Design

Gioia Errico, Cutolo Raffaele, D'Amico Christian, D'Amico Antonio Pio

Indice

1 Diagramma delle Classi

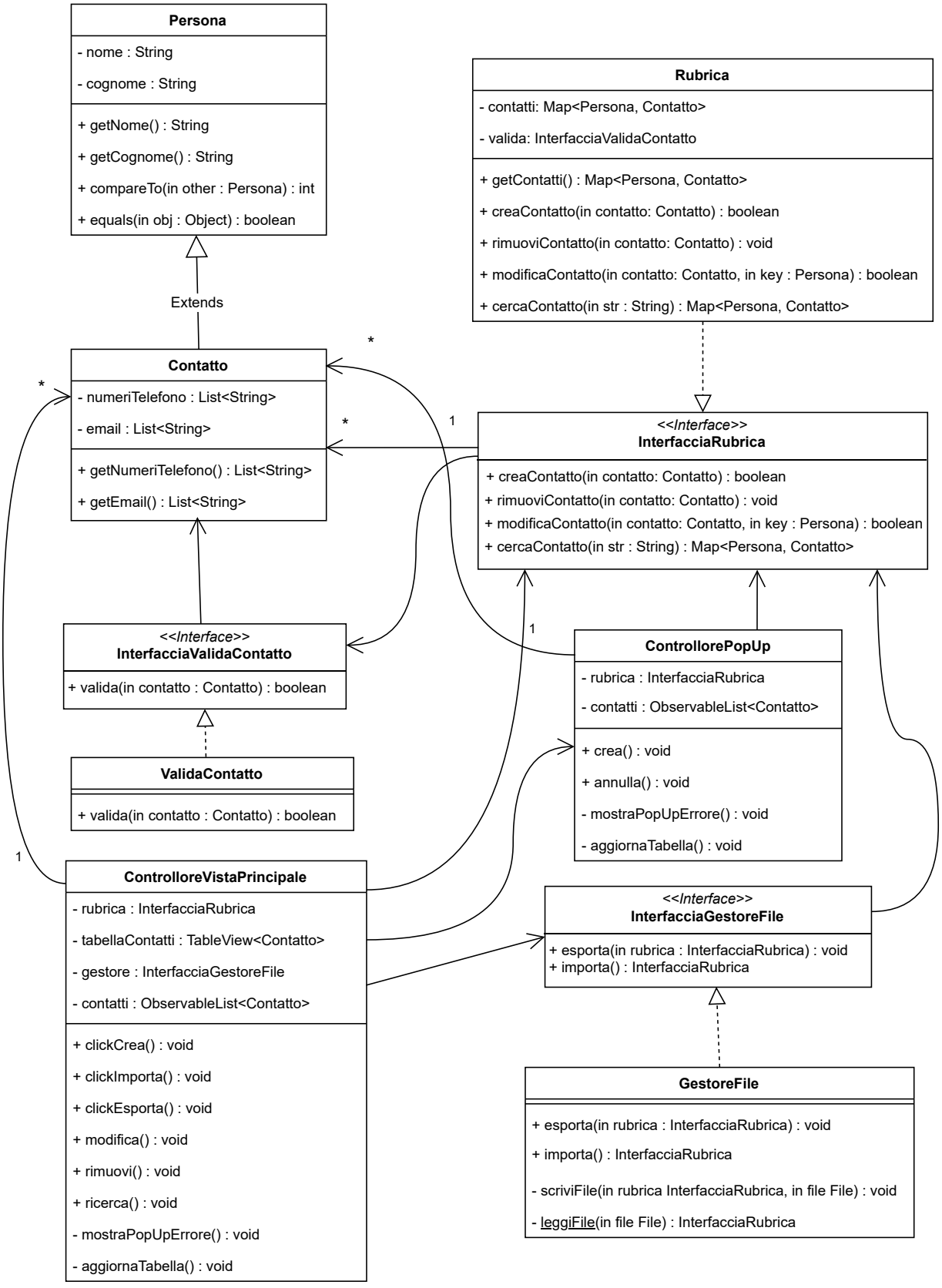

Figura 1: Diagramma delle Classi.

