si noti come la classe Rubrica preveda, inoltre, metodi come `esportaRubrica`, `salvaRubrica` e `caricaRubrica`; sebbene tali metodi amplino le funzionalità della classe, oltre alla sola gestione della collezione di contatti che essa rappresenta, ciò non ha un forte impatto sulla coesione della classe Rubrica. Si tenga comunque in considerazione che la progettazione del sistema ha previsto che questi metodi non facciano altro che delegare la logica di persistenza o di esportazione dati a classi separate, per favorire la separazione delle preoccupazioni e, di conseguenza, migliorare la manutenibilità e la testabilità della classe Rubrica.

Tuttavia, bisogna specificare che, in funzione dell'inserimento di tali ulteriori metodi nella classe Rubrica, essa giunge a possedere una coesione comunicazionale, in quanto essenzialmente offre funzionalità che lavorano sugli stessi dati.

Tale coesione è tuttavia accettabile: la classe Rubrica continua ad essere la responsabile principale per la gestione della rubrica nel suo complesso, cioè della gestione dei contatti di rubrica e delle operazioni relative al flusso di dati tra la rubrica e l'esterno, dunque non è forte indebolimento in termini di coesione.

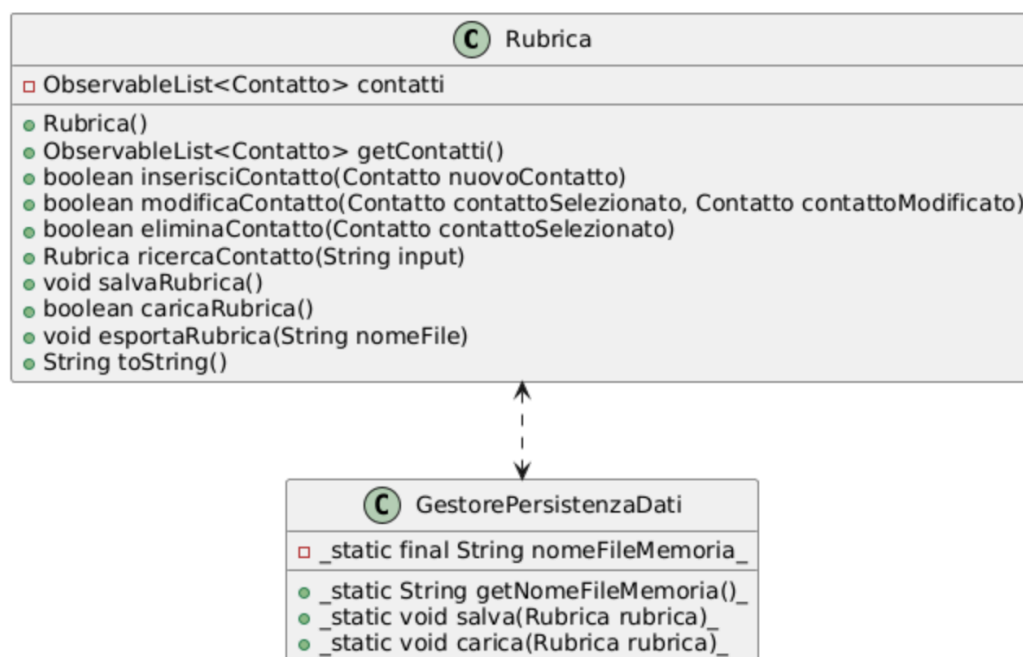
- **Accoppiamento** → la classe Rubrica ha un'inevitabile dipendenza con la classe Contatto; trattasi, però, di un accoppiamento per dati, essendo tale dipendenza limitata solo all'accesso dei dati di contatto, facilmente realizzabile attraverso metodi `getter/setter`, forniti dalla classe Contatto.

Inoltre la Rubrica non necessita di occuparsi della validazione dei contatti, in quanto i contratti dei metodi della classe Rubrica prevedono, laddove necessario, il passaggio di un contatto, già creato, dal client di tale classe. Cioè la rubrica non dovrà preoccuparsi che il contatto da inserire/aggiornare sia in uno stato coerente, in quanto ciò dovrà essere stato garantito dal suo client all'atto di invocazione dei metodi di creazione/modifica della classe contatto, sul contatto che avrà poi passato ai metodi di Rubrica. Il client di Rubrica avrà a sua volta dovuto rispettare i contratti dei metodi di contatto fornendo essi dati già validati, operazione facilitata grazie alle scelte progettuali precedentemente elaborate relativamente alla classe di `ControlliValidità`.

