Progettazione del Software e Design

Rubrica telefonica

Componenti Gruppo 17:

Di Marino Domenico
Matricola: 0612707421

Adinolfi Giovanni
Matricola: 0612708352

Di Crescenzo Francesco
Matricola: 0612708640

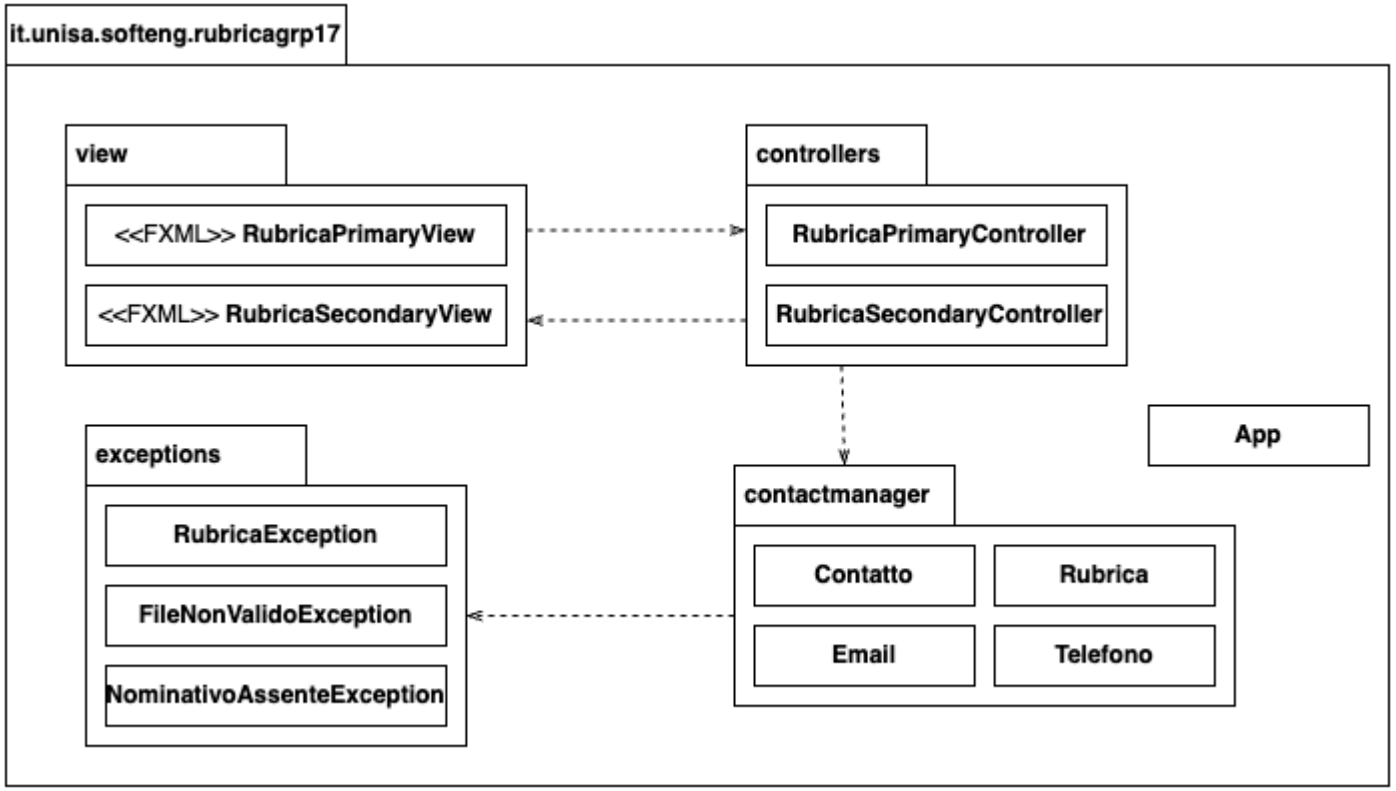
INDICE DEL DOCUMENTO

1.0 Design Architetturale

1.1 Diagramma dei package

I **diagrammi dei package** sono una componente dei diagrammi UML utilizzati per rappresentare l'organizzazione e la suddivisione logica di un sistema in moduli chiamati **package**. L'intero progetto risiede nel package "*it.unisa.softeng.rubricagrp17*", con all'interno di esso vari package per le varie funzioni presenti. Il package "*view*" contiene la struttura dell'interfaccia grafica e "*controllers*" quella che la gestisce. Il package "*contactmanager*" la gestione della rubrica e infine "*exceptions*" contenente le eccezioni che si potrebbero verificare durante l'esecuzione del programma.