2 Diagrammi delle Sequenze

2.1 Creazione Contatto

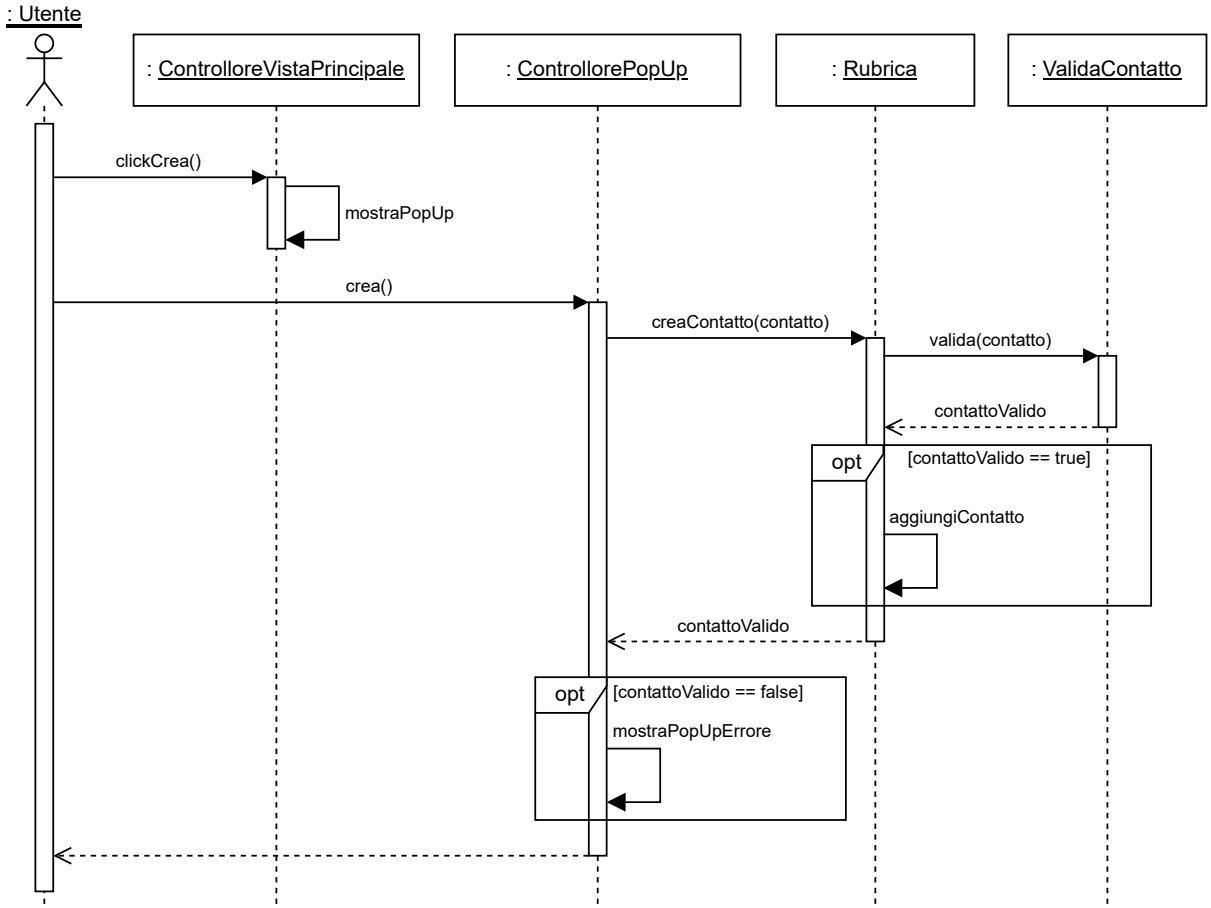

Figura 2: Diagramma di Sequenza - Creazione Contatto.

