

Corso di Ingegneria del Software Deliverable di progetto	2023-2024
---	-----------

“Ingegneria del Software” 2023-2024

Docente: Prof. Angelo Furfaro

Organigramma aziendale

Data	<30/12/2024>
Documento	Documento Finale – D3

Team Members		
Nome e Cognome	Matricola	E-mail address
Antonio Parise	230946	prsntn02p07h919p@studenti.unical.it

1	
---	--

Sommario

List of Challenging/Risky Requirements or Tasks.....	3
A. Stato dell'Arte.....	4
B. Raffinamento dei Requisiti	5
A.1 Servizi (con prioritizzazione)	5
A.2 Requisiti non Funzionali	6
A.3 Scenari d'uso dettagliati	6
A.4 Excluded Requirements	9
A.5 Assunzioni.....	9
A.6 Use Case Diagrams.....	10
C. Architettura Software	11
C.2 The dynamic view of the software architecture: Sequence Diagram	11
D. Dati e loro modellazione (se il sistema si interfaccia con un DBMS).....	12
E. Scelte Progettuali (Design Decisions).....	13
F. Progettazione di Basso Livello	15
G. Spiegare come il progetto soddisfa i requisiti funzionali (FRs) e quelli non funzionali (NFRs)	19
Appendix Prototype.....	20

List of Challenging/Risky Requirements or Tasks

Challenging Task	Date the task is identified	Date the challenge is resolved	Explanation on how the challenge has been managed
Identificare struttura, componenti, comportamenti e rappresentazione in memoria	02/12/2024	03/12/2024	Adotto il pattern creazionale Singleton per assicurarmi un'unica istanza delle classi DB e il pattern strutturale Composite per creare una gerarchia di Unità e comportamentale Observer per notificare se ci sono dei cambiamenti.
Implementare un'adeguata struttura dati per memorizzare le relazioni d'ordine	08/12/2024	08/12/2024	Sfrutto il pattern Composite che crea una gerarchia, salvandomi l'informazione della radice per poter ricavare la gerarchia ricorsivamente.
Implementare logica del salvataggio e recupero organigramma	09/12/2024	09/12/2024	Utilizzo il pattern Memento sia per le operazioni di salvataggio e recupero sia per avere la possibilità di implementare la redo e la undo.
Rappresentazione grafica della struttura e azioni per l'utente	13/12/2024	14/12/2024	Per la rappresentazione generale utilizzo JFrame, mentre per la rappresentazione dell'organigramma la libreria MXGraph.

A. Stato dell'Arte

I seguenti diagrammi sono stati d'ispirazione per la rappresentazione:

<https://app.smartdraw.com/?nsu=1>

Le funzionalità, invece, sono state racchiuse nei bottoni sottostanti al grafico.

Nella maggior parte dei casi vengono utilizzate delle lavagne grafiche per disegnare il proprio organigramma come, ad esempio, il software Microsoft Whiteboard.

B. Raffinamento dei Requisiti

A partire dai servizi minimali richiesti, raffinate la descrizione dei servizi offerti dal vostro applicativo. Descrivete anche i requisiti non funzionali.

A.1 Servizi (con prioritizzazione)

1. **Servizio 1:** dare la possibilità all'utente di poter estendere e gestire la struttura dell'organigramma. La struttura si compone di Unità e Sottounità, quest'ultime sono un'estensione delle prime. **Importanza: Alta. Complessità: Media.**
2. **Servizio 2:** per ogni unità e sottounità, l'utente deve essere in grado di indicare quali dipendenti detengono un ruolo all'interno di essa. **Importanza: Alta. Complessità: Bassa.**
3. **Servizio 3:** rimozione e modifica dell'unità/sottounità. L'utente deve essere in grado di aggiungere, rimuovere o modificare un'unità. **Importanza: Alta. Complessità: Media.**
4. **Servizio 4:** rimozione e modifica dei ruoli e dipendenti, ad esempio rimuovere un dipendente da un ruolo a lui assegnato, cambiare alcuni dei suoi dati oppure modificare un ruolo dall'unità. **Importanza: Alta. Complessità: Media.**
5. **Servizio 5:** la struttura deve essere visualizzabile in maniera adeguata affinché sia visibile la gerarchia esistente tra unità e sottounità. **Importanza: Media. Complessità: Media.**
6. **Servizio 6:** i dati presenti nell'organigramma devono essere correttamente salvati rendendo possibile la visualizzazione e la modifica dello stesso in un secondo momento. **Importanza: Media. Complessità: Bassa.**
7. **Servizio 7:** possibilità per l'utente in caso di errore di poter tornare indietro con la versione dell'organigramma. **Importanza: Media. Complessità: Media.**
8. **Servizio 8:** L'utente deve avere la possibilità di avere informazioni su una determinata unità, ovvero ruoli a lei assegnati e dipendenti appartenenti ai ruoli. **Importanza Media. Complessità Media.**

A.2 Requisiti non Funzionali

1. Efficienza nell'aggiornamento dell'organigramma a seguito di modifiche delle varie unità

A.3 Scenari d'uso dettagliati

Gli scenari di casi d'uso sono:

1. **Caricare un file:** scenario Primario, il quale necessita che il file esista e sia valido. Successivamente viene caricato e visualizzato l'organigramma scelto dall'utente.

1. Utente apre il menù per navigare nel file system
 2. Seleziona il File che vuole cercare
 3. Clicca su Open
2. **Salvataggio su file:** permette all'utente di navigare nel file system e di decidere il percorso dove salvare il file. Scenario Primario.

1. Clicca sul bottone salva e naviga sul file system
 2. Selezionare il percorso
 3. Premere Open
 4. Inserire il nome del file
3. **Inserimento di un'unità:** L'utente, cliccando sull'apposito bottone, aprirà la schermata dove potrà inserire il padre e il nome della nuova unità. Scenario Primario, il nodo viene inserito e viene aggiornata la struttura.