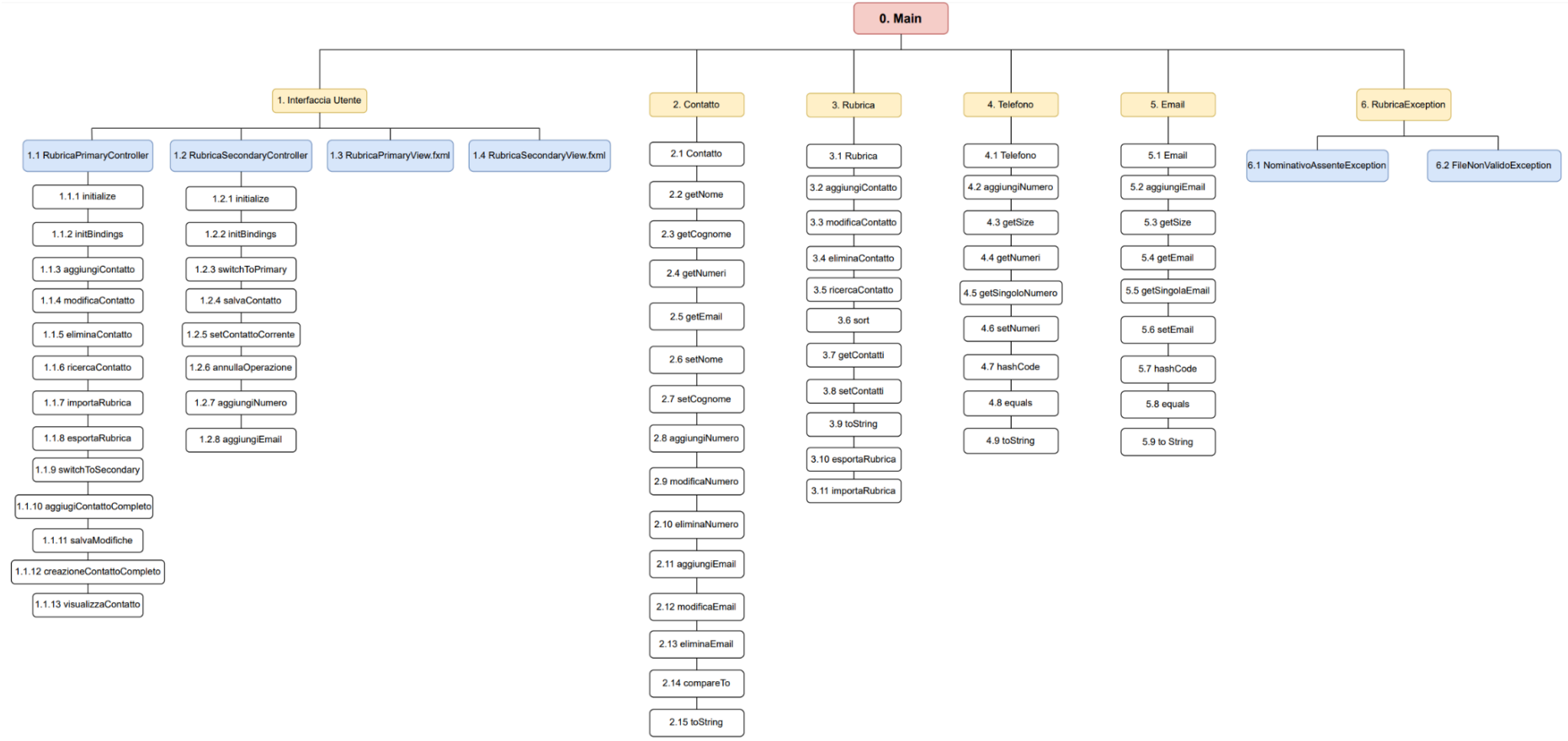
N.B. Per maggiore leggibilità visionare il PDF "PackageDiagram" presente nella cartella "Allegati".



1.2 Decomposizione dei moduli

La **suddivisione in moduli** ha l'obiettivo di semplificare l'implementazione del codice e migliorarne la struttura, rendendolo più leggibile, efficiente e facilmente modificabile. Consiste nel dividere il modulo principale in sottomoduli, ciascuno dedicato a una funzione specifica. Il diagramma seguente illustra come i moduli del nostro progetto sono stati suddivisi.

N.B. Per maggiore leggibilità visionare il PDF "ModuleDecomposition" presente nella cartella "Allegati".