2.2 Rimozione Contatto

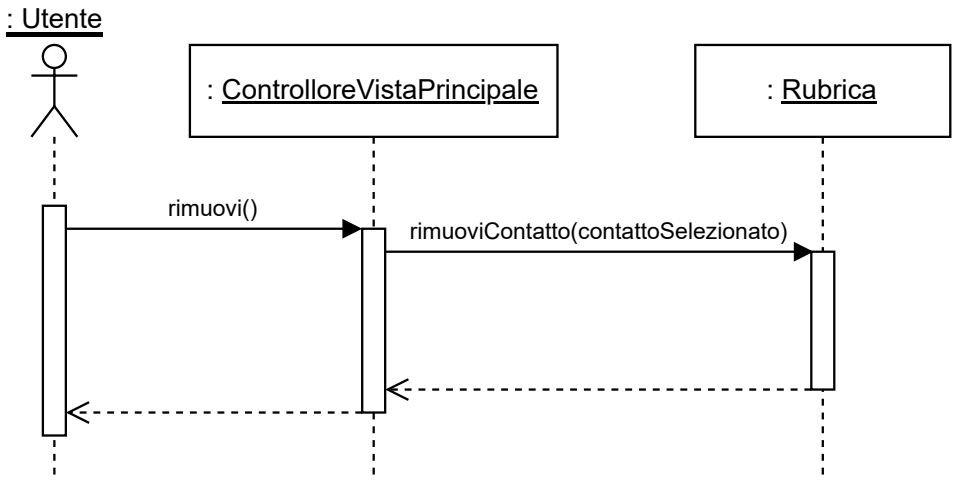

Figura 3: Diagramma di Sequenza - Rimozione Contatto.

2.3 Importa/Esporta Rubrica

Come per il caso `Creazione/Modifica` il diagramma di sequenza per l'azione `Importa` è fortemente affine a quello di `Esporta`, infatti, trascurando la questione dei nomi (ad esempio `clickImporta()` → `clickEsporta()`), la sequenza dei passi risulta identica.
Ciononostante, all'operazione di lettura del file dell'azione `Importa` corrisponde quella di scrittura nel caso di `Esporta`. Inoltre, il metodo `esporta()` di `GestoreFile` non restituisce alcun dato, a differenza di `importa()` che ritorna un oggetto di tipo `Rubrica`, e di conseguenza il `ControlloreVistaPrincipale` non necessita di aggiornare la rubrica.

