Gruppo 13

Catello Donnarumma **Luca Celentano** **Federico Cervo**
Francesco De Bonis

Project Design Report

2024/2025

Indice

1. Diagramma delle Classi

1.1 Commento al Diagramma Delle Classi

Single Responsibility Principle (SRP)

Ogni classe nel diagramma ha una singola responsabilità, in linea con il principio SRP:

- Contatto: Rappresenta un singolo contatto con metodi e proprietà per la gestione delle sue informazioni.
- Rubrica: Gestisce la collezione di contatti, includendo operazioni di aggiunta, ricerca, eliminazione e modifica, esportazione e importazione.
- Controller (AggiuntaContattoController, ModificaContattoController, VisualizzaSingoloContattoController, MainInterfaceController): Ogni controller si occupa di una specifica funzionalità dell'interfaccia grafica:

  AggiuntaContattoController: Gestisce l'aggiunta di contatti.

  ModificaContattoController: Gestisce la modifica di contatti esistenti.

  VisualizzaSingoloContattoController: Si occupa della visualizzazione dettagliata di un contatto.

  MainInterfaceController: Gestisce l'interfaccia principale e la gestione generale della rubrica.

In particolare, i controller JavaFX sono separati per singole funzionalità, evitando sovraccarichi di responsabilità.

Open-Closed Principle (OCP)

La classe Rubrica è aperta all'estensione, ma chiusa per modifiche. Se in futuro volessimo aggiungere nuovi formati di esportazione, potremmo farlo aggiungendo nuovi metodi senza alterare quelli esistenti.

I controller rispettano l'OCP, poiché le loro funzionalità possono essere estese (es. aggiunta di nuove validazioni o nuove viste) senza modificare il codice già esistente.

Liskov Substitution Principle (LSP)

La classe Contatto implementa l'interfaccia "Comparable", e può essere utilizzata ovunque sia richiesto un oggetto confrontabile. Questo segue il principio di sostituzione di Liskov.

Interface Segregation Principle (ISP)

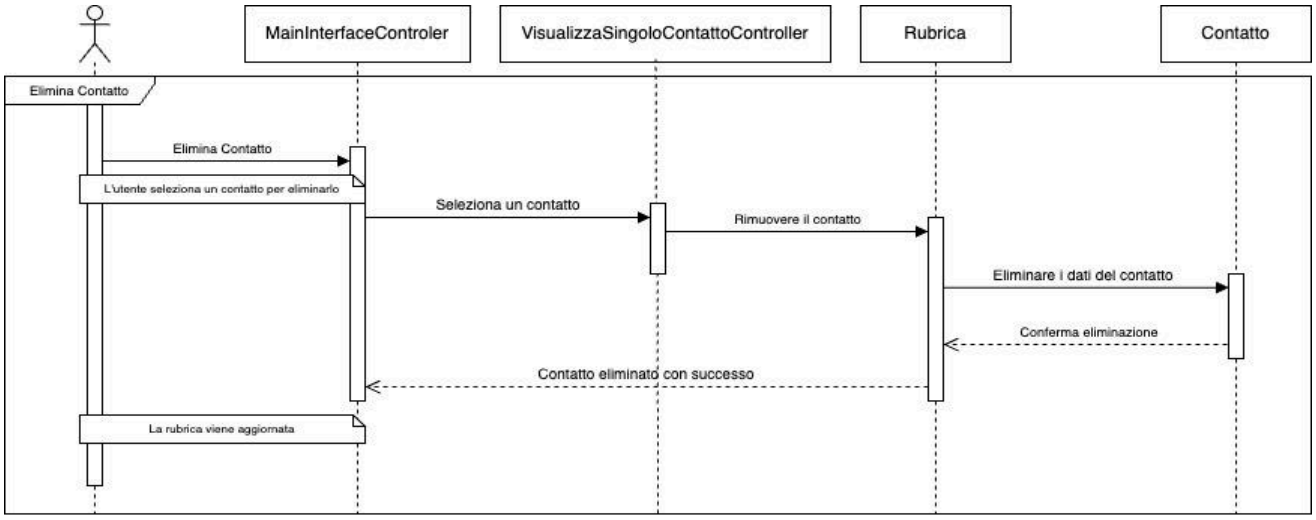
Il progetto non presenta interfacce ampie o non necessarie. Ogni classe si occupa solo delle interazioni essenziali. I controller non sono sovraccaricati con funzionalità inutili, rispettando il principio ISP.