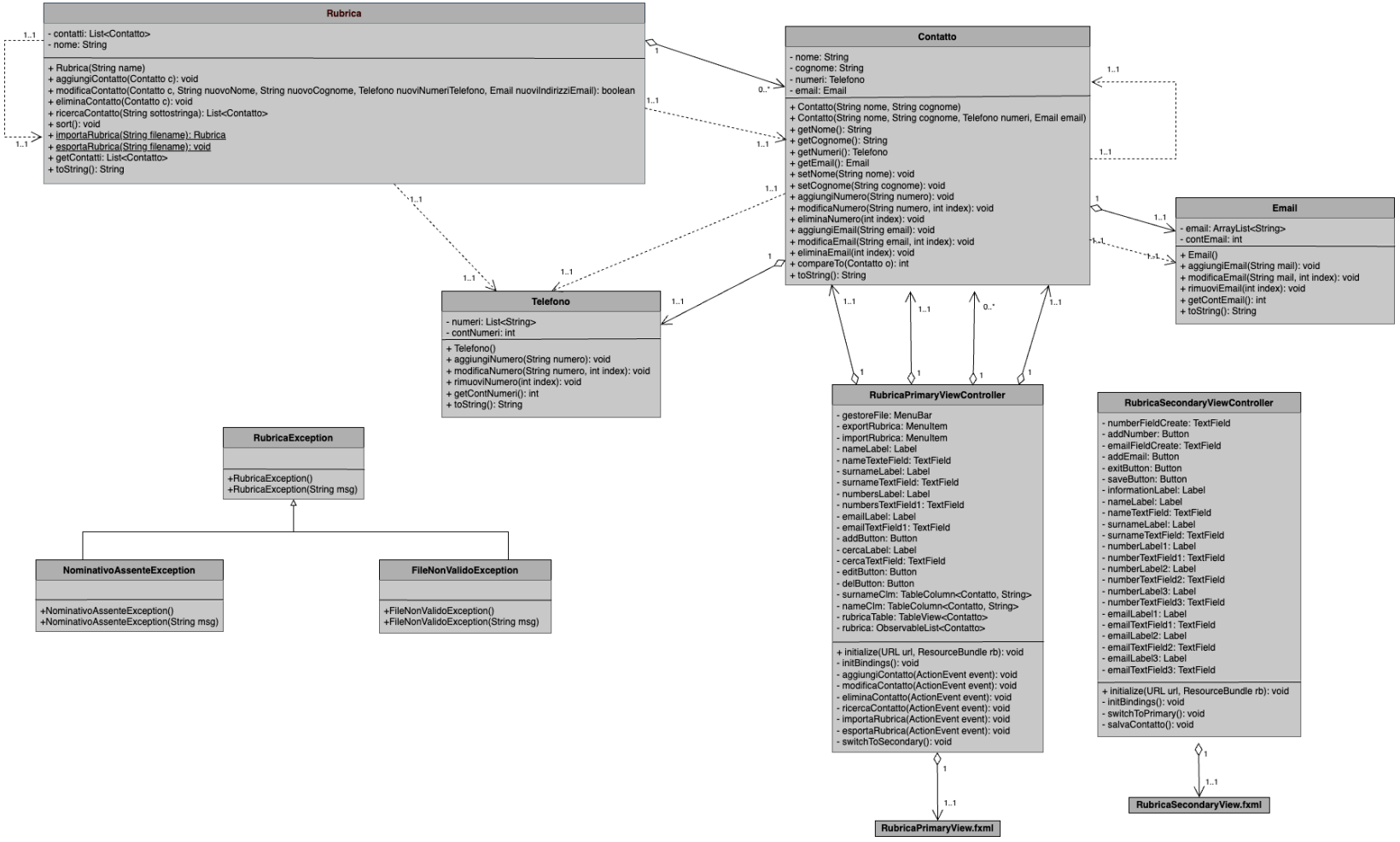


2.0 Design Funzionale

2.1 Diagramma di classe

Il seguente diagramma mostra la suddivisione del nostro sistema in “classi”, ognuna delle quali deve svolgere determinate operazioni. Il diagramma mostra come le classi si relazionano tra loro e eventuali dipendenze.

N.B. Per maggiore leggibilità visionare il PDF “*ClassDiagram*” presente nella cartella “*Allegati*”.