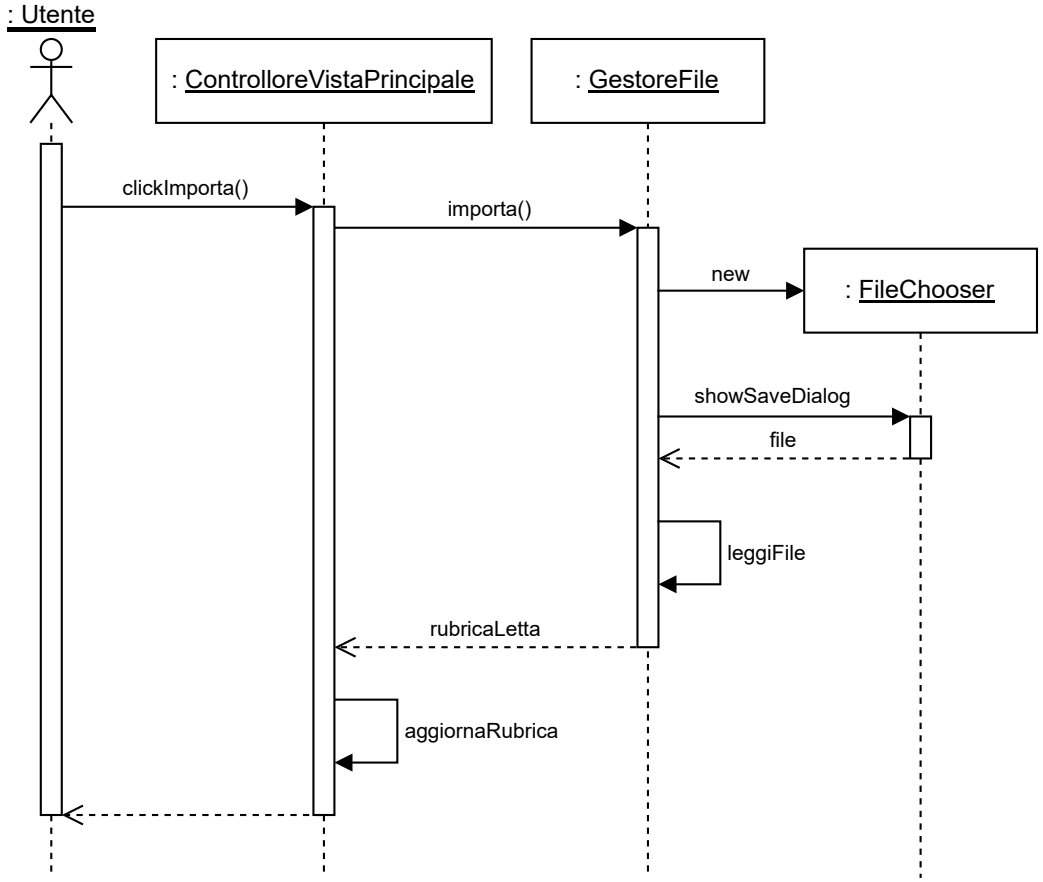

Figura 4: Diagramma di Sequenza - Importa/Esporta Rubrica.

2.4 Modifica Contatto

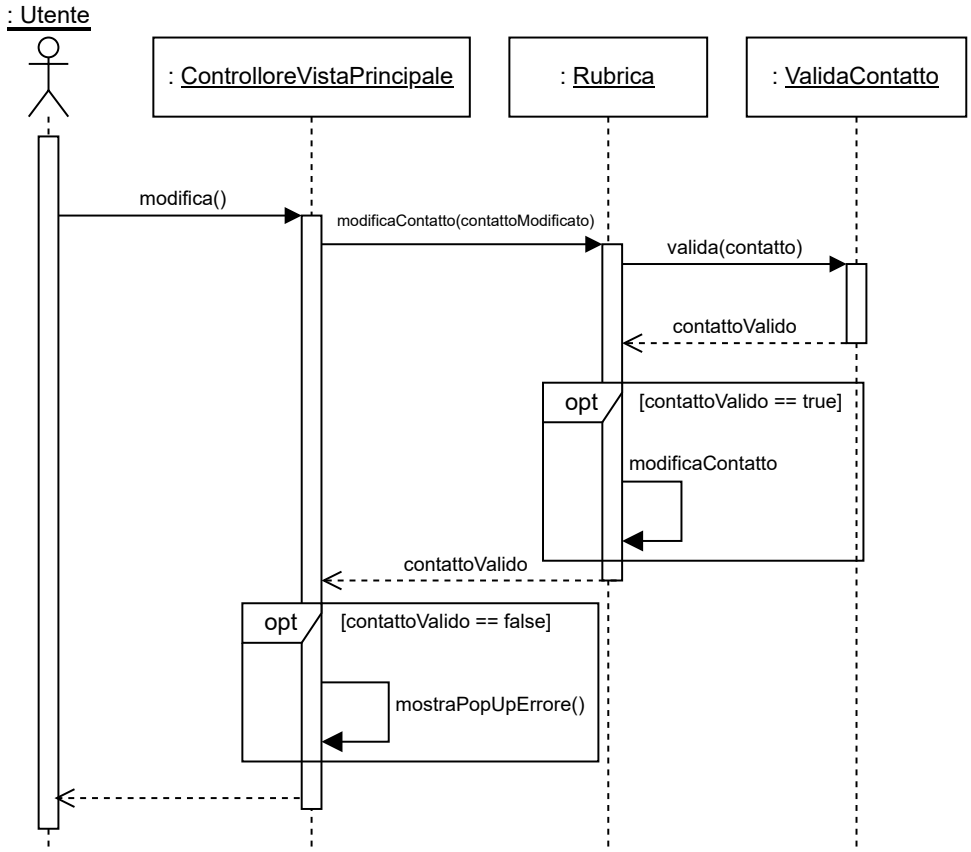

Figura 5: Diagramma di Sequenza - Modifica Contatto.

Eventuali metodi di utilità che sono stati rilevati a valle del diagramma di sequenza non sono stati inseriti nel diagramma delle classi in quanto non risultano di fondamentale importanza e si è deciso di non appesantire inutilmente il diagramma delle classi.