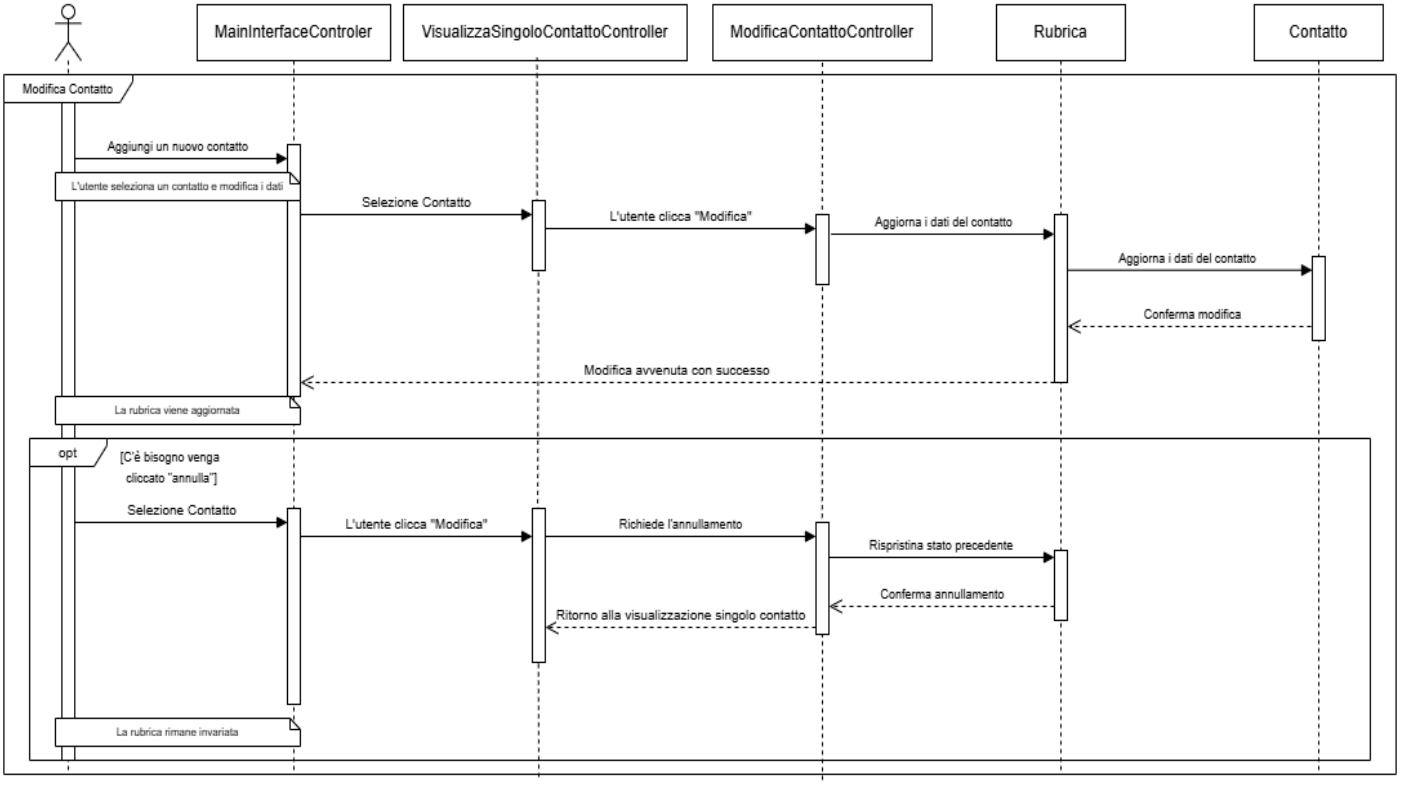
Dependency Inversion Principle (DIP)

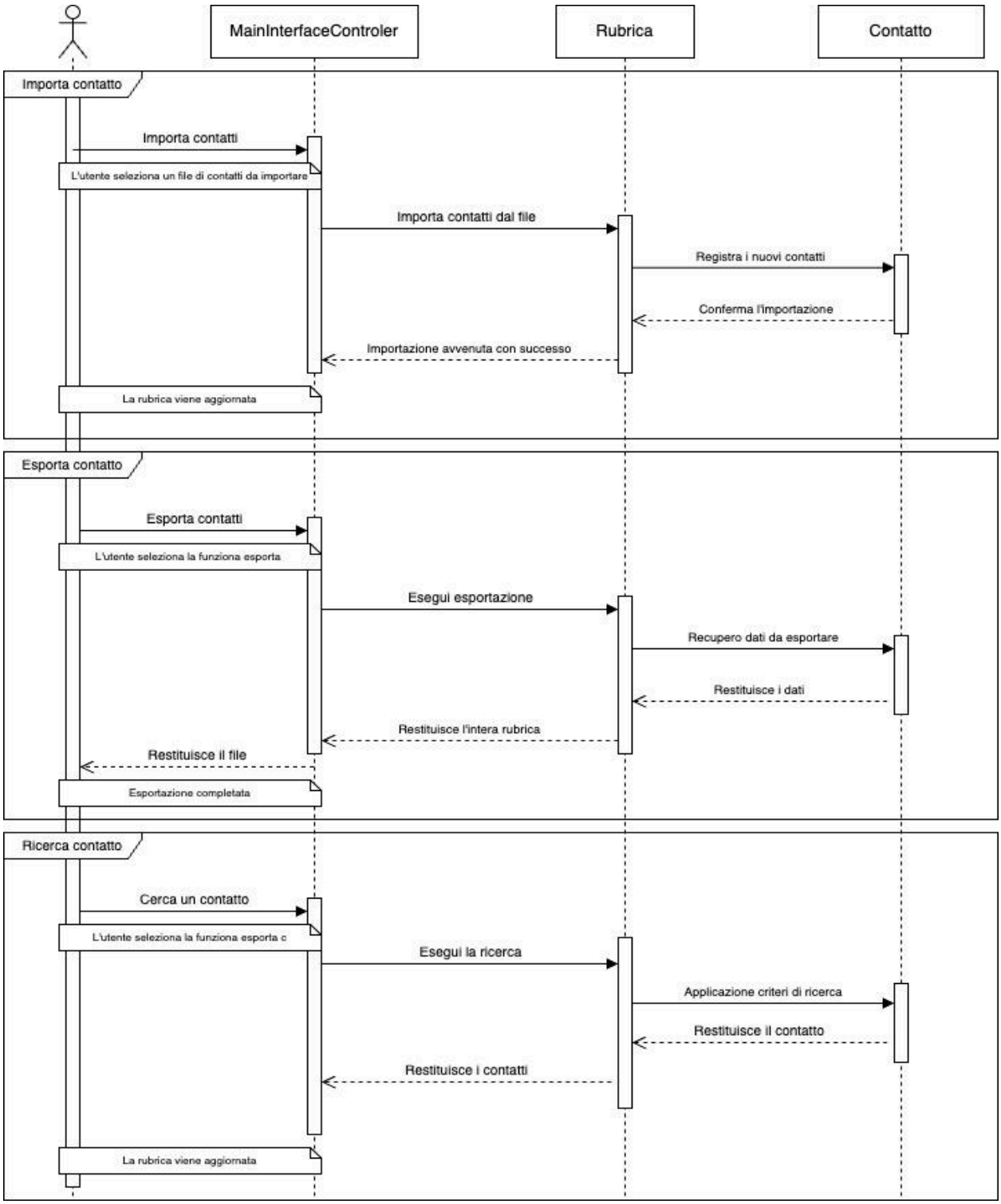
La classe App gestisce la rubrica condivisa tramite una variabile statica. Questo introduce una dipendenza forte tra i controller e la classe "App", violando leggermente il DIP.