2.2 Diagramma di sequenza

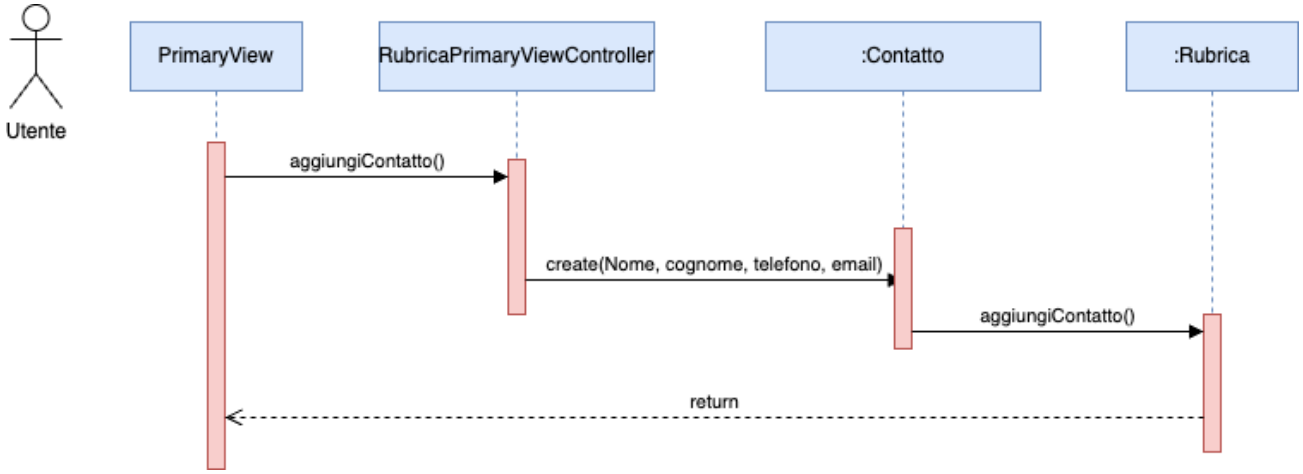
Ciascun diagramma presentato ha lo scopo di illustrare il funzionamento del sistema in specifici scenari. Per ogni scenario viene mostrata l'esecuzione di una possibile operazione effettuabile dall'utente. In linea generale, per le operazioni non rappresentate, le attività svolte dal sistema sono sostanzialmente simili, e il comportamento complessivo del sistema rimane invariato in ogni scenario.
N.B. Per maggiore leggibilità visionare il PDF "*SequenceDiagram*" presente nella cartella "*Allegati*".

2.2.1 Elenco Diagrammi di sequenza

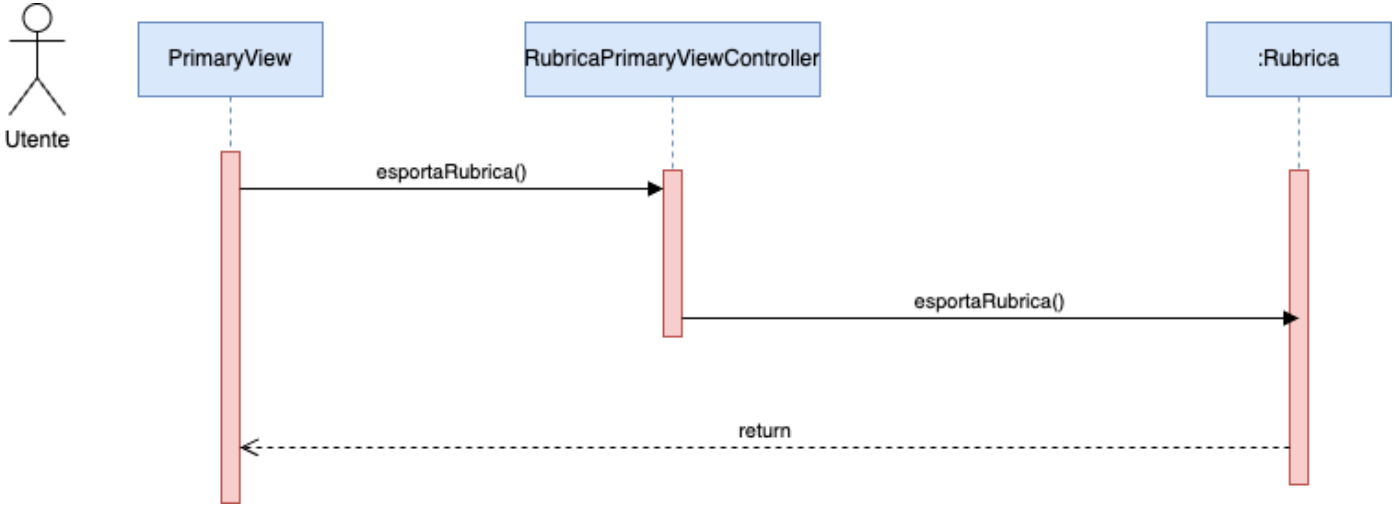
Di seguito è riportato l'elenco dei vari diagramma di sequenza più significativi:

- **Creazione contatto;**
- **Esportazione rubrica;**
- **Ricerca contatto;**
- **Importazione rubrica;**
- **Importazione rubrica con eccezione;**
- **Creazione contatto con eccezione.**