2.5 Ricerca Contatto

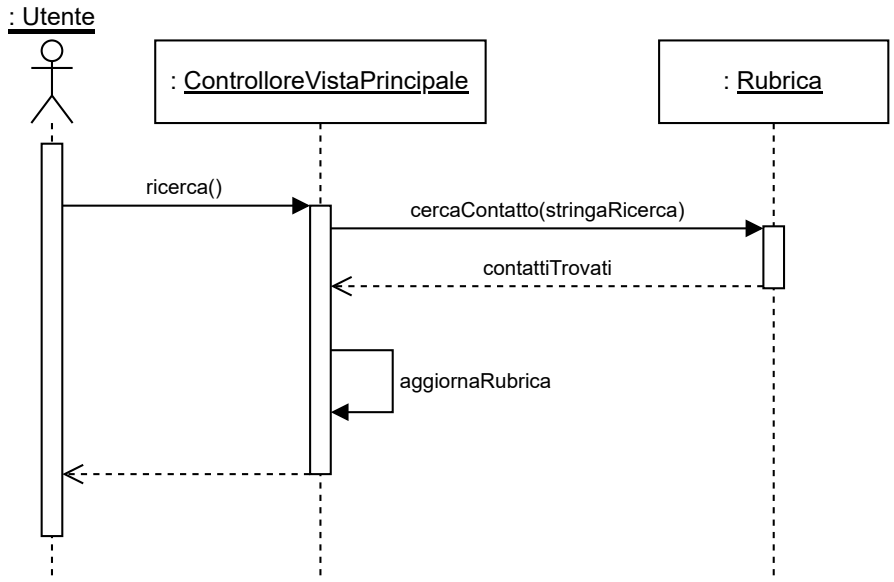

Figura 6: Diagramma di Sequenza - Ricerca Contatto.

3 Diagramma dei Package

Per organizzare le classi e delineare chiaramente le dipendenze tra i package, possiamo suddividerle.

- **Package:** `it.unisa.diem.softeng.modello`

  Contiene le classi che rappresentano gli oggetti di dominio, ossia i dati fondamentali utilizzati nella rubrica.

  1) `Persona`: Classe di base per rappresentare una persona con nome e cognome.
  2) `Contatto`: Estende la classe `Persona`, aggiungendo numeri di telefono ed email.

  Dipendenze: N/A

- **Package:** `it.unisa.diem.softeng.servizio`

  Contiene interfacce e classi responsabili della gestione della logica applicativa.

  1) `InterfacciaRubrica`: Interfaccia che definisce le operazioni della rubrica.
  2) `Rubrica`: Implementazione dell'interfaccia `InterfacciaRubrica`. Gestisce i contatti come una mappa osservabile.
  3) `InterfacciaValidaContatto`: Interfaccia per la validazione dei contatti.
  4) `ValidaContatto`: Implementazione dell'interfaccia `InterfacciaValidaContatto`.

  Dipendenze: `it.unisa.diem.softeng.modello`

- **Package:** `it.unisa.diem.softeng.persistenza`

  Contiene interfacce e classi per la gestione della persistenza dei dati.

  1) `InterfacciaGestoreFile`: Interfaccia per la gestione di import/export.
  2) `GestoreFile`: Implementazione dell'interfaccia `InterfacciaGestoreFile`.

  Dipendenze: `it.unisa.diem.softeng.servizio`

- **Package:** `it.unisa.diem.softeng.controllo`

  Contiene i controller responsabili dell'interazione tra la vista e il modello.

  1) `ControlloreVistaPrincipale`: Gestisce le azioni principali dell'applicazione (creazione, ricerca, modifica, eliminazione di contatti).
  2) `ControllorePopUp`: Gestisce l'interazione per creare o annullare modifiche ai contatti.
  3) `Main`: Richiama la vista principale dell'applicazione JavaFX e carica le risorse necessarie all'esecuzione.