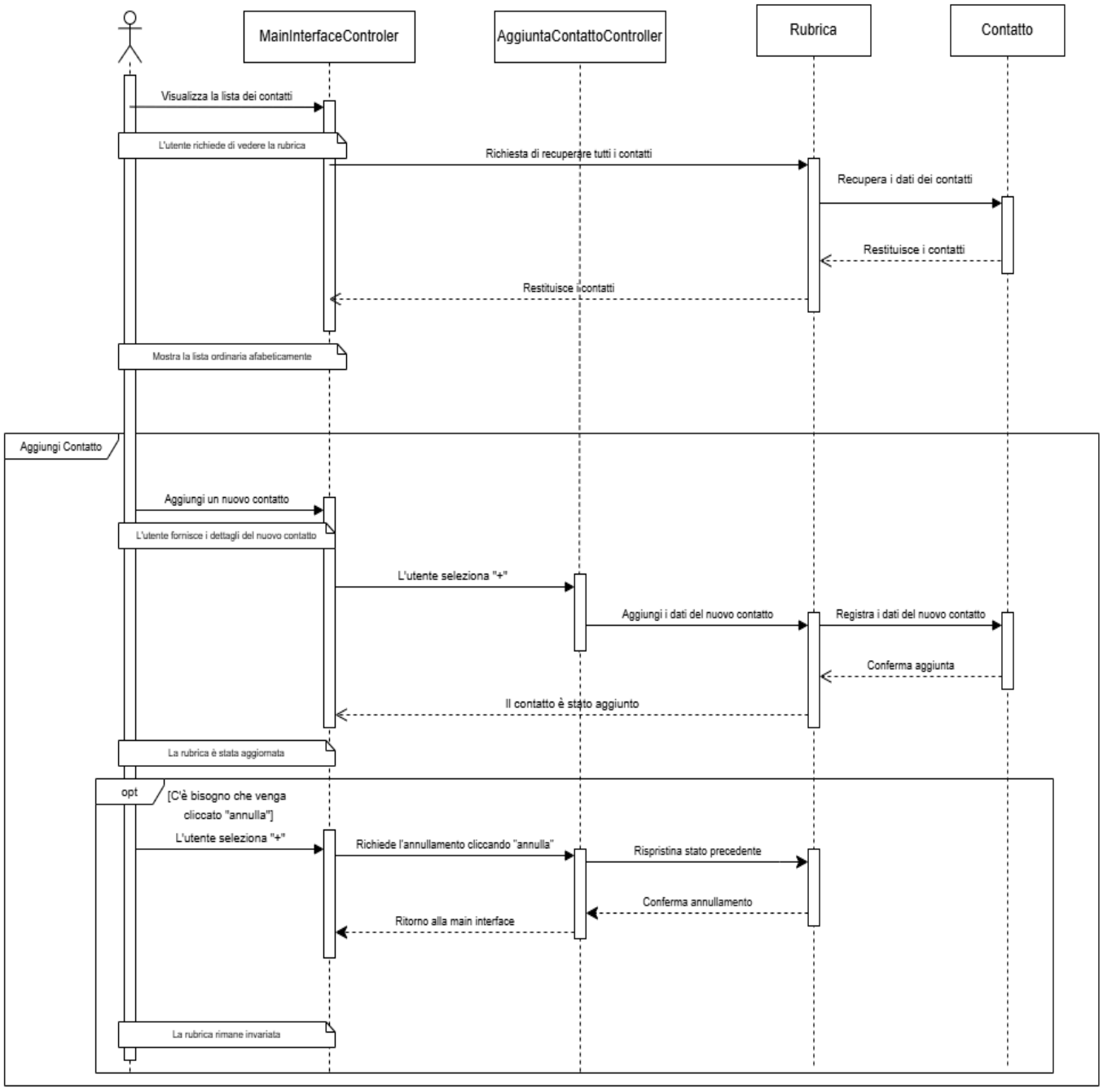
.