Si sottolinei la scelta di progettazione per la classe di rubrica, di usare una lista osservabile per la lista contatti che essa rappresenta. La lista osservabile rende più semplice mantenere la sincronizzazione tra il modello e la vista. Ad esempio, se si ha una *ObservableList<Contatto>* che rappresenta i contatti della rubrica, ogni volta che un contatto viene aggiunto o rimosso dalla lista, la vista si aggiorna automaticamente per riflettere questi cambiamenti, senza dover scrivere codice aggiuntivo per gestire l'aggiornamento dell'interfaccia utente.

JavaFX fornisce il supporto nativo per l'uso di *ObservableList* nelle *ListView*, *TableView* e altre strutture di visualizzazione. Se la rubrica è visualizzata tramite una *ListView<Contatto>* o una *TableView<Contatto>*, l'uso di una *ObservableList* consente una gestione semplificata dell'aggiornamento dei dati nella vista ogni volta che la lista cambia.

3.3.4 Rubrica - GestorePersistenzaDati



Di seguito, ci soffermiamo sul gruppo di classi utili alla realizzazione della persistenza dati dell'applicazione: la classe *Rubrica* e la classe *GestorePersistenzaDati*.

- **GestorePersistenzaDati:** la classe è chiaramente dipendente dalla classe *Rubrica*, in quanto i suoi servizi necessitano di un riferimento alla rubrica da cui prelevare i dati da salvare o su cui caricare i dati memorizzati in sessioni precedenti.

- **Coesione** → la classe implementa servizi offerti alla classe Rubrica per la gestione permanente dei suoi dati, raggiunge quindi un livello di coesione funzionale, essendo questa la sua sola responsabilità.
- **Rubrica:** similmente, la classe Rubrica dipenderà dalla classe GestorePersistenzaDati, in quanto dovrà usufruire dei servizi da essa forniti per realizzare le proprie funzionalità di salvataggio e caricamento di rubrica.
- **Accoppiamento** → si osserveranno dunque ulteriori dipendenze tra i componenti del sistema, tuttavia inevitabili. Queste generano semplicemente un accoppiamento per dati tra le due classi: le dipendenze aggiuntive non sono preoccupanti, e la scelta di progetto che le ha determinate ha invece i grandi vantaggi, secondo il principio di separazione delle responsabilità, di migliorare la manutenibilità e la testabilità del sistema. Si noti soprattutto come, laddove si voglia cambiare la tecnica di realizzazione della persistenza dati dell'applicazione (per esempio FormatoFile, Database) basterà solamente modificare i metodi implementati in tale classe.

Si intende infine specificare, come scelta di progettazione, che la persistenza dei dati dell'applicazione verrà realizzata attraverso l'uso di un file locale in formato binario.

3.3.5 Rubrica - Contatto - OperatoreFile



Infine, ci focalizziamo sul gruppo di classi utili allo scambio di dati con l'esterno: la classe Rubrica e la classe OperatoreFile.

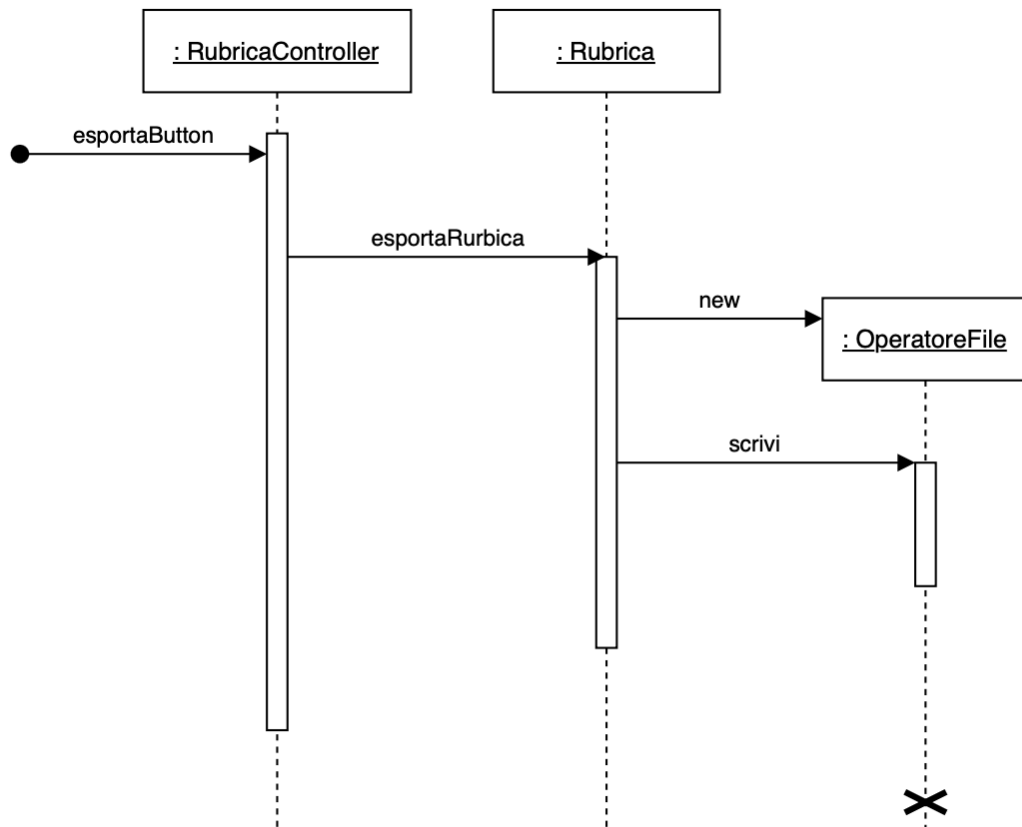
- **OperatoreFile:** la classe implementa operazioni di importazione ed esportazione dei dati di un oggetto di tipo Rubrica. La classe è, in modo evidente, dipendente dalle classi Rubrica e Contatto, in quanto i suoi servizi necessitano un riferimento alla rubrica da cui prelevare i dati da scrivere su file o su cui aggiungere i contatti da leggere da file.
 - **Coesione** → tale classe raggiunge, quindi, un livello di coesione funzionale, essendo la sua sola funzionalità permettere la gestione di flussi di dati tra File locali e la Rubrica gestita in applicazione.

