“Ingegneria del Software”
2020-2021

Docente: Prof. Angelo Furfaro

Organigramma Aziendale

Data	16/07/2021
Documento	Relazione progetto “Organigramma Aziendale”

Team Members		
Nome e Cognome	**Matricola**	**E-mail address**
Teodoro Sullazzo	**203803**	**teodoro.sullazzo@gmail.com**

Sommario

“Un organigramma è la rappresentazione grafica della struttura di una organizzazione in un dato momento storico” (da Wikipedia). Esso è principalmente costituito da *entità*: si tratta di organi, unità organizzative, reparti ecc…ecc…; e da *linee* che mostrano le relazioni gerarchiche tra le entità.
Un organigramma aziendale si configura perciò come una struttura gerarchica.
L’organigramma mostra, oltre alle varie entità, le relazioni che sussistono tra di esse e i ruoli svolti dai dipendenti dell’organizzazione nelle entità. Un dipendente può far parte di una o più entità ed in ognuna di esse svolge un certo ruolo; può comparire più volte nella medesima entità, ma con ruoli differenti.
Non tutti i ruoli sono occupabili per una certa entità, ovvero, esistono ruoli disponibili solo per certe unità.

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
Modellare gli elementi dell'organigramma in Java	**11/06/2021**	**12/06/2021**	**Si è scelto di considerare gli elementi principali di un organigramma: unità organizzative, aziende, dipendenti e ruoli. Ognuno di questi elementi è stato modellato usando una classe**
Dare una rappresentazione grafica alla struttura dell'organigramma	**11/06/2021**	**13/06/2021**	**Si è deciso di utilizzare, per rappresentare visivamente la struttura dell'organigramma, il JTree di javax.swing. Per rappresentare il contenuto in termini di ruoli possibili e di ruoli occupati dai dipendenti si è usata una semplice JTextArea**
Offrire un'interfaccia grafica per la manipolazione degli organigrammi	**11/06/2021**	**12/06/2021**	**Il task è stato realizzato utilizzando una serie di JButton o JMenuItem. La struttura del pannello dei comandi si è evoluta nel tempo con l'aggiunta delle nuove funzionalità richieste.**
Permettere la comunicazione tra il pannello dei comandi e la rappresentazione del modello dell'organigramma	**13/06/2021**	**13/06/2021**	**Si è deciso di risolvere questo problema ricorrendo al design pattern MVC (argomento che sarà più ampiamente discusso in seguito)**
Utilizzare un dbms come struttura di supporto all'applicazione	**13/06/2021**	**20/06/2021**	**Utilizzando il framework di jdbc si è stati in grado di far interfacciare l'applicazione con un dbms (si è usato MySql). È stato necessario ripensare al modello dei dati in chiave di comunicazione con un database.**
Permettere all'utente di inserire l'anagrafica relativa ad un nuovo dipendente	**20/06/2021**	**27/06/2021**	**Si è aggiunto un nuovo comando sul controller in grado di dare la possibilità al client compilare i dati relativi ad un dipendente (nome, cognome ecc…)**

Ottimizzazione delle operazioni sul dbms	**20/06/2021**	**27/06/2021**	**Si è deciso di modificare il modo in cui le operazioni venivano eseguite sul dbms, prediligendo un numero minore di statement eseguiti (le singole operazioni sono state condensate in dei batch)**
Aggiungere la possibilità di salvare e caricare file in formato Json (il file contiene i dati del database)	**27/06/2021**	**28/06/2021**	**Con l'utilizzo delle classi contenute nel framework org.json.simple si è potuto gestire il caricamento da e per file .json**

A. Stato dell'Arte

Per la realizzazione di questa applicazione è stato preso in considerazione il tool messo a disposizione da Microsoft, in particolar modo, si è preso spunto da Power Point; qui è infatti possibile realizzare degli organigrammi aziendali. L'applicativo di Microsoft utilizza una struttura ad albero per la rappresentazione sintetica degli organigrammi, posta a lato di quella grafica.
Questa idea è stata utilizzata per la realizzazione della parte relativa alla rappresentazione visiva dell'organigramma.

B. Raffinamento dei Requisiti

I requisiti richiesti in modo informale consistono nel: poter aggiornare dinamicamente la struttura di una determinata organizzazione, in particolar modo, poter definire delle unità organizzative, sottounità e/o organi di gestione organizzati gerarchicamente; l'applicazione deve permettere di definire i ruoli occupati dai vari dipendenti e di stabilire quali sono i ruoli possibili per ciascuna unità organizzativa. Tali ruoli, come anticipato, possono avere senso per alcune unità e non per altre. Ogni dipendente deve essere associato ad uno o più ruoli in determinate unità organizzative. L'applicazione deve inoltre consentire di memorizzare su memoria secondaria gli organigrammi che gestisce.