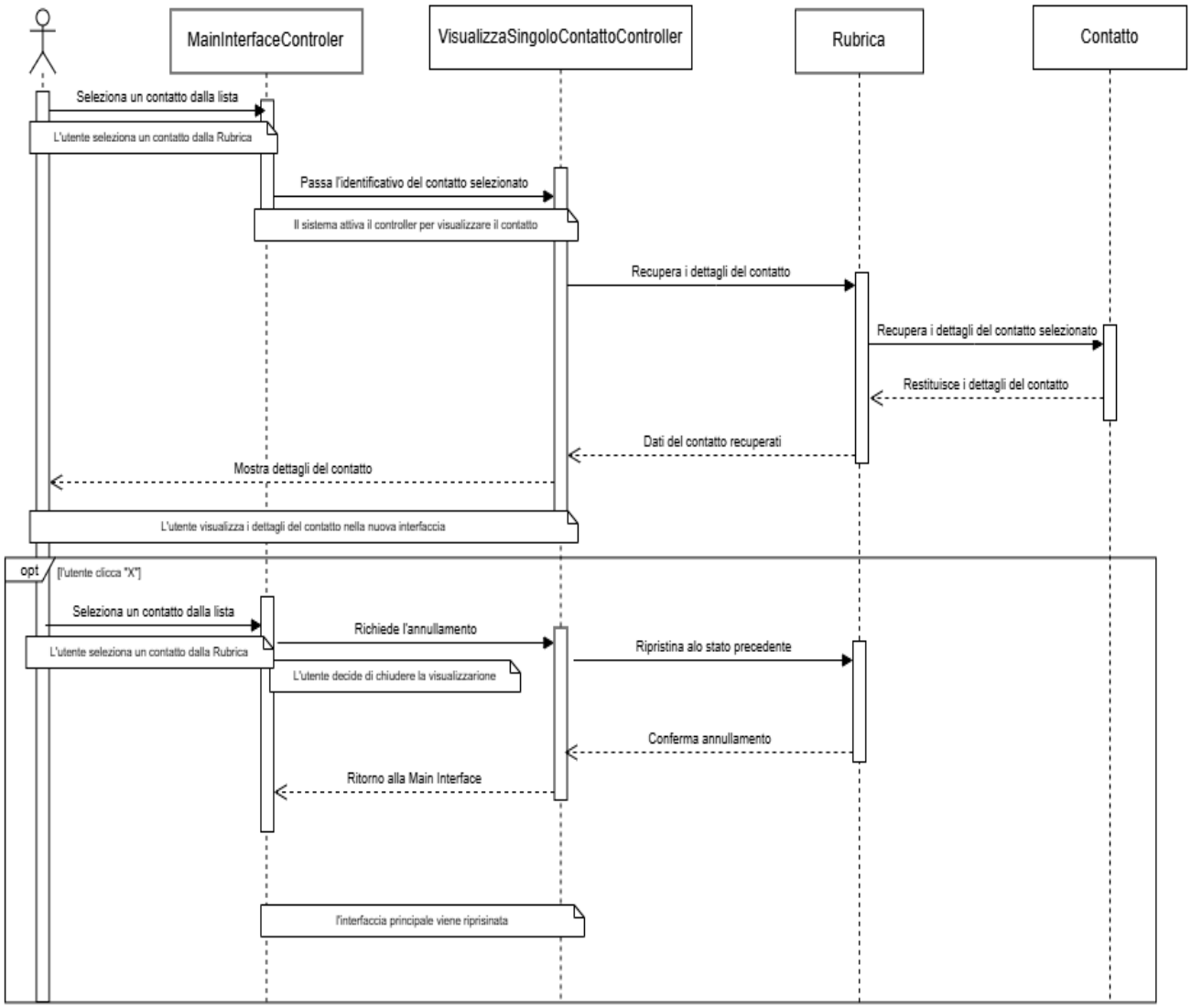
2. Diagrammi di Sequenza






MainInterfaceControler
Rubrica
Contatto
Importa contatto
Importa contatti
L'utente seleziona un file di contatti da importare
Importa contatti dal file
Registra i nuovi contatti
Conferma l'importazione
Importazione avvenuta con successo
La rubrica viene aggiornata
Esporta contatto
Esporta contatti
L'utente seleziona la funziona esporta
Esegui esportazione
Recupero dati da esportare
Restituisce i dati
Restituisce l'intera rubrica
Restituisce il file
Esportazione completata
Ricerca contatto
Cerca un contatto
L'utente seleziona la funziona esporta c
Esegui la ricerca
Applicazione criteri di ricerca
Restituisce il contatto
Restituisce i contatti
La rubrica viene aggiornata


MainInterfaceControler
AggiuntaContattoController
Rubrica
Contatto
Visualizza la lista dei contatti
L'utente richiede di vedere la rubrica
Richiesta di recuperare tutti i contatti
Recupera i dati dei contatti
Restituisce i contatti
Restituisce i contatti
Mostra la lista ordinaria afabeticamente
Aggiungi Contatto
Aggiungi un nuovo contatto
L'utente fornisce i dettagli del nuovo contatto
L'utente seleziona "+"
Aggiungi i dati del nuovo contatto
Registra i dati del nuovo contatto
Conferma aggiunta
Il contatto è stato aggiunto
La rubrica è stata aggiornata
opt
[C'è bisogno che venga cliccato "annulla"]
L'utente seleziona "+"
Richiede l'annullamento cliccando "annulla"
Rispristina stato precedente
Conferma annullamento
Ritorno alla main interface
La rubrica rimane invariata


MainInterfaceControler
VisualizzaSingoloContattoController
Rubrica
Contatto
Seleziona un contatto dalla lista
L'utente seleziona un contatto dalla Rubrica
Passa l'identificativo del contatto selezionato
Il sistema attiva il controller per visualizzare il contatto
Recupera i dettagli del contatto
Recupera i dettagli del contatto selezionato
Restituisce i dettagli del contatto
Dati del contatto recuperati
Mostra dettagli del contatto
L'utente visualizza i dettagli del contatto nella nuova interfaccia
opt
[l'utente clicca "X"]
Seleziona un contatto dalla lista
L'utente seleziona un contatto dalla Rubrica
Richiede l'annullamento
L'utente decide di chiudere la visualizzarione
Ripristina alo stato precedente
Conferma annullamento
Ritorno alla Main Interface
l'interfaccia principale viene riprisinata

2.1 Commenti Diagrammi Di Sequenza

1.Eliminazione Contatto

Single Responsibility Principle (SRP):

- MainInterfaceController: Gestisce la logica dell'interfaccia principale.
- VisualizzaSingoloContattoController: Si occupa esclusivamente della visualizzazione e selezione del contatto.
- Rubrica: Gestisce la logica di rimozione dei contatti, mantenendo separata la gestione dei dati.

Dependency Inversion Principle (DIP):

I controller invocano la classe Rubrica, ma la logica interna dei contatti è incapsulata nella classe Rubrica stessa. Questo rispetta il principio DIP.

DRY:

La conferma dell'eliminazione è centralizzata nella classe Rubrica, evitando duplicazioni di logica nei controller.

