UNIVERSITÀ DEGLI STUDI
DI SALERNO

.DIEM

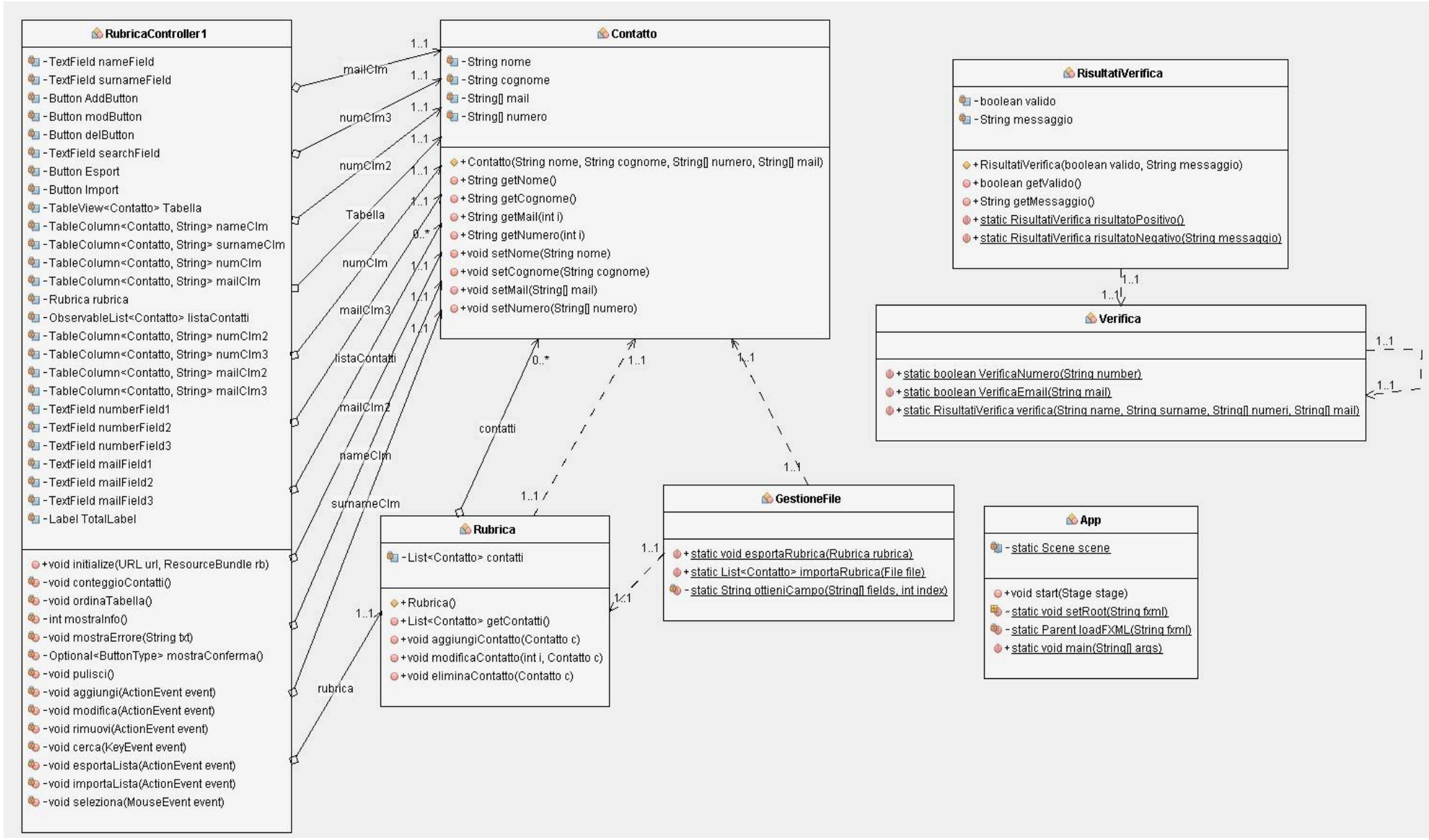
Team:

Fabio Genito (User su Github)

Michele Falcone (utente su Github)

Giacomo Cuccaro (gcucc su Github)

Diagramma delle Classi



Coesione e Accoppiamento

La classe `Contatto`

La classe rappresenta una singola entità, il contatto, e si occupa esclusivamente di mantenere le informazioni attribuitegli: Un Nome, Un Cognome (Almeno uno dei due è necessario), da 0 a 3 Numeri di telefono e da 0 a 3 E-mail.