  Dipendenze:

  1) `it.unisa.diem.softeng.servizio`
  2) `it.unisa.diem.softeng.persistenza`

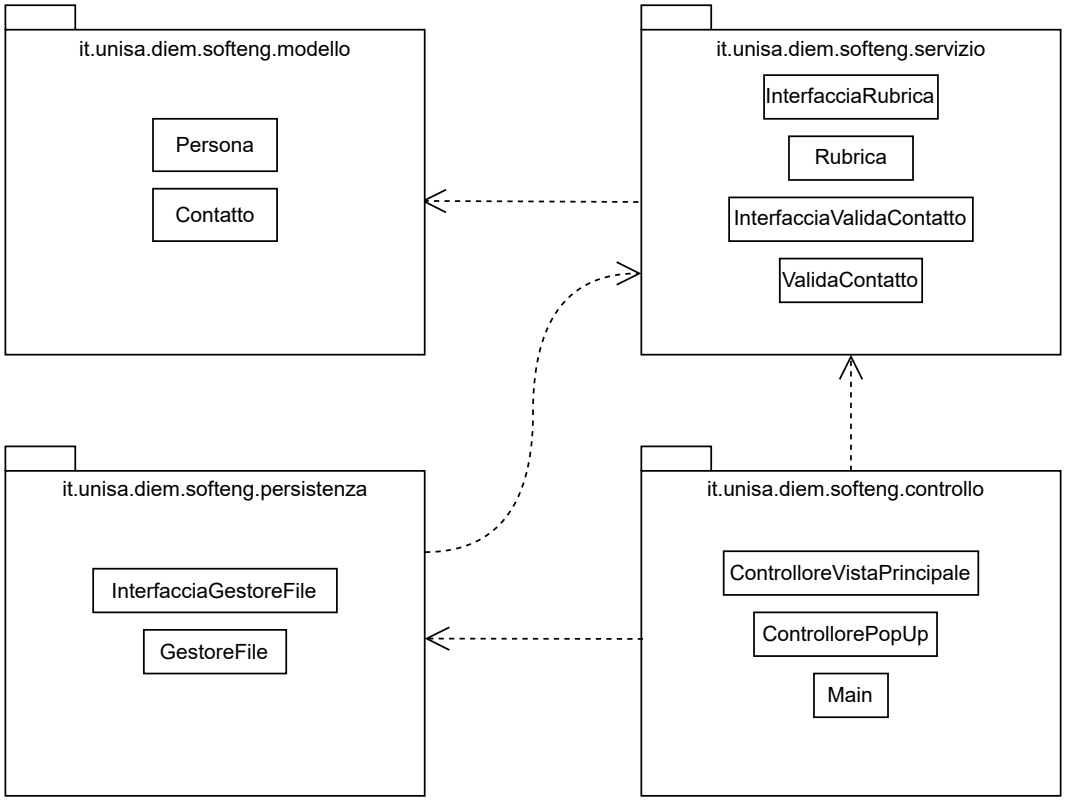

Figura 7: Diagramma dei Package.

4 Struttura del Sistema e Scelte Progettuali

4.1 Coesione

La coesione è una misura della qualità interna di un modulo e si riferisce al grado in cui le responsabilità al suo interno sono strettamente correlate e ben focalizzate. Un modulo con alta coesione svolge un unico compito ben definito e racchiude tutte le operazioni strettamente necessarie per completare tale compito. Al contrario, una bassa coesione indica che un modulo è dispersivo, svolge più compiti non correlati o dipende eccessivamente da altri moduli per completare le sue operazioni. L'obiettivo di questo sistema è massimizzare la coesione all'interno di ogni modulo per migliorare la manutenibilità, leggibilità e riusabilità del codice.
In questa sezione, analizzeremo il livello di coesione dei moduli presenti nel sistema.

- **Persona**

  Fornisce un modello base per rappresentare una persona con gli attributi nome e cognome. Questa classe ha un livello di coesione **funzionale**, poiché tutti i metodi servono ad un unico scopo: la gestione e il confronto. Questa classe viene estesa da `Contatto` per aggiungere nuove funzionalità, migliorando la riutilizzabilità.

- **Contatto**

  La classe si occupa di un unico compito: rappresentare un contatto, includendo numeri di telefono ed email. Ogni metodo contribuisce direttamente a questa responsabilità primaria. La coesione è **funzionale**, poiché tutte le sue funzionalità sono orientate esclusivamente alla gestione dei dati di un contatto.