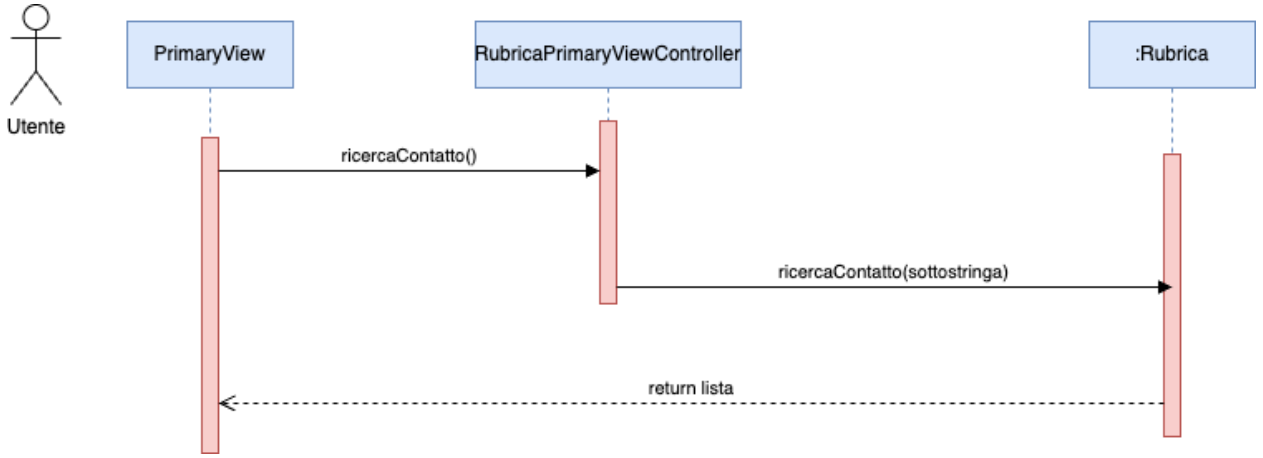
Creazione contatto



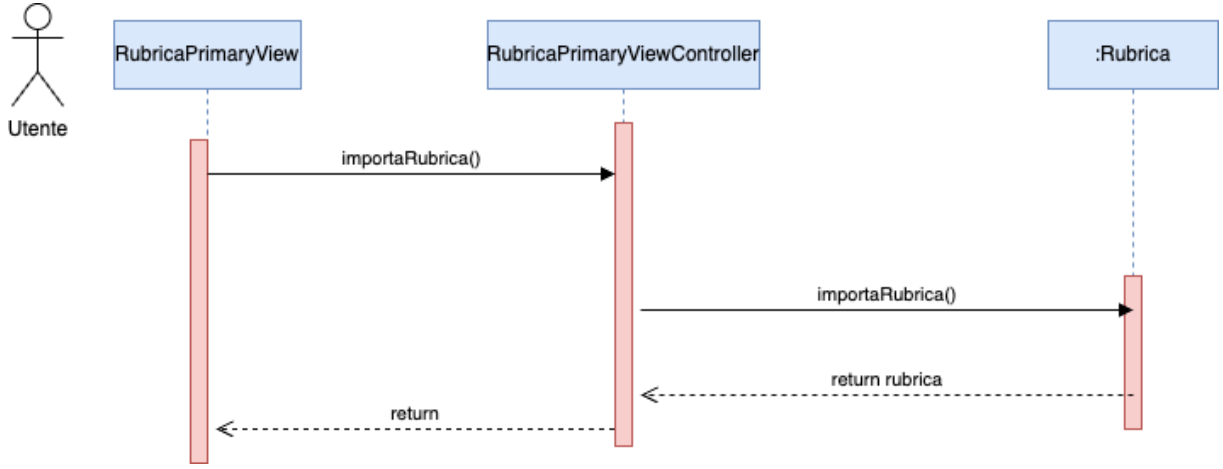
Esportazione rubrica



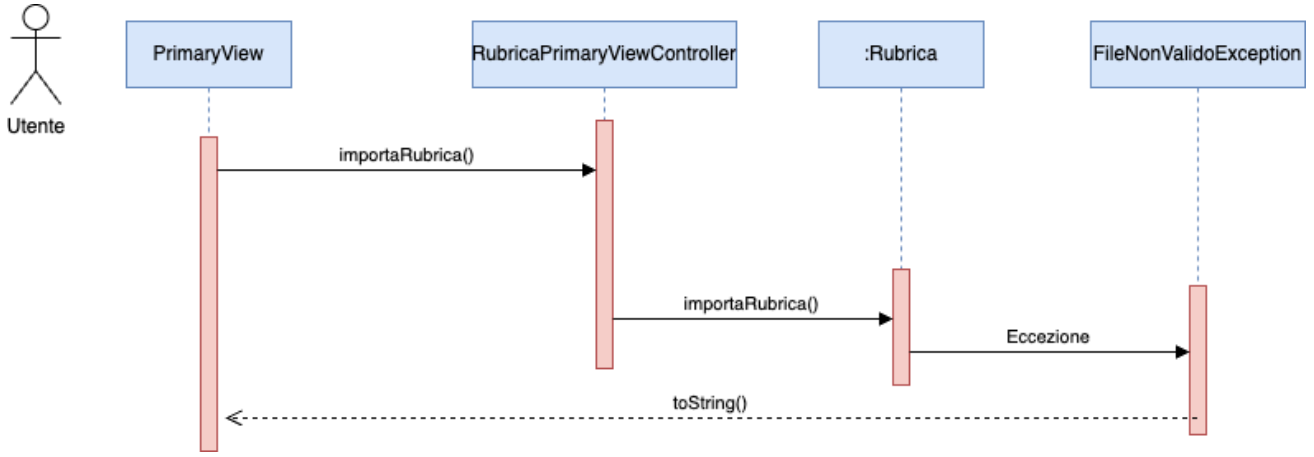
Ricerca contatto



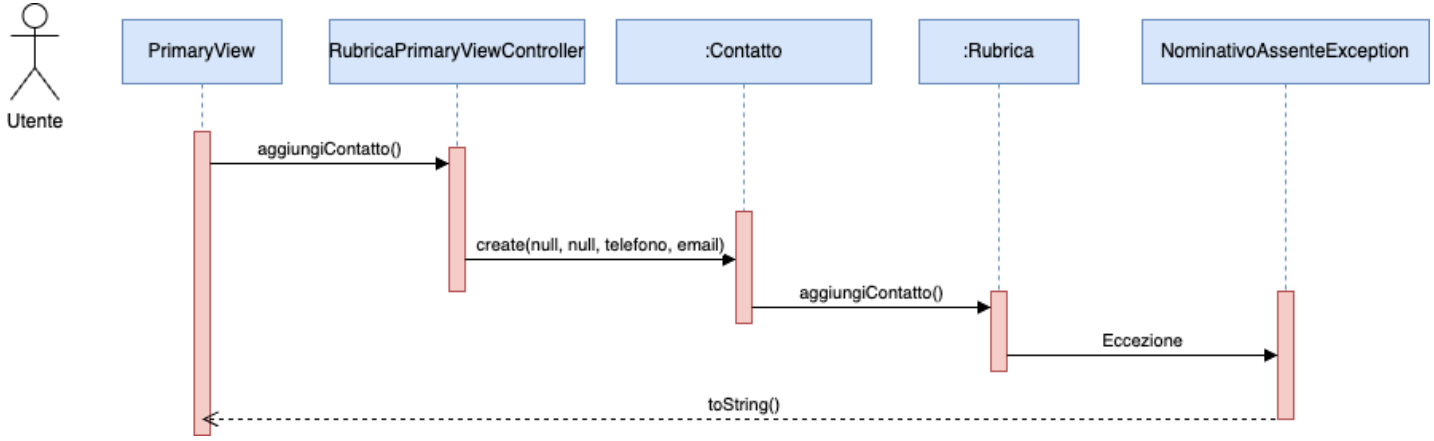
Importazione rubrica



Importazione rubrica con eccezione



Creazione contatto con eccezione



3.0 Principi generali di buona progettazione

Progettare accuratamente un software permette una produttività maggiore durante la fase di implementazione, in particolare nell'aggiunta di nuove funzionalità o nell'aggiornamento di quelle già esistenti. Bisogna quindi seguire dei principi di buona progettazione per poter riuscire nell'intento.

3.1 KISS: Keep It Simple, Stupid!

Secondo questo principio, l'obiettivo chiave è quello di evitare la complessità non necessaria, quindi preferire metodi brevi, classi semplici con poche operazioni e una struttura chiara e facile da comprendere. In questo progetto, le classi sono ben definite per ogni tipo di operazione.
In particolare, analizzando le classi **Rubrica**, **Contatto**, **Telefono** e **Email**:

- **Rubrica** gestisce la collezione di contatti;
- **Contatto** rappresenta un contatto con numeri di telefono e indirizzi email, oltre ai campi nome e cognome;
- **Telefono** gestisce i numeri di telefono;
- **Email** gestisce gli indirizzi email.

Ciascuna di queste classi ha quindi una responsabilità precisa e logica semplice.