2.Importazione, Esportazione e Ricerca Contatti

Single Responsibility Principle (SRP):

MainInterfaceController: Gestisce le azioni utente e l'interazione con l'interfaccia grafica.

Rubrica: Contiene i metodi per importare, esportare e cercare contatti, separando la gestione dei dati dalla GUI.

Open-Closed Principle (OCP):

La Rubrica accetta solo formati .CSV per l'importazione ed esportazione dei contatti.

DRY:

La logica di accesso ai contatti è centralizzata in Rubrica, evitando duplicazioni nei controller.

YAGNI:

Le funzionalità sono implementate in modo semplice, senza introdurre logiche non necessarie.

3. Modifica Contatto

Single Responsibility Principle (SRP):

- ModificaContattoController: Gestisce esclusivamente la logica di modifica dei contatti.
- Rubrica: Controlla l'aggiornamento dei dati del contatto.

Liskov Substitution Principle (LSP):

I controller lavorano in modo coerente con Contatto e Rubrica, rispettando i contratti delle classi.

Open-Closed Principle (OCP):

È possibile aggiungere nuove funzionalità di modifica estendendo Rubrica senza alterare il codice attuale.

YAGNI:

La funzione di annullamento è implementata solo se necessaria, evitando logica superflua.

4. Visualizzazione Dettagli Contatto

Dependency Inversion Principle (DIP):

I controller dipendono da Rubrica, ma non conoscono i dettagli interni della gestione dei dati.

DRY:

La logica di recupero dei dettagli è centralizzata in Rubrica, evitando duplicazioni nei controller.

5.Aggiunta Contatto

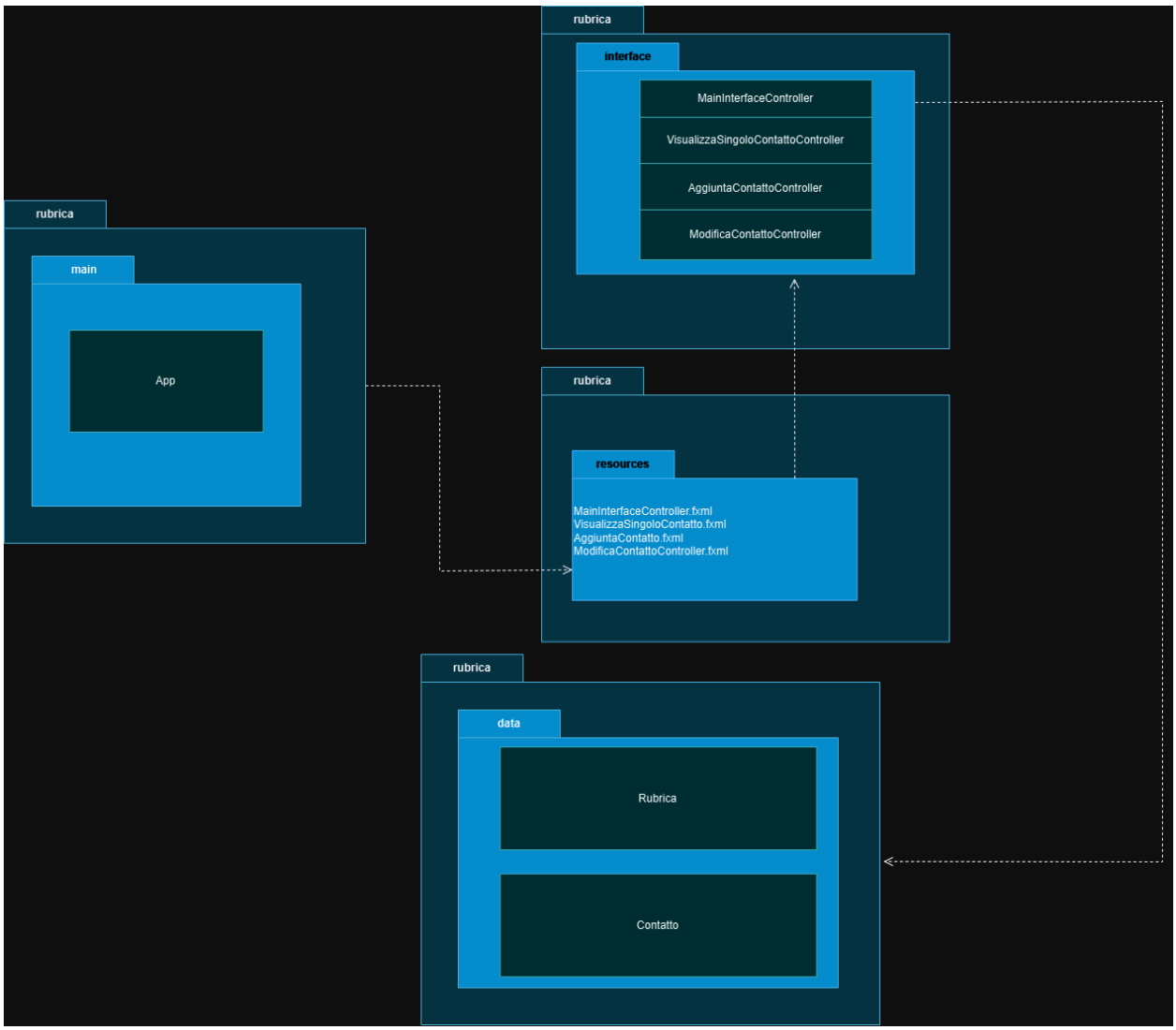
Open-Closed Principle (OCP):

La classe Rubrica può essere estesa per supportare nuovi tipi di validazioni o meccanismi di registrazione senza modificare i metodi esistenti.

YAGNI:

L'operazione di annullamento è implementata in modo minimale per garantire il corretto ripristino dello stato.