Coesione

La classe presenta una coesione di tipo **funzionale**: la classe è focalizzata su un singolo compito, ovvero la rappresentazione di un’entità concreta. Non ci sono metodi estranei o non correlati al concetto di "contatto", i metodi forniti sono strettamente legati al ruolo della classe: getter e setter per manipolare i dati.

Accoppiamento

La classe non dipende direttamente da altre classi o pacchetti, il che implica un basso o nullo livello di accoppiamento; essendo però la classe base del sistema, Contatto ha un minimo e inevitabile accoppiamento naturale (**per dati**) e quindi non può essere considerata completamente disaccoppiata. Va considerato che gli attributi “mail” e “num” sono rappresentati come array di stringhe, il che evita di introdurre dipendenze verso strutture dati personalizzate o complesse.

La classe `Rubrica`

La classe gestisce una collezione di contatti e fornisce metodi per aggiungere, rimuovere, modificare e cercare contatti.

Coesione

La classe un presenta una coesione di tipo **comunicazionale**: è progettata per gestire un insieme di oggetti della classe `Contatto` e fornisce le relative operazioni logiche. I metodi proposti (creazione, modifica, eliminazione, ricerca) sono pertinenti al concetto di “gestione contatti" e operano su una struttura dati comune, cioè il Contatto.

Accoppiamento

La classe presenta accoppiamento **per timbro** poiché la classe dipende da Contatto (come tipo). Ciò significa che Rubrica è progettata specificamente per funzionare con Contatto, riducendo la generalità; **per dati**, ovvero un accoppiamento debole e tollerabile, riguardante solo lo scambio di dati concreti come parametri o risultati di metodi.

La classe `RubricaController`

La classe gestisce le operazioni di comunicazione tra l'interfaccia utente (UI) e le altre classi, permettendo l'associazione di interazioni grafiche con metodi che operano su dati effettivi.

Coesione

La classe presenta una coesione **procedurale** poiché tutti i metodi servono a gestire eventi della View, modificare nel package **`progettoingsoftware.mode`**l la classe **`Rubrica`** e aggiornare la View stessa. La coesione è anche **comunicazionale** poiché, nonostante i metodi condividano le stesse strutture dati (es. listaContatti, Tabella, rubrica), ci sono segmenti che necessariamente duplicano la logica.

Accoppiamento

In termini di accoppiamento, la classe accede direttamente al package **`com.mycompany.testfunionzionate.model`** (Rubrica e Contatto) per aggiornare i dati. Questo accoppiamento è **per dati** (quindi accettabile nel pattern MVC) perchè il Controller deve coordinare le classi di **`com.mycompany.testfunzionante.model`** e **`com.mycompany.testfunzionante.view`**. Anche considerando la presenza del file **`View.fxml`**, l'accoppiamento rimane **per dati**, poiché il Controller accede direttamente agli elementi della UI (TextField, TableView, TableColumn, Button). Questo crea una forte dipendenza del Controller rispetto alla View e naturalmente il problema di rendere il Controller difficile da testare senza la UI. Inoltre se la View cambiasse (es. eliminazione di un campo o modifica del layout), il Controller dovrà essere aggiornato di conseguenza, aumentando il rischio di errori.

La classe `GestioneFile`

La classe gestisce i metodi da invocare in caso di operazioni che coinvolgono lo scambio di informazioni tra la Rubrica e File esterni. La classe interagisce direttamente con l'interfaccia grafica, per cui non può essere testata automaticamente mediante JUnit. Il testing deve essere fatto necessariamente in maniera manuale dall'utente, per ottenere un risultato soddisfacente (con JUnit il 66,67% dei test risultano superati). Questo accade perché si sta cercando di eseguire un'operazione che deve essere effettuata nel thread della GUI di JavaFX.