- **Rubrica**

  È il cuore del sistema, responsabile della gestione di una collezione di contatti attraverso operazioni strettamente correlate come `aggiungiContatto`, `rimuoviContatto`, oltre a `modificaContatto` e `cercaContatto`. Ogni metodo si occupa di una parte necessaria della gestione complessiva dei contatti. Il modulo presenta, dunque, un livello di coesione **funzionale**. La scelta di utilizzare una mappa osservabile ("ObservableMap") per memorizzare i contatti riflette l'intento di mantenere il sistema efficiente dal punto di vista computazionale.

- **GestoreFile**

  Questa classe è progettata per gestire esclusivamente l'importazione e l'esportazione dei dati della rubrica, mantenendo una singola responsabilità focalizzata sulle operazioni di I/O. La sua coesione è **funzionale**, poiché le attività svolte appartengono a un unico compito logico: la gestione dei file. Le operazioni sono accomunate dall'obiettivo di garantire un flusso organizzato per la lettura e la scrittura dei dati.

- **ValidaContatto**

  Una classe dedicata unicamente alla validazione dei dati di un contatto. La sua coesione è **funzionale**, in quanto si occupa esclusivamente di verificare l'integrità dei dati. Grazie a questa implementazione, la classe è facile da manutenere e può essere riutilizzata in altri contesti in cui è necessario verificare la validità di dati analoghi.

- **ControlloreVistaPrincipale**

  Questa classe si occupa di gestire la vista principale dell'interfaccia grafica, fungendo da mediatore tra la rubrica e gli elementi della vista. La coesione è **procedurale**, poiché la classe include funzionalità che vengono spesso usate insieme e coordinano operazioni principali come l'apertura di pop-up o la gestione della rubrica. Questi metodi sono utilizzati in stretta connessione tra loro per realizzare funzionalità che devono lavorare in sinergia.

- **ControllorePopUp**

  Si occupa della gestione della finestra PopUp utilizzata per la creazione di contatti o, eventualmente, l'annullamento di suddetta operazione. La coesione di questo componente è **procedurale**, poiché i metodi implementati all'interno della classe o del modulo fanno parte di un processo comune.

4.2 Accoppiamento

Nella progettazione di sistemi software, il concetto di accoppiamento descrive il grado di dipendenza tra moduli, ovvero il modo in cui i componenti di un sistema interagiscono e condividono informazioni. Un buon design mira a ridurre il livello di accoppiamento per migliorare la modularità, la manutenibilità e la riusabilità del codice. Tuttavia, è inevitabile che esista un certo grado di accoppiamento, poiché i moduli devono interagire per svolgere le loro funzionalità.
In questo contesto, analizzeremo il livello di accoppiamento tra i moduli presenti nel diagramma delle classi partendo dal livello peggiore.

Nel diagramma delle classi in questione non ci sono casi di **accoppiamento per contenuto**. La scelta di utilizzare delle interfacce quali `InterfacciaRubrica`, `InterfacciaGestoreFile` e `InterfacciaValidaContatto` fa si che tutti i moduli accedano agli altri tramite interfacce o metodi pubblici. Nessun modulo accede direttamente ai campi privati di un altro modulo.

Non è presente neanche un caso di **accoppiamento per aree comuni**. Nel diagramma in questione sono assenti proprietà di tipo `static`, ciò giustifica l'assenza di accesso ad aree comuni tra le classi.

Non sono presenti casi di **accoppiamento per controllo**, ovvero situazioni in cui un modulo passa informazioni di "controllo" a un altro, in base alle quali viene cambiata la logica di funzionamento di quest'ultimo.

Non sono presenti casi di **accoppiamento per timbro**. Nessun modulo passa ad un altro una struttura dati contenente anche informazioni che non sono necessarie all'altro modulo.

Sono presenti invece dei casi di **accoppiamento per dati**. L'accoppiamento tra `GestoreFile` e `Rubrica`, così come tra i controllori e `Rubrica`, è accoppiamento per dati, poiché ogni modulo interagisce con l'altro solo tramite i dati strettamente necessari al suo funzionamento, utilizzando un'interfaccia ben definita.
Nonostante la validazione di un contatto influisce sul comportamento di `Rubrica` (determinando se un'operazione come l'aggiunta o la modifica di un contatto può essere eseguita), questo caso rientra negli accoppiamenti per dati, poiché l'interazione si basa sui dati strettamente necessari.