3. Diagramma dei Package



3.1 Commento al Diagramma Dei Package

Struttura dei Package

- **rubrica.main**: Contiene la classe App, che rappresenta il punto di ingresso dell'applicazione. Gestisce l'avvio e la condivisione della rubrica.
- **rubrica.interface**: Include i controller principali per la gestione della GUI, suddivisi per responsabilità:
    - MainInterfaceController: Interfaccia principale della rubrica.
    - VisualizzaSingoloContattoController: Gestisce la visualizzazione di un contatto specifico.
    - AggiuntaContattoController: Responsabile dell'aggiunta di nuovi contatti.
    - ModificaContattoController: Gestisce la modifica di un contatto esistente.
- **rubrica.data**: Contiene le classi modello (Rubrica e Contatto) che rappresentano i dati principali dell'applicazione.
- **rubrica.resources**: Raccoglie i file FXML utilizzati dai controller per il rendering delle viste.

Applicazione dei Principi SOLID

Single Responsibility Principle (SRP)

Ogni package ha un determinato compito:

- **main**: Avvio e configurazione dell'applicazione.
- **interface**: Gestione delle interfacce utente tramite controller. Ogni classe controller ha una singola responsabilità legata alla vista specifica.
- **data**: Contiene i modelli dati (Contatto e Rubrica), mantenendo separata la logica di gestione dei dati dalla logica della GUI.
- **resources**: Archivia i file di configurazione delle viste (FXML), mantenendo il codice separato dalla logica dell'interfaccia.

Questa separazione rispetta il principio SRP, riducendo le responsabilità all'interno di ciascun package.

Open-Closed Principle (OCP)

Estensione: La struttura a pacchetti permette di aggiungere nuove funzionalità senza modificare quelle esistenti. Ad esempio:

- Si possono aggiungere nuovi controller nel package interface per gestire ulteriori viste.
- Si possono estendere la classe Rubrica nel package data per supportare nuove funzionalità (es. filtri avanzati, statistiche).

Modifica: Nessun package dipende direttamente dalle implementazioni interne degli altri. La separazione tra GUI, logica di business e dati facilita le modifiche.

Liskov Substitution Principle (LSP)

Le classi Rubrica e Contatto nel package data rispettano il principio LSP, in quanto possono essere utilizzate senza violare le aspettative delle dipendenze nei controller.

I controller nel package interface utilizzano correttamente i dati forniti da Rubrica e Contatto senza dipendere dai dettagli implementativi.

Interface Segregation Principle (ISP)

Non sono presenti interfacce molto generiche o inutilmente ampie. Ogni classe controller (MainInterfaceController, AggiuntaContattoController, ecc.) ha metodi specifici per la gestione della propria vista.

Dependency Inversion Principle (DIP)

La classe App nel package main gestisce la rubrica condivisa, ma utilizza una dipendenza statica per condividerla con i controller. Questo potrebbe violare leggermente il DIP.

DRY (Don’t Repeat Yourself)

I file FXML nel package resources centralizzano la definizione delle viste, evitando duplicazioni nei controller.

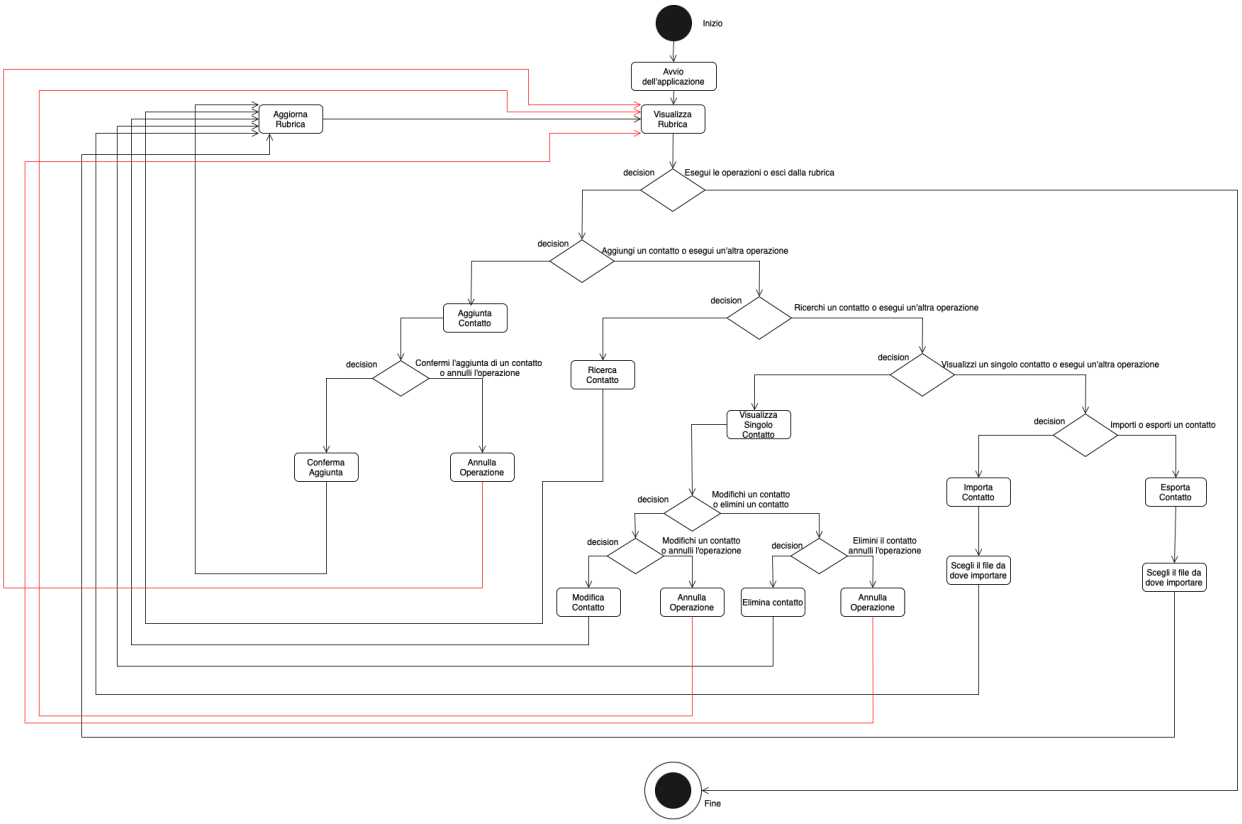
La gestione della rubrica è centralizzata nella classe Rubrica, evitando logiche duplicate nei controller.