Coesione

La classe ha una coesione **funzionale**, tutti i metodi servono ad uno scopo specifico e unitario, ovvero gestire file per importare o esportare dati relativi alla rubrica; grazie a questa focalizzazione sulle operazioni di gestione dei file, vengono evitate logiche non pertinenti.

Accoppiamento

Per quanto riguarda l'accoppiamento, la classe è accoppiata con il pacchetto standard Java per la gestione dei file (java.io), che è un accoppiamento **per dati** naturale e inevitabile per questa responsabilità (questo accoppiamento non è un problema, purché l'accesso a queste classi sia ben incapsulato nei metodi della classe `GestioneFile`). Il metodo `importaRubrica` restituisce una lista di oggetti Contatto (`ArrayList<Contatto>`). Questo accoppiamento è **per dati**, ed è accettabile perché la gestione dei file riguarda il trasferimento di dati tra il package `com.mycompany.testfunzionante.model` e fonti esterne. È importante che questa classe non conosca dettagli specifici di come i dati di Contatto sono organizzati, ma si limiti a trasferire dati serializzati o deserializzati.

La classe `Verifica`

La classe **`Verifica`** fornisce metodi statici per validare i dati di input. Include controlli specifici per:

- Validità di un numero (solo cifre).
- Validità di un indirizzo e-mail.
- Validità di un contatto completo, restituendo un oggetto **`RisulatiVerifica`** che indica l'esito della validazione.

La classe **`Verifica`**, così come **`RisulatiVerifica`** è stata introdotta (successivamente) per migliorare la separazione delle responsabilità all'interno del sistema. In particolare, il suo scopo è gestire i controlli sugli input forniti dall'utente, evitando di demandare tali operazioni alla sola classe **`RubricaController`**. Questo approccio promuove un design più modulare e testabile, separando la logica di validazione dagli aspetti di gestione dell'interfaccia e del modello. Tuttavia questa classe è difficile da testare a livello unitario in quanto dipende strettamente da **`RisultatiVerifica`** e dunque servirebbe un test di integrazione per assicurarsi che tutto funzioni come previsto; gli unit tests infatti non riescono a verificare il completo funzionamento della classe (tramite JUnit si è riusciti a raggiungere il 90% dei test superati). Inoltre data la natura della classe Verifica (Helper Class) risulterebbe eccessivo separare ulteriormente la logica delle operazioni per poter ottenere una percentuale maggiore di successo.

Coesione

La coesione è **funzionale**, in quanto tutti i metodi sono progettati per un obiettivo specifico e coerente: verificare la validità dei dati di input.Ogni metodo (ad esempio **`VerificaNumero, VerificaMail, verifica`**) contribuisce a questa responsabilità primaria.

Accoppiamento

La classe non dipende da alcun componente esterno complesso o specifico del progetto, ad eccezione di **`RisulatiVerifica`** , con cui ha un accoppiamento **per dati**. **`Verifica`** utilizza **`RisulatiVerifica`** solo per rappresentare il risultato delle verifiche, senza legarsi a dettagli di implementazione, i suoi metodi infatti accettano input come stringhe o array e restituiscono semplici valori booleani.

La classe `RisultatiVerifica`

La classe **`RisultatiVerifica`** rappresenta l'esito di un'operazione di validazione. È progettata come una semplice struttura dati per indicare se la validazione è riuscita (**`Valido`**) e per poi fornire un messaggio di errore in caso di esito negativo (**`Messaggio`**). Come per la classe Verifica si tratta di una Helper Class creata in un secondo momento per favorire un design più modulare e testabile, separando la logica di validazione dagli aspetti di gestione dell'interfaccia e del modello. A differenza della classe Verifica, però, essa può essere testata in quanto non dipende da altre classi.

Coesione

Anche in questo caso la coesione è **funzionale**, infatti tutti gli attributi e i metodi della classe sono orientati a rappresentare e gestire i risultati di una validazione. La presenza di metodi statici (**`risultatoPositivo`** e **`risultatoNegativo`**) rafforza la coesione, consentendo la creazione standardizzata di oggetti di questa classe.