1. Clicca su aggiungi unità
 2. Selezionare padre
 3. Inserire nome
 4. Clicca su conferma per creare nuova unità
4. **Modifica unità:** l'utente cliccando sul bottone “modifica unità” e selezionando in seguito l'unità che si vuole modificare potrà inserire un nuovo nome che verrà modificato mantenendo la gerarchia. Scenario Primario.

1. Clicca su modifica unità
2. Seleziona unità da modificare
3. Inserisci nome
4. Clicca su conferma

5. **Rimuovi unità:** l'utente può rimuovere un'unità selezionandola. Scenario Primario. Il nodo selezionato non può essere la radice.

1. Clicca bottone rimuovi unità
2. Seleziona unità
3. Conferma

6. **Aggiungi Ruolo:** l'utente può aggiungere un ruolo selezionando l'unità a cui deve appartenere quest'ultimo e inserendone il nome. Scenario Secondario.

1. Seleziona unità
2. Inserisci nome Ruolo
3. Clicca su conferma

7. **Modifica di un ruolo:** l'utente visualizza i ruoli per unità, una volta selezionati viene inserito il nuovo nome del ruolo con il quale vogliamo rinominarlo. Scenario Secondario.

1. Clicca Modifica ruolo
2. Seleziona unità
3. Visualizza e seleziona Ruoli dell'unità
4. Clicca su conferma

8. **Rimuovi ruolo:** l'utente visualizza i ruoli per unità, una volta selezionata l'unità e il ruolo da rimuovere quest'ultimo verrà rimosso. Scenario Secondario.

1. Clicca Rimuovi ruolo
2. Seleziona unità
3. Visualizza e seleziona Ruoli dell'unità
4. Clicca su conferma

9. **Aggiungi dipendente:** l'utente può aggiungere un dipendente selezionando: ruolo, unità e compilando i campi richiesti; a seguito il dipendente verrà aggiunto. Scenario Secondario.

1. Clicca Aggiungi Dipendente
2. Seleziona unità
3. Visualizza e seleziona Ruoli
4. Compila i campi
5. Clicca conferma

10. **Modifica dipendente:** l'utente visualizza i dipendenti per ruolo e per unità, una volta selezionati entrambi e compilati i campi, verrà modificato il dipendente. Scenario Secondario.

1. Clicca su modifica Dipendente
2. Seleziona unità
3. Visualizza e seleziona i ruoli
4. Visualizza e seleziona i dipendenti
5. Clicca su conferma

11. **Rimuovi dipendente:** l'utente visualizza i dipendenti per ruolo e per unità, una volta selezionati il dipendente verrà rimosso. Scenario Secondario.

1. Clicca su rimuovi dipendente
2. Seleziona unità
3. Visualizza e seleziona ruolo
4. Visualizza e seleziona dipendente
5. Clicca su conferma

12. **Visualizza Informazioni unità:** l'utente può visualizzare sia le informazioni di un'unità tramite il bottone info sia i ruoli e i dipendenti di ognuna di esse. Scenario Primario.

1. Clicca su info
2. Seleziona unità dal menu a tendina

A.4 Excluded Requirements

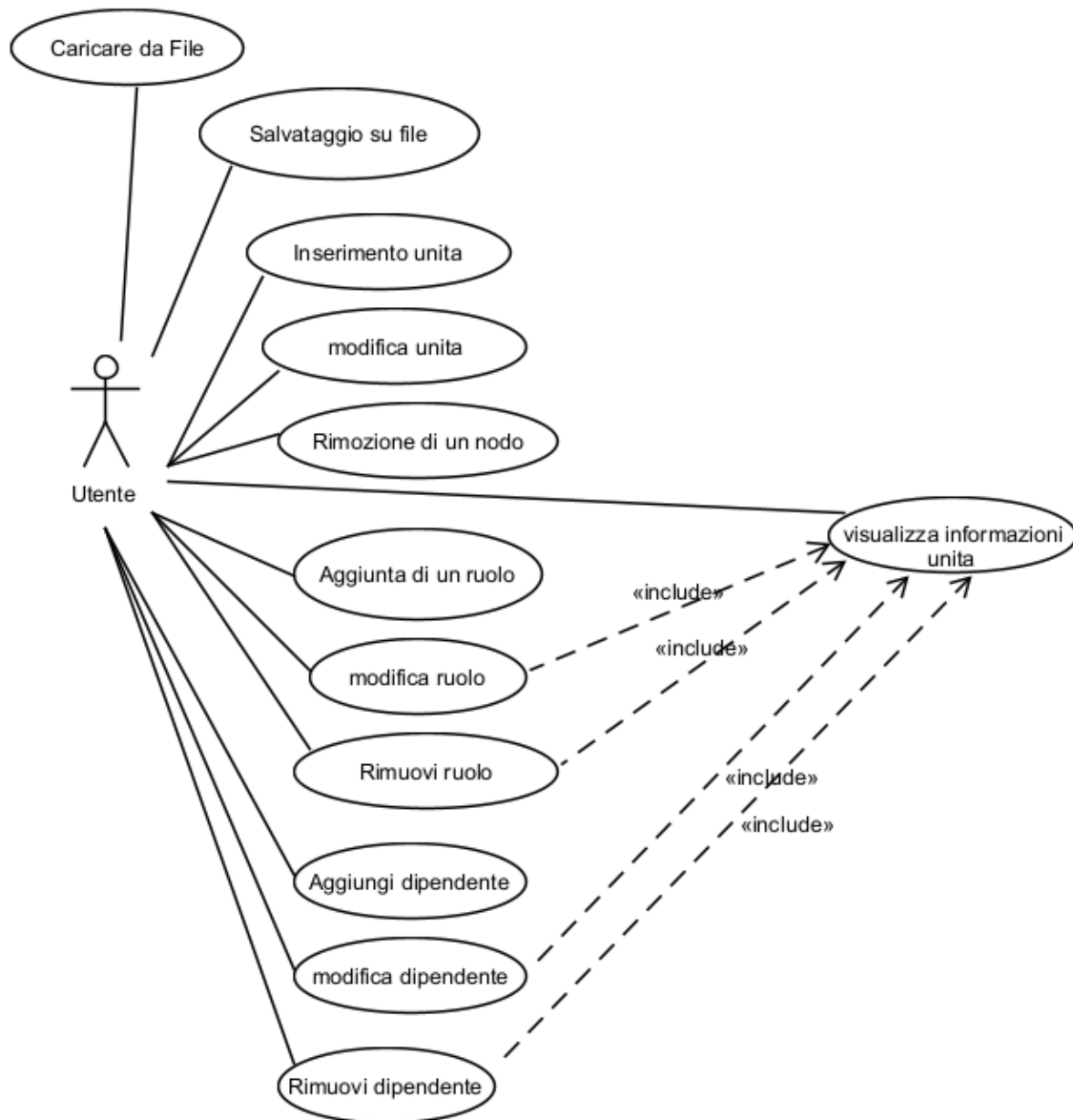
Al fine di garantire una versione funzionante e in grado di rispondere a tutti i requisiti funzionali con importanza alta, sono stati esclusi i seguenti servizi:

1. Possibilità di indicare ruoli differenti per uno stesso dipendente.
2. Possibilità di specificare per un'unità quali siano i ruoli disponibili.

A.5 Assunzioni

1. Si assume che non può esistere un organigramma senza l'unità radice.
2. Si assume che un'unità avrà sempre un padre, quindi in caso di rimozione le sottounità appartenenti ad essa verranno assegnate al padre.
3. L'albero può crescere in un'unica direzione, quella del Sistema è verso il basso

A.6 Use Case Diagrams



C. Architettura Software

C.2 The dynamic view of the software architecture: Sequence Diagram

Aggiunta unità



D. Dati e loro modellazione (se il sistema si interfaccia con un DBMS)

Gli organigrammi possono essere salvati su file, i quali possono essere riaperti successivamente dalla relativa funzione. È stato utilizzato il pattern memento per strutturare un meccanismo di salvataggio molto semplice e anche per poter implementare i meccanismi di undo e di redo, affinché l'organigramma si aggiorni ad ogni modifica dei dati. Inoltre, per la scrittura su file viene utilizzato il meccanismo di serializzazione di java, in particolare si inserisce nel file l'oggetto "originator" con l'ultima versione, il quale ha come attributi le informazioni necessarie per poter ricomporre l'organigramma. Infatti, vengono passate la lista dei dipendenti, dei ruoli, delle unità e la radice, la quale serve per poter ristampare in maniera ricorsiva la struttura dell'organigramma. I dipendenti, ruoli e unità sono salvati nelle "Hash Map", le quali hanno permesso un accesso rapido alle informazioni di cui necessitano e hanno reso più facile mantenere le relazioni dipendente-ruolo-unità.

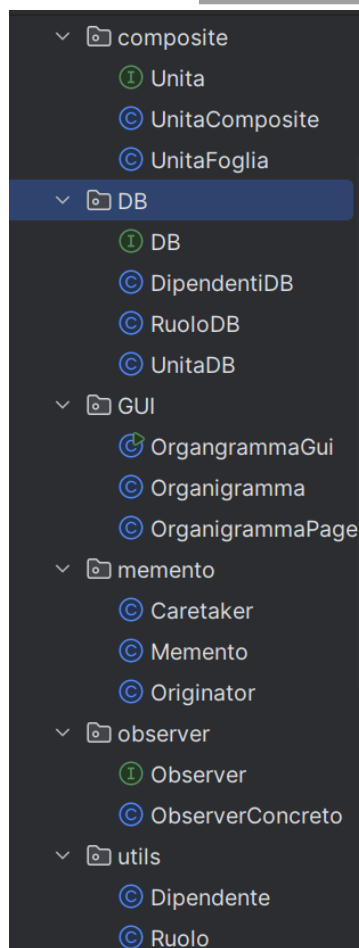
CLASS DIAGRAM



Durante la progettazione sono stati utilizzati diversi design pattern:

1. **Pattern Singleton:** utilizzato per assicurarmi un'unica istanza delle classi appartenenti al package DB, poiché avendo funzione di registro serviva la certezza che l'istanza nella quale ho effettuato determinate operazioni fosse sempre la stessa.
2. **Pattern Memento:** utilizzato per due motivi:
 1. Poter memorizzare lo stato dell'oggetto;
 2. Implementare la memorizzazione degli stati passati.
3. **Pattern Observer:** utilizzato per mantenere sempre aggiornata la mia struttura, notificando tale cambiamento ai miei registri e al mio file. In quest'ultimo caso viene notificato il cambiamento dello stato affinché venga memorizzato quello precedente.
4. **Pattern Composite:** utilizzato per far fronte alla necessità di dover rappresentare logicamente una gerarchia. Il pattern è stato estremizzato al caso in cui non ci sono classi di tipo foglia, scelta dovuta all'assunzione di voler far crescere la struttura in maniera illimitata.