La separazione dei controller per funzionalità specifiche garantisce che il codice non sia ripetuto.

YAGNI (You Aren’t Going to Need It)

Ogni componente del progetto ha un utilizzo chiaro e immediato. Non ci sono funzionalità superflue o non necessarie. I file FXML sono limitati a quelli realmente utilizzati dai controller

4. Diagramma delle attività



4.1 Commento al Diagramma delle Attività

Applicazione dei Principi di Buona Progettazione

Single Responsibility Principle (SRP)

Ogni attività rappresentata nel diagramma corrisponde chiaramente a un'operazione specifica:

- **Aggiunta Contatto**: Responsabilità chiara dell'aggiunta di un contatto.
- **Visualizzazione e Modifica Contatto**: Attività separate per visualizzare e modificare un contatto, rispettando il principio SRP.
- **Eliminazione Contatto**: Separata e con un flusso di conferma chiaro.

- **Importazione/Esportazione**: Sono rappresentate come operazioni indipendenti e isolate, con la responsabilità ben definita.

Questa chiarezza garantisce che ciascuna attività sia autonoma e focalizzata su un solo compito.

Open-Closed Principle (OCP)

Il diagramma mostra una struttura **aperta all'estensione** e **chiusa alla modifica**:

- Se si volessero aggiungere nuove operazioni (es. **filtraggio avanzato dei contatti**), si potrebbero inserire come nuovi rami senza modificare le attività esistenti.
- Il design attuale consente facilmente di estendere funzionalità come la gestione dei file di esportazione/importazione.

Liskov Substitution Principle (LSP)

Ogni attività (es. modifica, aggiunta, eliminazione) utilizza operazioni che interagiscono correttamente con gli oggetti contatto e rubrica,

- La sostituzione dei dati o il cambio delle operazioni non violerebbe il principio LSP, poiché le attività rispettano i comportamenti previsti per contatto e rubrica.

Interface Segregation Principle (ISP)

Ogni attività è ben separata e non ci sono operazioni presenti inutilmente o ripetitive.

- Ad esempio, **Aggiunta Contatto** e **Eliminazione Contatto** non interferiscono con la gestione della ricerca o importazione.

Dependency Inversion Principle (DIP)

L'interazione tra attività principali e la gestione della rubrica è gestita nel seguente modo:

- Le attività come visualizza rubrica e modifica contatto interagiscono indirettamente con Rubrica e Contatto.
- La separazione ci dice che le dipendenze sono gestite in modo appropriato tra il livello GUI e il livello dati.

Principio DRY (Don’t Repeat Yourself)

Non sono presenti duplicazioni di operazioni o attività nel diagramma:

- Ad esempio, i rami di **Annulla Operazione** sono chiaramente riutilizzati in più punti (Aggiunta Contatto, Modifica Contatto, Eliminazione Contatto).
- Questo garantisce che il flusso sia pulito e riduca inutili ripetizioni.

Principio YAGNI (You Aren’t Going to Need It)

Ogni attività rappresentata nel diagramma è essenziale per il funzionamento dell'applicazione.

- Non sono presenti operazioni ridondanti o funzionalità non richieste.
- Il design è semplice, con un focus esclusivo sulle operazioni di base della rubrica.

5. Struttura del progetto e scelte progettuali

5.1 Contatto

La classe rappresenta un singolo contatto con i suoi attributi (nome, cognome, numeri di telefono, email, ecc.). La coesione è **funzionale**, il livello più alto, poiché tutte le funzionalità della classe lavorano insieme per rappresentare e manipolare le informazioni di un contatto. Gli attributi, i metodi getter e setter, e il metodo compareTo servono un unico obiettivo: fornire una rappresentazione consistente di un contatto e consentirne il confronto e la gestione. Non ci sono metodi che si discostano dallo scopo principale della classe, mantenendo alta la coesione. L'accoppiamento della classe Contatto è **per dati**, un livello basso e desiderabile. Essa interagisce con altre classi (es. Rubrica) passando i propri oggetti o ricevendo dati tramite costruttori o metodi. La dipendenza da LocalDate per la gestione della data di compleanno è

appropriata e segue il principio di usare oggetti specifici per rappresentare dati complessi.
Non ci sono dipendenze a livello di implementazione interna con altre classi, e il metodo statico per validare email e numeri di telefono migliora la modularità evitando duplicazione di logica.

5.2 Rubrica

La classe Rubrica ha una **coesione funzionale**, il livello più alto, poiché tutte le sue funzionalità sono strettamente correlate alla gestione di una collezione di contatti. Ogni metodo serve un obiettivo preciso e pertinente, come aggiungere, cercare, ordinare, esportare e importare i contatti. L'implementazione riflette un'ottima progettazione orientata agli oggetti, in cui la classe Rubrica incapsula la gestione della lista di contatti e fornisce metodi che mantengono il controllo centralizzato sulla lista. L'accoppiamento è prevalentemente **per dati**, poiché Rubrica lavora con oggetti Contatto, sia per aggiungerli che per modificarli o eliminarli. Questa relazione è naturale e necessaria, data la dipendenza della rubrica dalla struttura dei contatti. Le dipendenze con i metodi di esportazione e importazione dei file sono a basso accoppiamento (usano java.io e formati standard come CSV), rendendo il codice facilmente estendibile.