Accoppiamento

In questo caso l'accoppiamento è inesistente; la classe non dipende da nessun altro componente del sistema, il che ne facilita il riutilizzo e la testabilità. La relazione è unidirezionale e rende l'accoppiamento **inesistente** dal punto di vista di **`RisultatiVerifica`** rispetto a **`Verifica`**.

La classe `App`

Si tratta della MainClass e dunque ha come scopo quello di avviare l'applicazione.

Coesione

La classe presenta una coesione di tipo **funzionale**. Ogni metodo contribuisce ad un obiettivo comune: avviare l'applicazione e configurare l'interfaccia grafica, infatti:

- main: Chiama launch(args), che avvia l'applicazione JavaFX.
- start: Configura la scena iniziale e carica il file FXML, che rappresenta il layout dell'interfaccia utente.

La classe non presenta metodi che si discostano da questa responsabilità principale.

Accoppiamento

Come per la classe "base" **`Contatto`**, la MainClass **`Applicazione`** presenta un accoppiamento debole **per dati** con i componenti che utilizza. Le dipendenze sono ridotte al minimo e gestite tramite JavaFX.

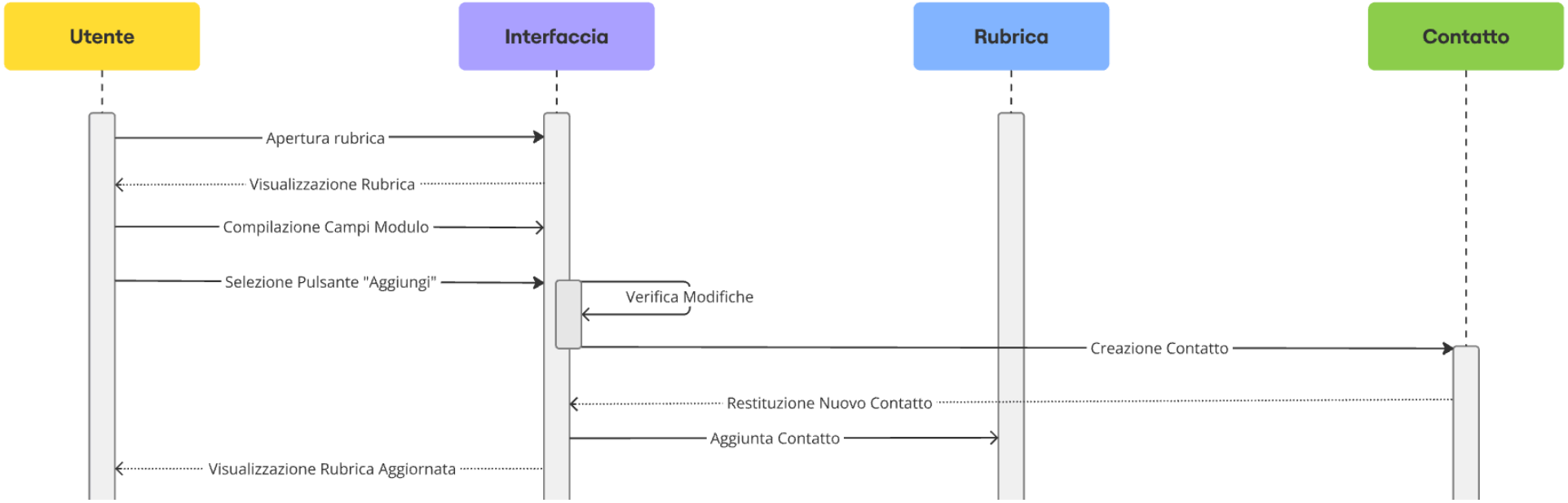
Analisi di buona progettazione

	Contatto	Rubrica	GestioneFile	RubricaController	Applicazione	Verifica	RisultatiVerifica
SRP	Gestisce un singolo contatto, con metodi di accesso e modifica per nome, cognome, mail e numeri.	Gestisce una lista di contatti, fornendo operazioni di aggiunta, modifica, eliminazione e ricerca.	Si occupa di importare ed esportare una rubrica su file esterni.	Collega la View (UI) con il modello (dati) gestendo eventi e aggiornamenti.	Si occupa solo dell'avvio dell'applicazione e della configurazione iniziale della scena JavaFX.	Focalizzata sulla verifica di input, separando le responsabilità dalla classe RubricaController.	Rappresenta il risultato delle verifiche di input, rendendo chiara la gestione degli esiti validi o invalidi.