F. Progettazione di Basso Livello



Qui di fianco possiamo vedere la struttura e la suddivisione del progetto. Partendo dal pattern Composite sono state sviluppate un'interfaccia e due classi concrete anche se la classe concreta foglia non viene utilizzata. L'interfaccia Unità è Serializzabile in quanto le sue istanze dovranno essere salvate in un file e perciò utilizzo la serializzazione e la de serializzazione di java.

```
public interface Unita extends Serializable { 55 usages 2 implementations ↗ Parise07

    String getNome(); 2 implementations ↗ Parise07
    List<Unita> getSottoUnita(); 5 usages 2 implementations ↗ Parise07
    void setPadre(String u); 2 usages 2 implementations ↗ Parise07
    String getPadre(); 3 usages 2 implementations ↗ Parise07
    void addRuolo(Ruolo r); 1 usage 2 implementations ↗ Parise07
    void removeRuolo(Ruolo r); 1 usage 2 implementations ↗ Parise07
    HashMap<String, Ruolo> getRuoli(); 13 usages 2 implementations ↗ Parise07
    void addDipendente(Dipendente d, Ruolo r); 1 usage 2 implementations ↗ Parise07
    void removeDipendente(Dipendente d, Ruolo r); 1 usage 2 implementations ↗ Parise07
    HashMap<String, Dipendente> getDipendenti(Ruolo r); 3 usages 2 implementations ↗ Parise07
    void copiaStato(Unita u); 1 usage 2 implementations ↗ Parise07
    void addSottoUnita(Unita u); 2 usages 2 implementations ↗ Parise07
    Ruolo getRuolo(String r); 14 usages 2 implementations ↗ Parise07
}
```

L'unità composite avrà come attributi:

1. Una serial version per verificare il cambiamento dello stato;
2. Il proprio nome che fungerà da identificativo;
3. Una mappa contenente i ruoli;

4. Una lista contenente le sottounità.

Troviamo attributi simili nelle classi “Ruolo” e “Dipendente del Package utils”.
Queste classi rappresentano gli oggetti che verranno inseriti all’interno del “DB”.

```
public class Dipendente implements Serializable { 40 usages  ↳ Parise07
    private static final long serialVersionUID = 1L; no usages
    private String nome; 2 usages
    private String cognome; 2 usages
    private String email; 2 usages
```

```
public class Ruolo implements Serializable { 54 usages  ↳ Parise07
    private static final long serialVersionUID = 1L; no usages
    private String nome; 6 usages
    private HashMap<String,Dipendente> dipendenti; 12 usages
```

Nel package DB troviamo delle classi che rappresentano i registri utilizzati nel sistema per raggruppare le varie istanze delle classi sopra indicate. Per assicurarmi di modificare l’istanza corretta di queste classi, era necessario che fosse anche l’unica ed è per questo motivo che ho applicato il pattern Singleton, il quale, basato a oggetti non permette la creazione di nuove istanze al di fuori di quella creata la prima volta.

```
public class UnitaDB implements DB<Unita>{ 10 usages  ↳ Parise07
    private static final long serialVersionUID = 1L; no usages
    private static UnitaDB unitaDB; 3 usages
    private HashMap<String,Unita> unita= new HashMap<>(); 12 usages
    private Observer osservatori; 3 usages
    private String radice; 4 usages
    private Originator file; 5 usages
```

Nel Package Observer è stato implementato il medesimo pattern, utile per notificare al sistema che sono state apportate modifiche alle strutture dati. Il metodo aggiorna, carica gli stati delle strutture dati nel file e provvede a salvare il file stesso (utile per il secondo utilizzo del pattern memento).

```
public class ObserverConcreto implements Observer{ 4 usages  ↳ Parise07
    private UnitaDB unita=UnitaDB.getInstance(); 1 usage
    private RuoloDB ruoli=RuoloDB.getInstance(); 1 usage
    private DipendentiDB dipendenti=DipendentiDB.getInstance(); 1 usage
    private Originator file; 2 usages
    private Organigramma o; 1 usage

    public ObserverConcreto(Organigramma o,Originator file){ 2 usages  ↳ Parise07
        this.file=file;
        this.o=o;
    }

    public void aggiorna(){ 3 usages  ↳ Parise07
        unita.salva();
        ruoli.salva();
        dipendenti.salva();
        file.save();
    }
}
```

Funzionalità importante del sistema è quella di poter salvare e recuperare i dati del sistema. Questo è reso possibile grazie al pattern memento, infatti caricheremo e salveremo la classe Originator.


```
public class Originator implements Serializable { 18 usages  ↳ Parise07
    private HashMap<String, Unita> unitaMap=new HashMap<>(); 5 usages
    private HashMap<String, Ruolo> ruoloMap=new HashMap<>(); 5 usages
    private HashMap<String, Dipendente> dipendentiMap=new HashMap<>(); 5 usages
    private String radice; 5 usages
    private final Caretaker c=new Caretaker(); 3 usages

    public void setStato(Originator o){ 1 usage  ↳ Parise07
        this.unitaMap=o.getUnitaMap();
        this.radice=o.getRadice();
        this.ruoloMap=o.getRuoloMap();
        this.dipendentiMap=o.getDipendentiMap();
        save();
    }
}
```

Altra utilità del memento è quella di poter creare uno storico dei vari cambiamenti dell'Originator. Infatti, la classe memento è di comunicazione tra l'originator e il Caretaker. Poiché la classe memento conosce lo stato interno dell'originator e possiede il poco che basta per ripristinare lo stato di quest'ultimo; non permette l'accesso alla struttura dati se non all'originator; mentre al Carateker conosce una piccola parte dell'interfaccia, infatti quest'ultimo è responsabile della memorizzazione del memento, non invoca operazioni e non esamina il memento.

```
import java.io.Serializable;
import java.util.LinkedList;

public class Caretaker implements Serializable { 2 usages  ↳ Parise07
    private final LinkedList<Memento> storico = new LinkedList<>(); 5 usages
    private final LinkedList<Memento> redoList = new LinkedList<>(); 4 usages

    // Salva uno stato nel Caretaker
    public void salva(Memento memento) { 1 usage  ↳ Parise07
        storico.addFirst(memento); // Inserisce in fondo alla coda
        redoList.clear(); // Dopo un nuovo salvataggio, lo stack di redo viene svuotato
        System.out.println("Stato salvato.");
    }
}
```

Per ultimo c'è il package Gui dove avremo una classe organigramma, la quale serve da collante per tutte le classi, poiché qui richiamo le principali operazioni da fare sulla struttura e sui dati, in particolare salva e carica file.

```
public void salvaOrganigramma(){ 1 usage  ↳ Parise07
    registraObserver();
    ObjectOutputStream oo;
    try{
        oo=new ObjectOutputStream(new FileOutputStream(filePath));
        System.out.println("Salvataggio: " + file);
        oo.writeObject(file);
        System.out.println("Organigramma salvato correttamente.");
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

Grazie agli stream di tipo object posso scrivere e leggere in maniera agile il file.