- **Rubrica:** similmente, la classe Rubrica dipenderà dalla classe OperatoreFile, in quanto dovrà usufruire dei servizi da essa forniti per realizzare le proprie funzionalità di esportazione di rubrica.
 - **Accoppiamento** → anche in questo caso si osserva la presenza di ulteriori dipendenze, inevitabili, che generano un accoppiamento per dati tra le classi OperatoreFile, Rubrica e Contatto. Le dipendenze aggiuntive non sono preoccupanti e la scelta di progetto che le ha determinate ha invece gli stessi grandi vantaggi già spiegati per il gruppo di classi Rubrica e GestorePersistenzaDati.

Si intende infine specificare, come scelta di progettazione, che l'importazione e l'esportazione di rubrica verrà realizzata attraverso l'uso di un file locale in formato CSV. Sebbene sarebbe stato possibile facilitare l'estensione delle funzionalità di importazione ed esportazione via file a formati ulteriori per upgrade futuri dell'applicazione, si è voluto invece rispettare il principio di progettazione *YAGNI* (*You Aren't Gonna Need It*) in quanto, limitarsi alla sola richiesta del cliente, senza preoccuparsi dell'estendibilità del sistema per upgrade futuri, consente una notevole semplificazione del sistema, originariamente pensato per essere di dimensioni moderate.