3.2 DRY – Don't Repeat Yourself

Per questo principio, ogni "*idea*" deve essere implementata in un solo punto del codice sorgente. Ciò vuol dire che, se ci sono due pezzi di codice molto simili nel programma, è preferibile creare un'astrazione di questo algoritmo e richiamarlo due volte nel programma.
Il **DRY** è ben rispettato in quasi tutte le classi, specialmente nei controllori, che sono i punti focali per l'interazione con l'utente.

3.3 YAGNI - You Aren't Going to Need It

Il principio **YAGNI** consiglia di non introdurre funzionalità non necessarie. Ciò perché dedicare tempo a queste funzionalità sottrae tempo utile per lo sviluppo di funzionalità che invece sono fondamentali per il funzionamento dell'applicazione.
Tutte le funzionalità introdotte nel progetto dell'applicazione seguono i requisiti richiesti senza implementare funzionalità aggiuntive.

3.4 Separazione delle preoccupazioni (Separation of Concerns) e ortogonalità

È importante fare in modo che la gestione di aspetti diversi del sistema sia affidata a moduli distinti e non sovrapposti, in modo da semplificare lo sviluppo e la manutenzione delle applicazioni software e favorire la manutenibilità. Nel design proposto, i controlli dell'interfaccia sono separati dalla gestione dei dati implementata nelle classi Rubrica e Contatto. Inoltre, insieme a queste ultime due classi, anche le classi Telefono e Email rappresentano chiaramente i dati da gestire.
Si ha poi una suddivisione specifica delle eccezioni, in particolare per "*NominativoAssenteException*" e "*FileNonValidoException*". Queste scelte progettuali garantiscono una maggiore scalabilità, manutenibilità e facilità di test.

3.5 Principio della minima sorpresa

Il **principio della minima sorpresa** consiglia di strutturare il codice in modo da non confondere il lettore o sviluppatore che si occuperà della manutenzione. Ciò vuol dire che bisogna fare in modo da usare sempre le stesse convenzioni per i nomi ed evitare di scrivere il codice in maniera contorta. Con questa proposta di design si punta a mantenere una coerenza di nominativi per non indurre in confusione il lettore.

4.0 Principi generali di buona progettazione orientata agli oggetti

4.1 SOLID

La parola **SOLID** è un acronimo riferito a cinque principi dello sviluppo del software orientato agli oggetti. Questi principi favoriscono le caratteristiche di alta coesione e basso accoppiamento:

- **Principio della singola responsabilità** (*Single responsibility principle*): Afferma che ogni classe dovrebbe avere una ed una sola responsabilità, interamente incapsulata al suo interno. Ogni classe del nostro progetto ha una singola responsabilità chiara.
  Ad esempio, le classi Telefono e Email si occupano solo di metodi riguardanti numeri di telefono e email. Ciò favorisce la manutenibilità permettendo al manutentore di apportare modifiche solo nella classe interessata.

- **Principio aperto/chiuso** (*Open/closed principle*): Afferma che un'entità software dovrebbe essere aperta alle estensioni, ma chiusa alle modifiche, in maniera tale che un'entità possa permettere che il suo comportamento sia modificato senza alterare il suo codice sorgente. Le classi sfruttano il principio dell'incapsulamento, vietando quindi l'accesso ai dettagli implementativi e "chiudendo alla modifica". Sono strutturate in modo tale da poter essere estendibili senza apportare modifiche al codice.

- **Principio di sostituzione di Liskov** (*Liskov substitution principle*): Secondo questo principio, gli oggetti dovrebbero poter essere sostituiti con dei loro sottotipi, senza alterare il comportamento del programma che li utilizza. Nel nostro progetto è presente "*RubricaException*" che rappresenta l'eccezione generica dell'applicazione, ma anche i sottotipi "*FileNonValidoException*" e "*NominativoAssenteException*" che rappresentano eccezioni specifiche ma mantengono il comportamento della superclasse.

- **Principio di segregazione delle interfacce** (*ISP*): Il principio consiglia di usare più interfacce specifiche costituite da pochi metodi, che una singola generica interfaccia costituita da più metodi che poi saranno inutilizzati. Nel nostro design è presente solo l'interfaccia "*Comparable*" che è usata solo dalla classe Contatto, per permettere il confronto per il nominativo, e le interfacce "*Initializable*" per i controller, necessarie per il loro corretto funzionamento.

- **Principio di inversione della dipendenza** (*DIP*): Secondo questo principio, una classe dovrebbe dipendere dalle astrazioni, non da classi concrete. Come accennato in precedenza, non ci sono superclassi, se non "*RubricaException*" che si occupa solo dell'eccezione generica.