```
public void caricaOrganigramma() throws UsageException {
    try (ObjectInputStream oi = new ObjectInputStream(new FileInputStream(filePath))) {
        file.setStato((Originator) oi.readObject());
        observer = new ObserverConcreto(this, file);
        registraObserver();
        System.out.println("Caricamento: " + file);
        unita.setFile(file);
        ruoli.setFile(file);
        dipendenti.setFile(file);
        unita.carica();
        ruoli.carica();
        dipendenti.carica();

        System.out.println("Organigramma caricato correttamente.");
    } catch (FileNotFoundException e) {
        System.err.println("File non trovato: " + filePath);
    } catch (IOException e) {
        System.err.println("Errore durante la lettura del file: " + e.getMessage());
    } catch (ClassNotFoundException e) {
        System.err.println("Classe non trovata durante la deserializzazione: " + e.getMessage());
    }
}
```

Le classi OrganigrammaGui e OrganigrammaPage compongono la nostra interfaccia; la prima rappresenta la schermata iniziale dove sceglieremo se creare o caricare un organigramma, mentre la seconda ci permette di fare operazioni su di esso.

G. Spiegare come il progetto soddisfa i requisiti funzionali (FRs) e quelli non funzionali (NFRs)

Il Sistema è suddiviso in 6 package, 5 dei quali vertono a implementare la business logic e la gestione dell'organigramma. Si tratta dei package:

1. Composite
2. DB
3. Memento
4. Observer
5. Utils

L'insieme delle classi contenute in questi package concretizza tutti i requisiti funzionali, considerando anche la classe Organigramma nel package Gui.

La gestione dell'input e la disponibilità dell'interfaccia grafica sono invece affidate al sesto package, denominato GUI. In particolare, i requisiti non funzionali sono:

1. Verifica delle modifiche apportate dall'utente.
2. Aggiornamento della struttura e dei vari componenti grazie al pattern Observer, che richiama la libreria graph per aggiornare il grafico.

Appendix Prototype



Fig-1 All'avvio, l'applicazione presenta due bottoni. A sinistra ci permette di caricare un organigramma già esistente, sulla destra invece di crearne uno nuovo.

Partiamo con analizzare il caso in cui creo un nuovo organigramma, poiché caricandolo le operazioni successive e le schermate sono uguali.

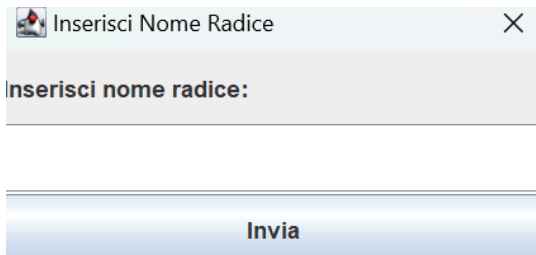


Fig-2 Cliccando sul bottone Nuovo mi si aprirà questa schermata dove mi viene richiesto di inserire la radice. Cliccando il tasto invia passeremo alla seconda schermata.

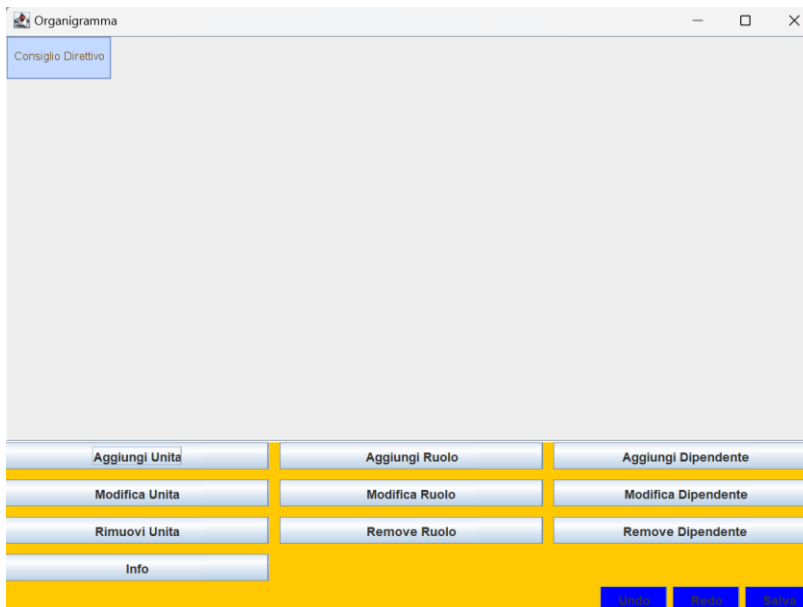