OCP	Non è necessario per il suo scopo limitato.	Aperta alle modifiche e all'estensione.	Chiusa per estensioni.	Non è aperta: ogni modifica alla gestione degli eventi richiede cambiamenti nella classe stessa.	Aperta a modifiche, ma non particolarmente estensibile.	Aperta a estensioni (nuove regole di validazione), ma chiusa per modifiche.	Non applicabile, poiché la classe è limitata al suo ruolo.
DRY	Nessuna ripetizione, i metodi sono unici e ben definiti.	Alcune funzioni ripetono l'uso della struttura List, ma la ripetizione è accettabile.	Non presenta ripetizioni.	Ripete operazioni simili per l'interazione con campi di testo (es. aggiunta di contatti).	Non presenta ripetizioni: le operazioni di avvio sono uniche e concise.	Non presenta ripetizioni, ogni controllo è ben definito.	Non presenta ripetizioni.

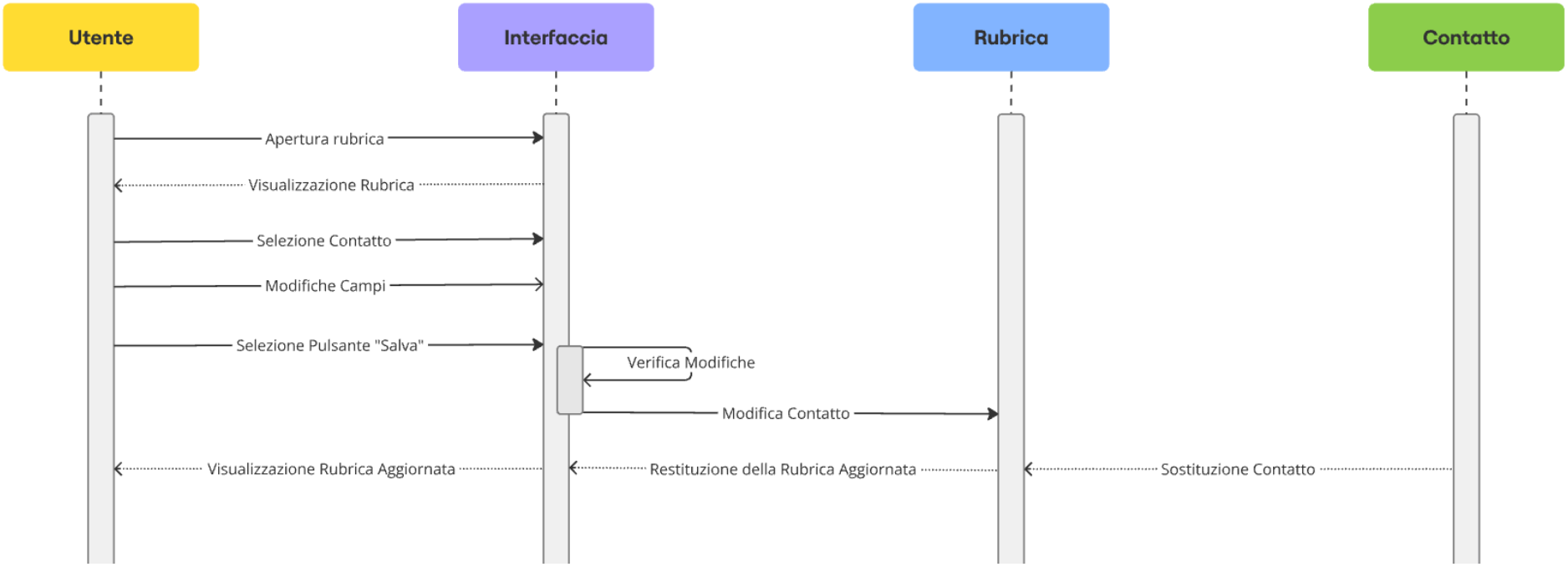
	Contatto	Rubrica	GestioneFile	RubricaController	Applicazione	Verifica	RisultatiVerifica
KISS	Semplice e diretta: implementa i getter e setter in modo chiaro.	Mantiene la semplicità.	Semplice nel design.	Mantiene la semplicità, ma alcune operazioni ripetute potrebbero essere centralizzate.	Design semplice e lineare.	Semplice e chiara: i metodi sono auto-esplicativi e modulari.	Mantiene la semplicità, con costruttori e metodi statici ben definiti.
Ereditarietà	Non utilizza ereditarietà.	Non utilizza ereditarietà.	Non utilizza ereditarietà.	Non utilizza ereditarietà.	Estende Application di JavaFX, quindi sfrutta correttamente l'ereditarietà per il ciclo di vita dell'applicazione	Non utilizza ereditarietà.	Non utilizza ereditarietà.