5.3 AggiuntaContattoController

La classe AggiuntaContattoController ha una **coesione funzionale**, poiché tutte le sue funzionalità sono incentrate sull'aggiunta di un nuovo contatto alla rubrica. Gestisce l'interfaccia grafica e interagisce con i campi di input per raccogliere dati necessari alla creazione di un nuovo oggetto Contatto. I metodi come aggiungiContatto, annulla, e pulisciCampi sono strettamente legati all'obiettivo principale della classe, senza deviazioni verso altre responsabilità.
L'inizializzazione tramite initialize e la gestione della rubrica condivisa (Rubrica) garantiscono una coesione elevata senza introduzioni esterne.. L'accoppiamento della classe è **per dati**, poiché interagisce con la classe Rubrica per aggiungere contatti e con la classe Contatto per crearli.

Parliamo anche di **accoppiamento per controllo** nei metodi che gestiscono la navigazione tra le schermate (es. caricamento di nuove scene tramite FXMLLoader). Questo introduce una leggera dipendenza dalle specifiche dell'interfaccia grafica.

5.4 MainInterfaceController

La classe MainInterfaceController ha una **coesione funzionale**, poiché si occupa esclusivamente della gestione dell'interfaccia principale della rubrica. Ogni metodo è legato all'interazione con la GUI, come la visualizzazione dei contatti, la ricerca, l'importazione, l'esportazione e la navigazione verso altre schermate. Il metodo initialize gestisce l'inizializzazione della tabella dei contatti e configura gli ascoltatori per la ricerca in tempo reale, un'implementazione strettamente legata al suo scopo. L'accoppiamento è prevalentemente **per dati**, poiché la classe interagisce con Rubrica per ottenere, ordinare e filtrare i contatti.

Tuttavia, esistono segnali di **accoppiamento per controllo** nei metodi che gestiscono la navigazione tra schermate. Ad esempio, il metodo displayAggiungiContatto utilizza direttamente FXMLLoader per caricare nuove scene e passare la rubrica condivisa al controller successivo. Questo introduce una dipendenza diretta dalla struttura dell'interfaccia grafica.

5.5 ModificaContattoController

La classe ModificaContattoController ha una **coesione funzionale**, in quanto tutte le sue funzionalità si concentrano sulla modifica di un contatto selezionato. Questa responsabilità include l'aggiornamento dei dati, la validazione degli input e il reindirizzamento dell'utente dopo la modifica o l'annullamento.I metodi principali, come modificaContatto, annulla, e setContatto, lavorano insieme per completare il flusso di modifica. La coesione è ulteriormente evidenziata dal fatto che la classe utilizza correttamente i dati di Contatto per popolare i campi e aggiornarli, senza introdurre logica estranea o funzionalità non correlate. L'accoppiamento è principalmente **per dati**, poiché la classe interagisce direttamente con l'oggetto Contatto per leggere e scrivere i dati modificati. La relazione è necessaria e ben gestita. Possiamo evidenziare segni di **accoppiamento per controllo** nel metodo annulla e nel metodo modificaContatto, dove viene utilizzato FXMLLoader per cambiare schermata. Questo introduce una dipendenza diretta dalla

struttura dell'interfaccia grafica, rendendo la classe meno indipendente. La classe dipende anche da Rubrica attraverso il metodo modificaContatto (via App.getRubricaCondivisa), che è un esempio di accoppiamento per dati.

5.6 VisualizzaSingoloContattoController

La classe VisualizzaSingoloContattoController ha una **coesione funzionale**, poiché è progettata per un unico scopo: visualizzare i dettagli di un contatto selezionato. Ogni metodo contribuisce a questo obiettivo. Il metodo setContatto popola i campi grafici con i dati del contatto selezionato, mentre aggiornaDatiContattogarantisce che la visualizzazione sia sincronizzata con eventuali modifiche apportate al contatto. Anche i metodi che gestiscono la navigazione (displayMainInterface e displayModificaContatto) si mantengono focalizzati sull'obiettivo principale della classe, che è facilitare la gestione dei dettagli del contatto, rendendo elevato il livello di coesione. L'accoppiamento è prevalentemente **per dati**, in quanto la classe interagisce con un oggetto Contatto per leggere i dettagli del contatto da visualizzare. La classe ha anche un **accoppiamento per controllo** nei metodi che gestiscono la navigazione tra le schermate (es. displayModificaContatto), dove si utilizza FXMLLoader per caricare altre scene. L'uso di App.getRubricaCondivisa() per accedere alla rubrica condivisa introduce un accoppiamento diretto con la classe App.

5.7 App

La classe App ha una **coesione funzionale**, in quanto il suo scopo è esclusivamente quello di avviare l'applicazione JavaFX e gestire la scena principale. Tutti i metodi e le variabili servono questo obiettivo. Il metodo start inizializza la rubrica condivisa e configura la scena iniziale dell'interfaccia utente, mantenendo una chiara separazione delle responsabilità. La coesione è ulteriormente evidenziata dalla presenza di un singolo punto d'accesso per ottenere la rubrica condivisa (getRubricaCondivisa), che semplifica la gestione centralizzata dei dati.
L'accoppiamento della classe App è prevalentemente **per dati**, poiché fornisce l'istanza condivisa della rubrica (Rubrica) alle altre parti dell'applicazione. Questa relazione è necessaria per mantenere una coerenza nello stato dei dati tra le diverse viste. Con l’utilizzo di un metodo

statico (getRubricaCondivisa) si introduce una forma di accoppiamento globale. L'uso diretto di FXMLLoader per caricare l'interfaccia introduce un **accoppiamento per controllo**, poiché la classe si basa su file FXML specifici e controller per configurare le scene. Non ci sono dipendenze dirette con altre funzionalità non pertinenti, mantenendo il livello di accoppiamento basso e appropriato per il ruolo della classe.