Fig-3 Questa è la schermata appena dopo la creazione della Radice. Qui possiamo trovare le operazioni sulle Unità (U), sui Ruoli (R) e sui dipendenti (D).

Fig-3.U.1 Aggiunta unità specificheremo il padre e inseriremo il nome dell'unità

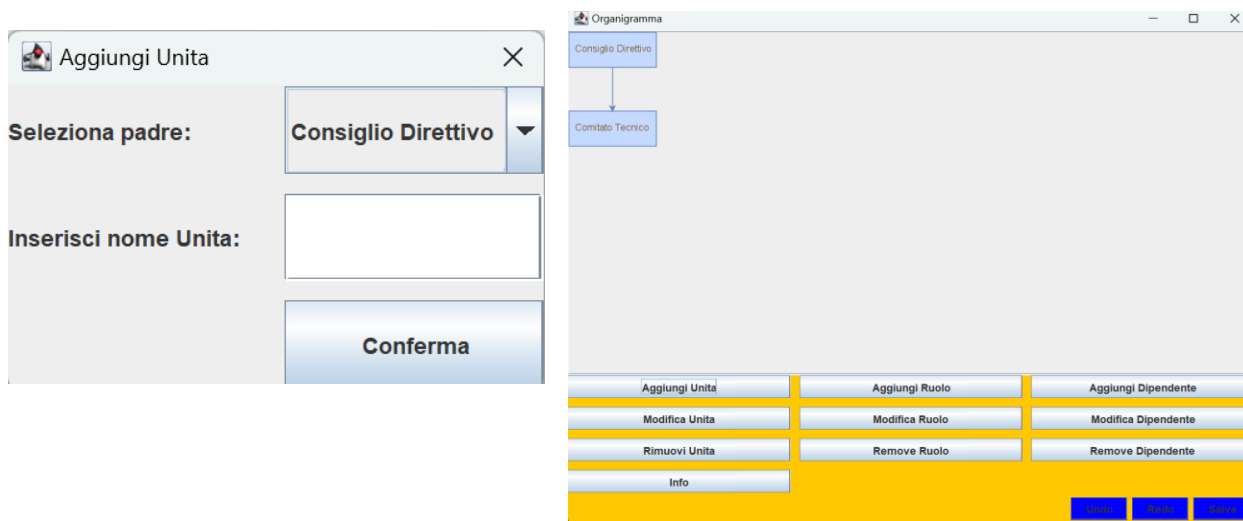


Fig-3.U.2 Modifica Unità con questo bottone modifichiamo il nome dell'unità che desideriamo, eccetto la radice.

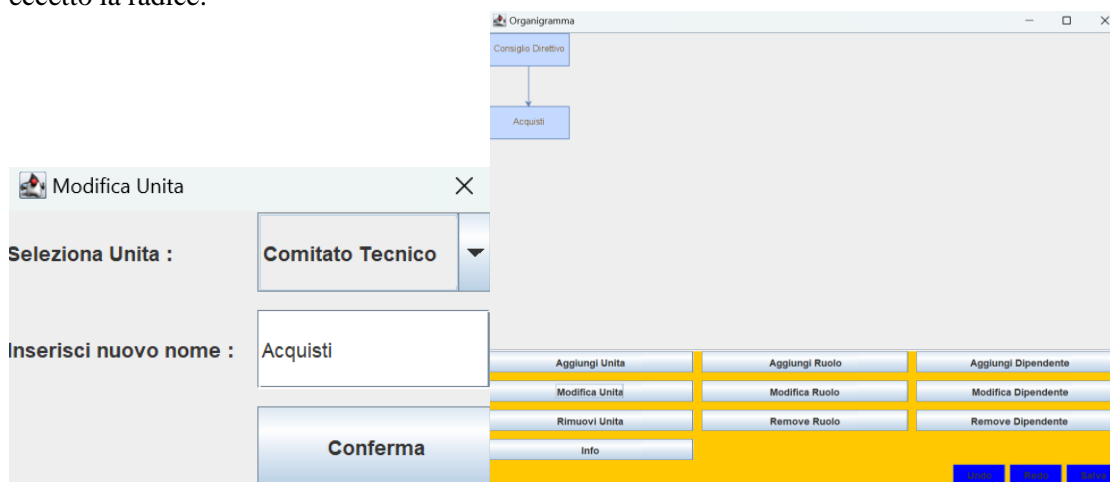
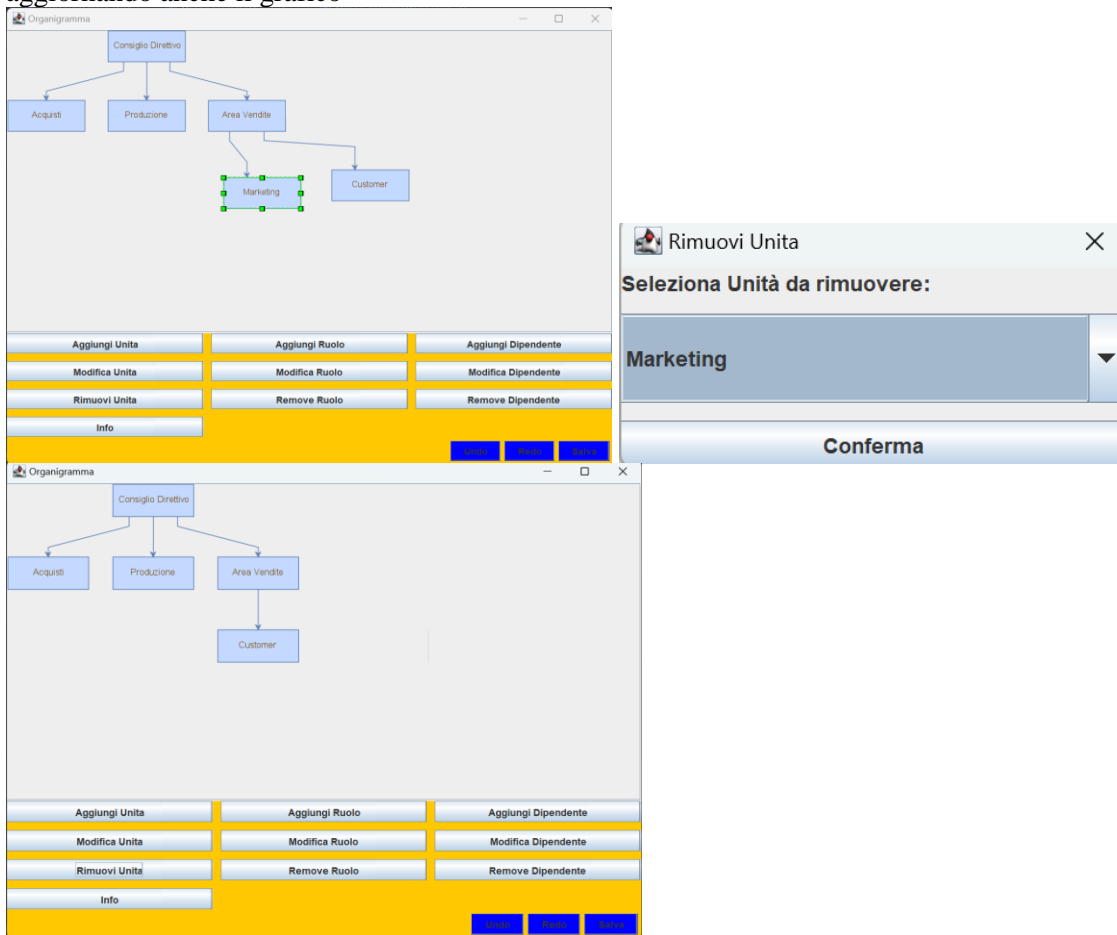


Fig-3.U.3 Rimuovi Unità selezionando l'unità che si vuole rimuovere essa verrà rimossa aggiornando anche il grafico



Gli altri bottoni per Ruoli e utente Dipendente sono molto simili ecco di seguito le loro rispettive schermate

Fig-3.R.1 Aggiunta Ruolo

The image shows the 'Aggiungi Ruolo' dialog box. It has a title bar with a close button. Inside, there is a label 'Seleziona padre:' followed by a dropdown menu showing 'Produzione'. Below this is a label 'Inserisci nome Ruolo:' followed by a text input field containing the text 'Designer'. At the bottom of the dialog is a 'Conferma' button.

Fig-3.R.2 Modifica Ruolo

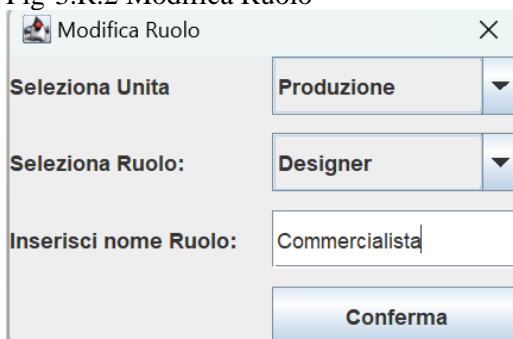


Fig-3.R.3 Rimozione Ruolo

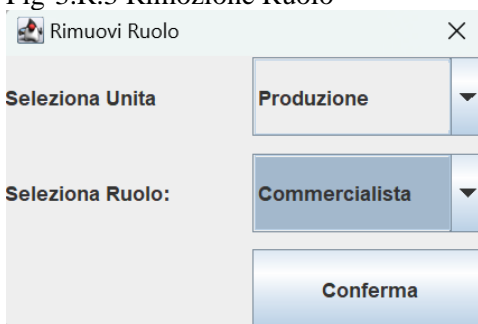


Fig-3.D.1 Aggiunta Dipendente

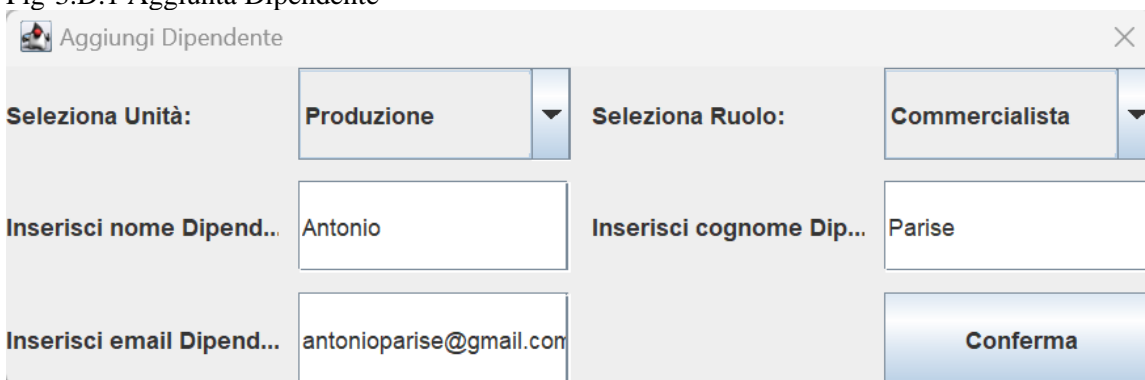


Fig-3.D.2 Modifica Dipendente