Diagrammi di Sequenza

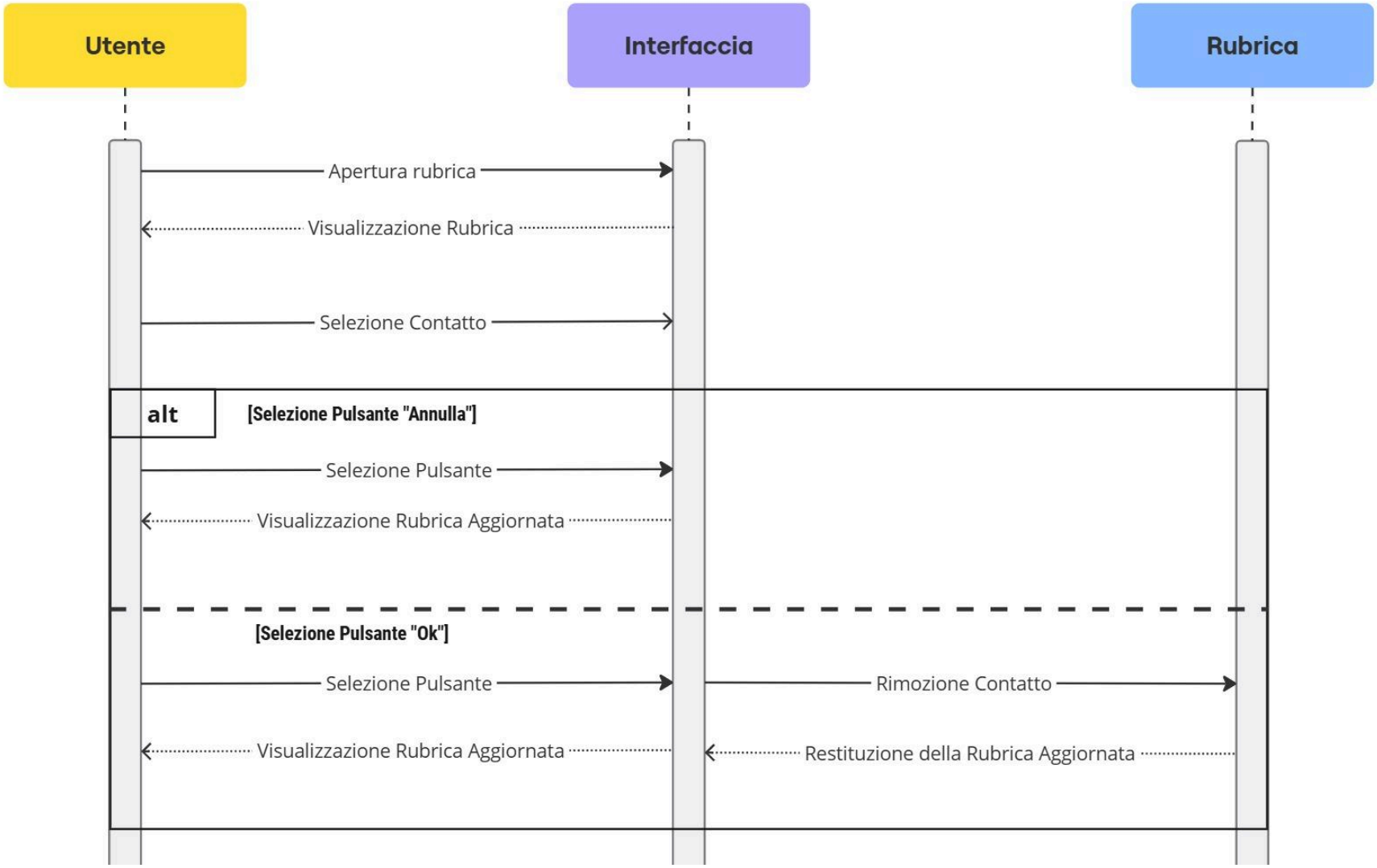
Creazione Contatto