3.4 Diagrammi delle Sequenze

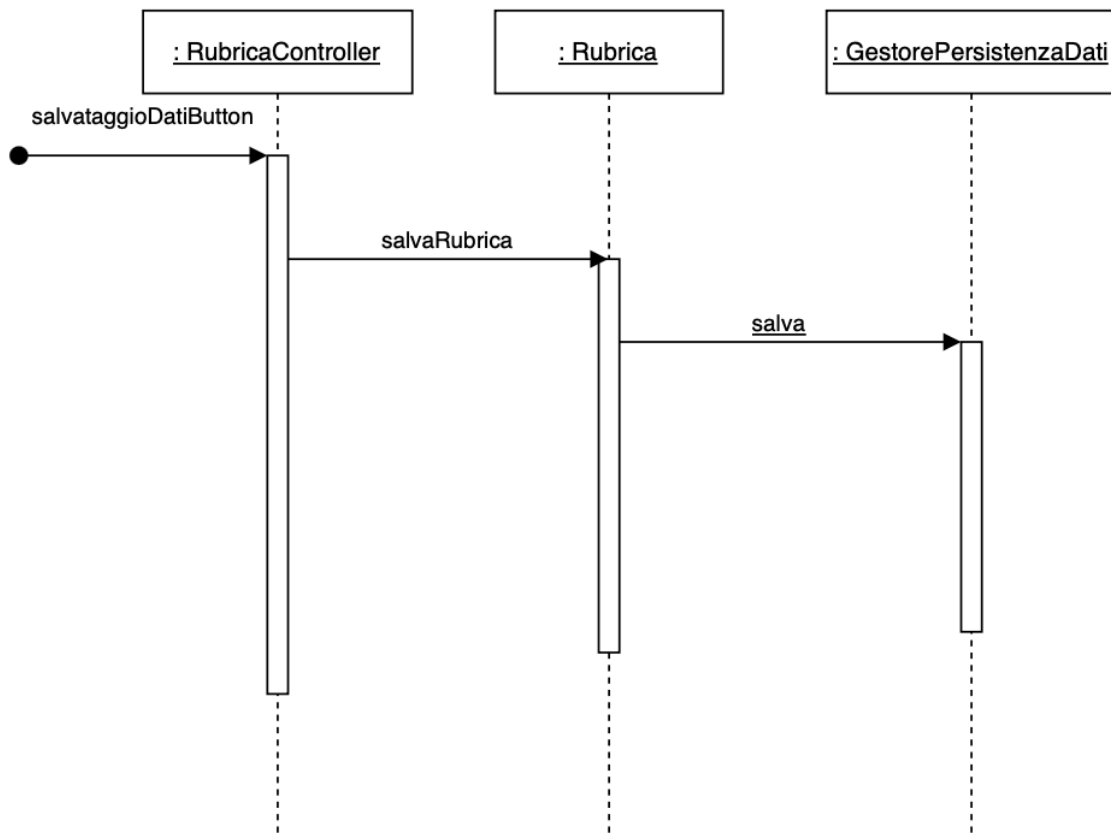
3.4.1 Esportazione Rubrica



DESCRIZIONE

1. Utente → clicca sul bottone collegato a Controller: `esportaButton()`
2. Controller: `esportaButton()` → invoca Rubrica: `esportaRubrica()`
3. Rubrica: `esportaRubrica()` → crea un oggetto `OperatoreFile`
4. Rubrica: `esportaRubrica()` → invoca `OperatoreFile: scrivi()`
5. `OperatoreFile: scrivi()` → si occupa della scrittura dei contatti di rubrica su file

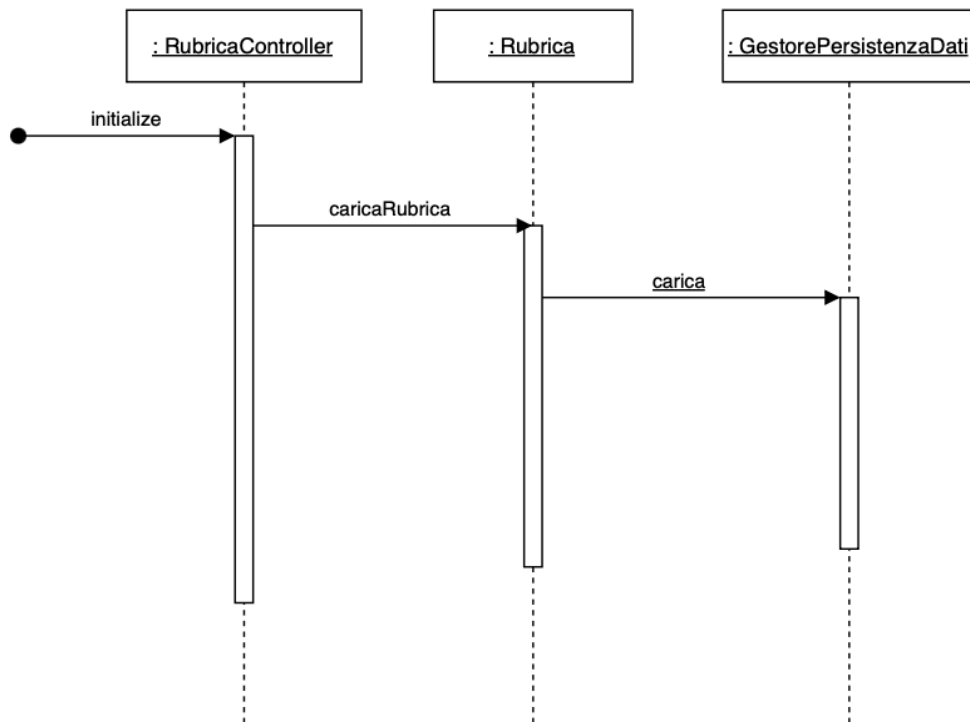
3.4.2 Salvataggio dati



DESCRIZIONE

1. Utente → clicca sul bottone collegato a Controller: `salvataggioDatiButton()`
2. Controller: `salvataggioDatiButton()` → invoca Rubrica: `salvaRubrica()`
3. Rubrica: `salvaRubrica()` → invoca GestorePersistenzaDati: `salva()`
4. GestorePersistenzaDati: `salva()` → si occupa del salvataggio dei contatti di rubrica in modo permanente

3.3.3 Caricamento dati



DESCRIZIONE

1. `Controller:initialize()` → invoca `Rubrica: caricaRubrica()`
2. `Rubrica: caricaRubrica()` → invoca `GestorePersistenzaDati: carica()`
3. `GestorePersistenzaDati: carica()` → si occupa del caricamento dei contatti su rubrica salvati localmente in una precedente sessione