Come previsto nella maggior parte dei sistemi, non sono presenti casi di **nessun accoppiamento**. Nel diagramma in questione ogni modulo interagisce almeno con un altro, direttamente o indirettamente, tramite metodi pubblici o interfacce.

5 Analisi dei Principi di Progettazione

Il diagramma delle classi è un esempio di come i principi di buona progettazione possano essere applicati per creare un sistema modulare, estensibile e manutenibile. In questo caso, i principi KISS (Keep It Simple, Stupid!), SINE (Simple Is Not Easy), DRY (Don't Repeat Yourself), YAGNI (You Aren't Going to Need It), separazione delle preoccupazioni e ortogonalità, oltre ai principi di Single Responsibility Principle (SRP), Open-Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP) e Dependency Inversion Principle (DIP), sono stati rispettati in modo efficace.

5.1 Principi di Buona Progettazione

Il sistema è progettato per essere semplice e facilmente comprensibile, grazie alla chiara separazione delle responsabilità tra le varie classi, che facilita la comprensione e l'uso del codice. Le interfacce sono piccole e focalizzate, rispettando il principio di ortogonalità, il che significa che ogni componente del sistema può essere sviluppato e testato in modo indipendente.

Inoltre, il principio DRY è rispettato poiché non ci sono duplicazioni di codice; le funzionalità comuni sono astratte in interfacce, permettendo di riutilizzare il codice senza ripetizioni. Il sistema è anche estensibile, consentendo l'aggiunta di nuove funzionalità senza la necessità di modificare il codice esistente, in linea con il principio YAGNI, che incoraggia a non implementare funzionalità non necessarie.

Infine, un esempio di manutenibilità del sistema è garantito dalla possibilità di cambiare l'implementazione della validazione senza dover intervenire sulla classe `Rubrica`, dimostrando così un'adeguata separazione delle preoccupazioni. In sintesi, il design del sistema non solo rispetta, ma incarna questi principi fondamentali, rendendolo robusto e facile da gestire.

5.2 Principi SOLID

La classe `Rubrica` è un esempio di come il **Single Responsibility Principle** possa essere applicato. La classe si occupa di gestire i contatti, ma non si occupa della validazione degli stessi. Questa responsabilità è stata delegata alla classe `ValidaContatto`, che implementa l'interfaccia `InterfacciaValidaContatto`. Questo approccio consente di separare le responsabilità e di creare un sistema più modulare.

La classe `Rubrica` è anche un esempio di come l'**Open-Closed Principle** possa essere applicato. La classe può essere estesa per supportare diverse strategie di validazione dei contatti semplicemente implementando nuove classi che seguono l'interfaccia `InterfacciaValidaContatto`. Questo consente di aggiungere nuove funzionalità senza modificare il codice esistente.

La classe `ValidaContatto` è un esempio di come il **Liskov Substitution Principle** possa essere applicato. La classe implementa l'interfaccia `InterfacciaValidaContatto` e può essere utilizzata al posto di altre classi che implementano la stessa interfaccia. Questo garantisce che le classi derivate possano sostituire le classi base senza problemi.

Le interfacce `InterfacciaValidaContatto`, `InterfacciaGestoreFile` e `InterfacciaRubrica` sono esempi di come l'**Interface Segregation Principle** possa essere applicato. Le interfacce sono specifiche per la validazione dei contatti, la gestione dei file e la gestione dei contatti, rispettivamente. Questo consente di mantenere le interfacce piccole e focalizzate, facilitando l'implementazione da parte di classi diverse.

Infine, la classe `Rubrica` è un esempio di come il **Dependency Inversion Principle** possa essere applicato. La classe dipende dall'interfaccia `InterfacciaValidaContatto` piuttosto che da una classe concreta. Questo riduce il coupling e rende il sistema più flessibile, poiché è possibile cambiare l'implementazione della validazione senza modificare la classe `Rubrica`.

In sintesi, il diagramma delle classi fornito è un esempio di come i principi di buona progettazione possano essere applicati per creare un sistema modulare, estensibile e manutenibile. La separazione delle responsabilità, l'uso di interfacce e la riduzione del coupling sono tutti aspetti importanti del design del sistema.