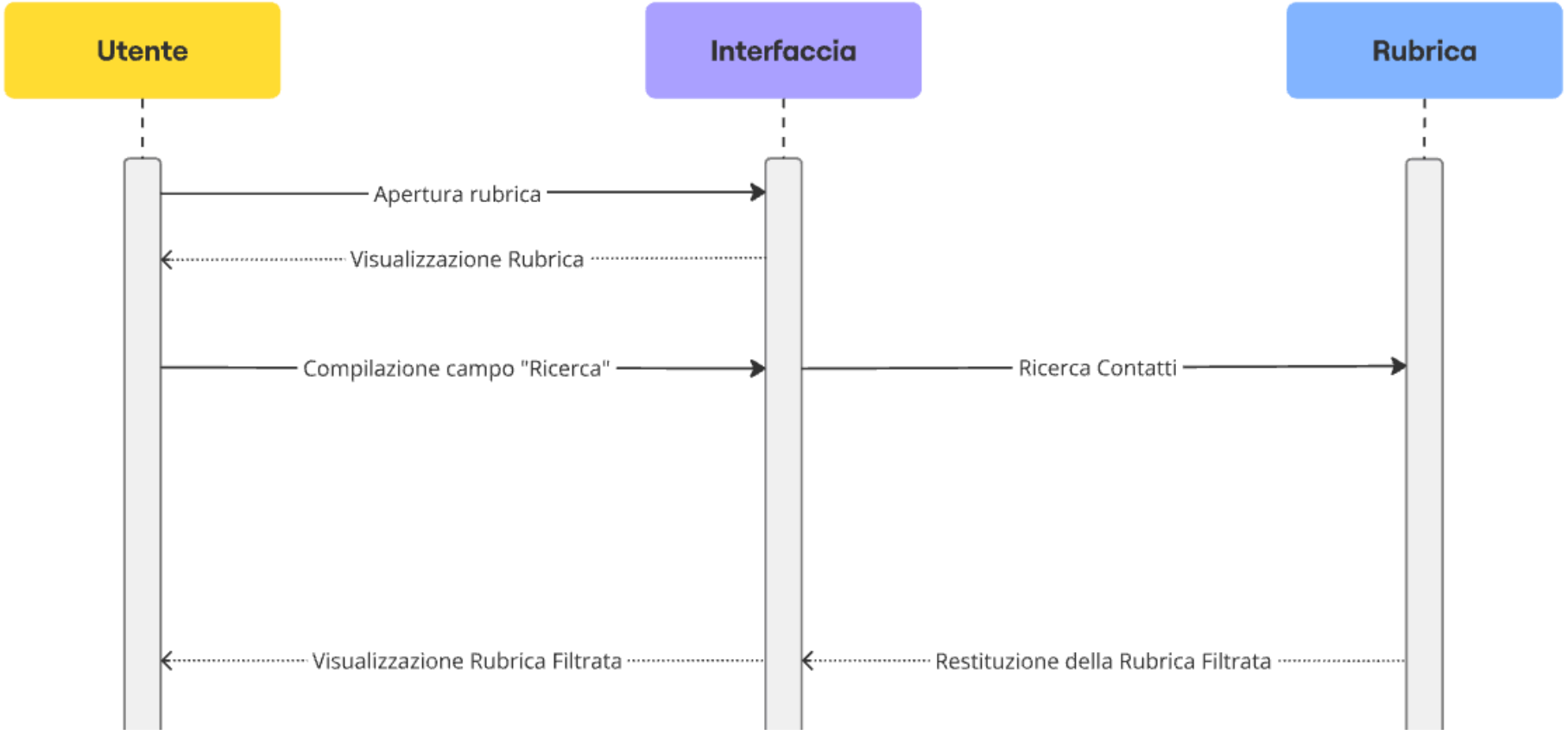
Modifica Contatto



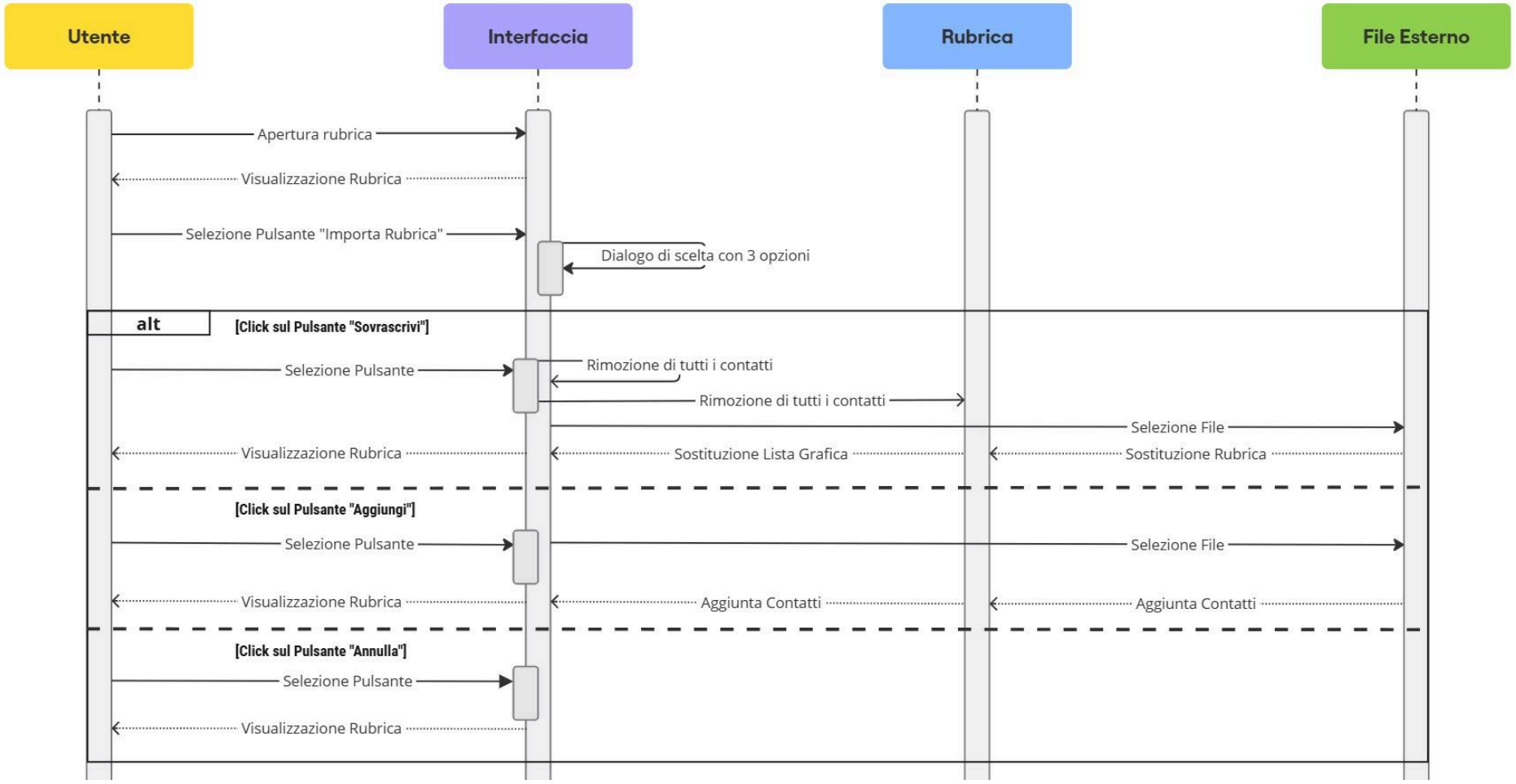
Eliminazione Contatto



Ricerca Contatto



Importazione Rubrica



Esportazione Rubrica