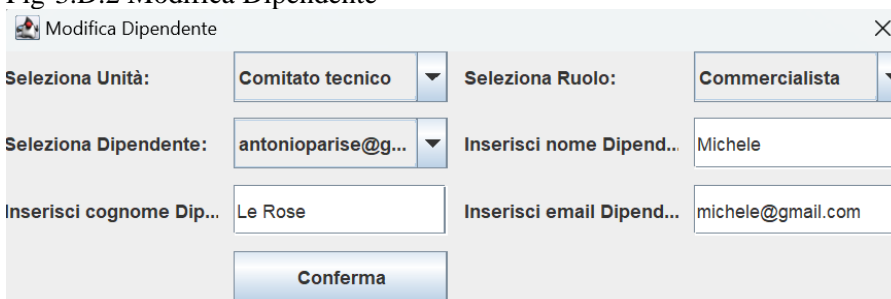


Fig-3.D.3 Rimuovi Dipendente

Rimuovi Dipendente

Seleziona Unità: **Produzione**

Seleziona Ruolo: **Commercialista**

Seleziona Dipendente (Email): **antonioparise@gmail.com**

Conferma

Fig-4 Bottone info una delle scelte progettuali, è stata la decisione di non volere visionare i dati durante la creazione dell'organigramma per questo ho inserito questa finestra che mi restituisce tutte le informazioni per ogni unità

Informazioni Unità

Seleziona padre: **Ricerca e Sviluppo**

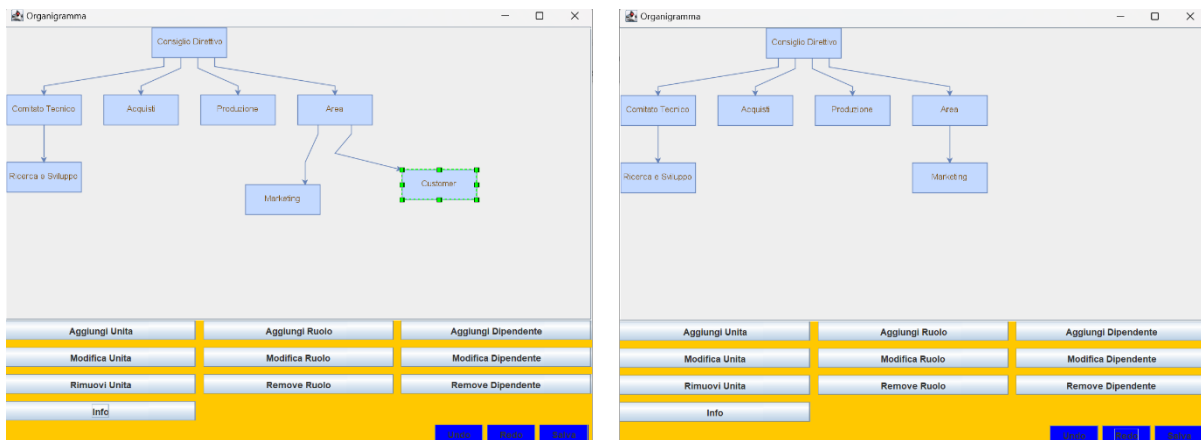
Sviluppatore:
- Michele Le Rose michele30@gmail.com

Ricerca e Sviluppo:

Chiudi

Fig-5 Bottoni Redo e undo che ci permettono di andare avanti o indietro delle versioni del nostro Organigramma.

Undo



Redo

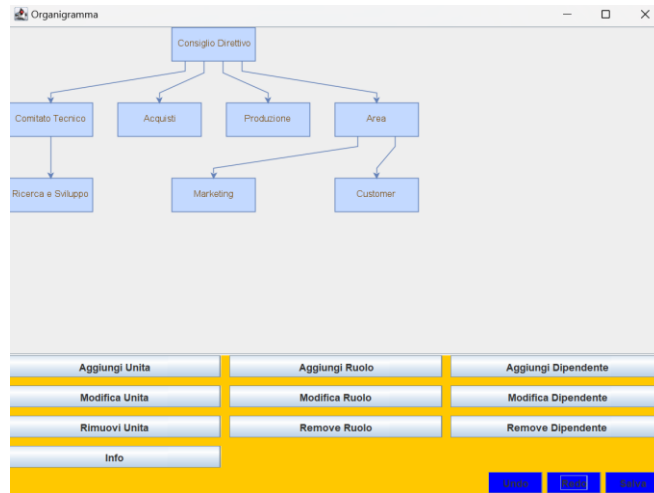


Fig-6 bottone salva questo bottone mi permette di aprire le schermate per decidere il nome del file e il percorso dove voglio salvare quest'ultimo.

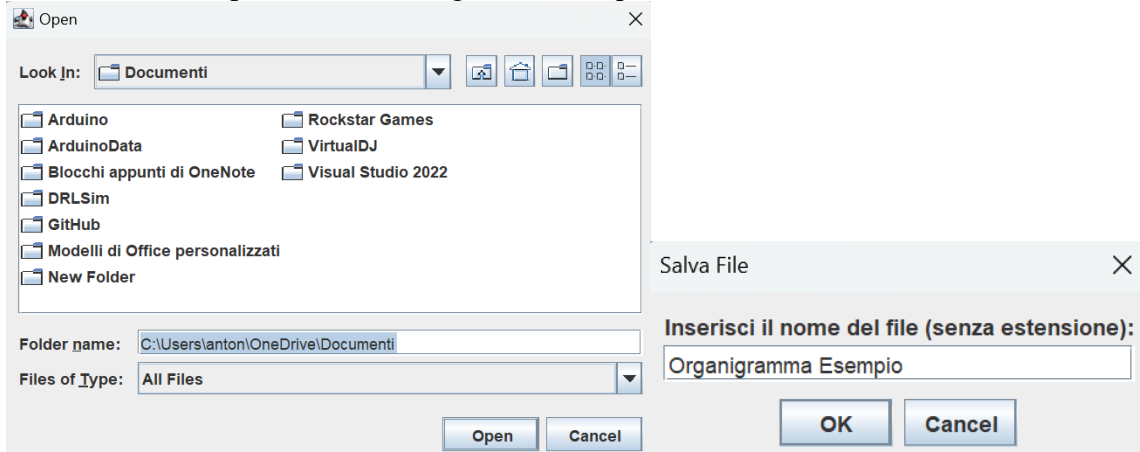


Fig-2.Bis bottone per caricare un Organigramma dalla memoria secondaria una volta fatto click su di esso ci verrà aperta la schermata di FileChooser, una volta trovato il file e premuto ok si ripartirà dalla schermata a Fig-3 ovviamente con tutti i dati precedentemente inseriti e salvati nel file.