4.2 Privilegiare l'associazione rispetto all'ereditarietà

Per avere un minore accoppiamento tra le classi, è preferibile usare l'associazione piuttosto che l'ereditarietà. Questo perché una sottoclasse si impegna a rispettare il contratto della classe base, ed è più facile infrangere il principio di sostituzione di Liskov.

Facendo riferimento al nostro design, questo principio è rispettato con molti casi di aggregazione, ad esempio tra le classi "*Rubrica*" e "*Contatto*", o anche tra "*Contatto*" e "*Telefono*" o "*Email*". Infatti, queste ultime due classi sono da intendere come "*parte di*" Contatto, così come la classe "*Contatto*" è parte di "*Rubrica*".

4.3 Principio di robustezza

I componenti dell'applicazione vanno progettati in modo tale da essere "**robusti**", ossia bisogna limitare i danni nel caso in cui le precondizioni non siano rispettate. Ciò significa garantire che non ci sia una perdita di dati per il client e che il sistema non si porti in uno stato inconsistente. L'uso delle eccezioni personalizzate contribuisce a rendere il sistema robusto e a prevenire errori imprevisti.

5.0 Coesione

La **coesione** misura quanto le parti che sono incluse nello stesso modulo sono legate tra di loro:

- La classe **Telefono** ha coesione *funzionale*, perché ogni metodo opera direttamente e specificamente sull'attributo numero. Ogni metodo contribuisce a una singola responsabilità come rappresentare e/o manipolare un numero di telefono;

- **Email**, così come per la classe Telefono, ha coesione *funzionale*, dato che si agisce sull'attributo mail;

- **Contatto**, tutte le funzionalità presenti nella classe contribuiscono alla gestione della struttura dati Contatto. I metodi operano tutti direttamente sugli attributi interni per mantenere coerente lo stato della struttura dati. Di conseguenza, questa classe ha coesione *funzionale*;

- La classe **Rubrica** ha una coesione *comunicazionale*. Molti metodi condividono la stessa struttura dati sottostante contatti e ne modificano lo stato;

- **Controller** hanno una coesione *temporale*, perché contengono i metodi "initialize" e "initBindings" che vengono eseguiti all'apertura dell'applicazione.

6.0 Accoppiamento

6.1 Coppie di moduli e tipi di accoppiamento

1. **Rubrica** ↔ **Contatto**

   Tipo di accoppiamento: Per dati
   La classe Rubrica dipende direttamente dalla classe Contatto, in quanto gestisce una collezione di oggetti Contatto. I metodi di Rubrica manipolano i dati di Contatto attraverso i metodi di aggiunta, rimozione, modifica contatto ecc.

2. **Rubrica** ↔ **Telefono**

   Tipo di accoppiamento: Nessuno
   Non c'è una relazione diretta tra Rubrica e Telefono. La classe Telefono è una proprietà di Contatto, e quindi qualsiasi interazione avviene tramite il modulo Contatto.

3. **Rubrica** ↔ **Email**

   Tipo di accoppiamento: Nessuno
   Analogamente a Telefono, non c'è una relazione diretta tra Rubrica ed Email. Anche qui, ogni interazione avviene attraverso Contatto.

4. **Contatto** ↔ **Telefono**

   Tipo di accoppiamento: Per dati
   La classe Contatto ha una lista di oggetti Telefono come proprietà. I metodi di Contatto manipolano direttamente questa lista (aggiunta, rimozione, ecc.).

5. **Contatto** ↔ **Email**

   Tipo di accoppiamento: Per dati
   Simile a Telefono, la classe Contatto gestisce una lista di oggetti Email.

6. **RubricaPrimaryController ↔ Rubrica**

   Tipo di accoppiamento: Per dati
   Il controller primario (RubricaPrimaryController) interagisce con la classe Rubrica per manipolare la struttura dati sottostante (aggiunta, ricerca, rimozione di contatti). Il controller dipende da metodi pubblici di Rubrica e utilizza i dati forniti dalla stessa.

7. **RubricaPrimaryController ↔ Contatto**

   Tipo di accoppiamento: Per timbro
   Il controller lavora con oggetti Contatto per visualizzarli e modificarli nella GUI. Questo accoppiamento avviene principalmente nei parametri e nei ritorni dei metodi.

8. **RubricaPrimaryController ↔ RubricaSecondaryController**

   Tipo di accoppiamento: Per controllo
   Dovendo gestire una finestra secondaria (per la modifica di un contatto) bisogna passare dati al controller secondario, quindi si ha un accoppiamento per controllo.

9. **RubricaSecondaryController ↔ Contatto**

   Tipo di accoppiamento: Per timbro
   Il controller secondario gestisce oggetti Contatto per modificare o aggiungere dati alla rubrica.