Possiamo quindi passare, dopo la descrizione informale dei requisiti, ad un raffinamento di essi.
Poiché è richiesto di poter aggiornare dinamicamente la struttura di una determinata organizzazione deve essere possibile:

- Aggiungere unità (gerarchicamente sottoposte ad entità madri)
- Rimuovere unità
- Cambiare il nome delle unità

Poiché dobbiamo poter definire dei ruoli e associarli a dei dipendenti dobbiamo essere in grado di:

- Aggiungere un ruolo ad un dipendente, in relazione ad una data entità
- Rimuovere un ruolo ad un dipendente, in relazione ad una data entità

Poiché inoltre, alcuni ruoli hanno senso in alcune unità ed in altre no, dobbiamo essere in grado di:

- Aggiungere un ruolo ammissibile ad una unità
- Rimuovere un ruolo ammissibile ad una unità

Con l'aggiunta di queste due ultime funzionalità, ovviamente, potremo aggiungere un ruolo ad un dipendente in relazione ad una determinata unità solo se questa presenta, nella lista dei ruoli ammessi, quel ruolo che si desidera assegnare al dipendente.

Dobbiamo, essere in grado di poter:

- salvare gli organigrammi delle aziende create su dispositivo di memorizzazione secondaria.

In più, potrebbe essere utile gestire l'anagrafica dei dipendenti dall'applicazione; deve quindi essere possibile:

- creare un nuovo dipendente

Deve anche essere possibile gestire dinamicamente l'aggiunta dei ruoli, deve quindi essere possibile:

- creare nuovi ruoli

Il tutto deve essere reso disponibile all'utente tramite interfaccia grafica.

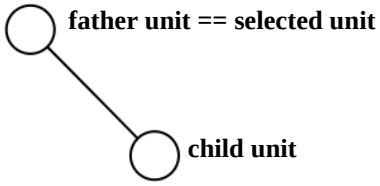
A.1 Servizi (con prioritizzazione)

L'applicazione che andremo a realizzare, verosimilmente, dovrà essere utilizzata all'interno di una azienda.
Possiamo quindi supporre che l'applicativo verrà utilizzato anche per la registrazione dei dipendenti, e al contempo, sempre dall'applicazione, anche per la registrazione di nuovi ruoli. L'applicativo deve quindi essere molto flessibile nel suo utilizzo: non è necessario conoscere a priori né i dipendenti, né i ruoli che all'interno dell'azienda verranno definiti, in quanto questi possono essere introdotti dinamicamente dall'applicazione.
È possibile anche supporre che, nell'uso dell'applicazione, sia necessario interfacciarsi con un dbms, in grado di memorizzare i dati degli organigrammi, e che funga da supporto, in particolar modo, anche nel salvataggio su file e caricamento.

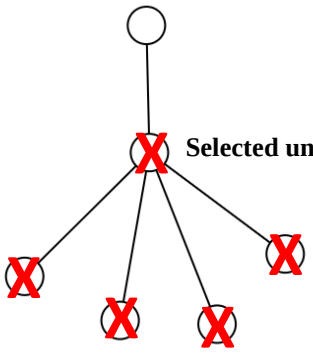
Risulta evidente che tra i servizi definiti in precedenza, esistono aspetti più prioritari, ed altri meno rilevanti.

1. **Aggiunta di nuove unità [priorità: alta] [complessità: media]**
2. **Rimozione di unità esistenti [priorità: alta] [complessità: media]**

Il cuore dell'applicazione sta proprio nella possibilità di poter aggiungere e rimuovere nuove unità. Questo deve essere effettuato scegliendo, prima di effettuare l'aggiunta, l'entità madre, ovvero, l'entità alla quale quella creata sarà legata da un rapporto di subordinazione.



Nella rimozione di un nodo invece, dopo averlo selezionato, verrà eliminato non solo quel nodo, ma anche tutti i figli di quest'ultimo.



3. **Aggiornamento nome unità [priorità: alta] [complessità: bassa]**

È importante dare agli utenti la possibilità di cambiare il nome alle unità create.

4. **Aggiunta ruolo ammissibile ad un dipendente, relativamente ad un'unità [priorità: media] [complessità: bassa]**
5. **Rimozione ruolo ammissibile ad un dipendente, relativamente ad un'unità [priorità: media] [complessità: bassa]**

Sebbene queste ultime due specifiche siano state definite sin dall'inizio, si potrebbe pensare di implementarle anche in una versione successiva dell'applicativo, essendo queste non strettamente legate agli elementi fondamentali dell'organigramma (i.e. *unità* e *linee*). L'applicazione, tuttavia, non può ovviamente dirsi completa senza tali servizi.

6. **Aggiungere un ruolo possibile ad una unità [priorità: media] [complessità: bassa]**
7. **Rimuovere un ruolo possibile ad una unità [priorità: media] [complessità: bassa]**

Per le medesime ragioni di prima, possiamo supporre di fornire al committente, almeno in una prima istanza, l'applicativo con i soli servizi relativi alla gestione e nomenclatura delle unità. I servizi relativi all'anagrafica e all'assegnamento dei ruoli possono essere eseguiti in un secondo momento.

Ognuna delle nostre unità avrà quindi, ad essa associata, un insieme di ruoli ammissibili e ruoli assegnati ai dipendenti. Per aggiungere un ruolo ammissibile basterà aggiungere a questo insieme il nuovo ruolo, per eliminarlo basterà toglierlo.
L'aggiunta di un dipendente che assolve ad un certo ruolo deve essere effettuata controllando che il ruolo in considerazione faccia parte dell'insieme dei ruoli ammissibili.

8. **Salvataggio su file [priorità: bassa] [complessità: alta]**

Il salvataggio ci permette di rendere persistenti nel tempo i dati dei nostri organigrammi, ciò non è strettamente necessario al funzionamento dell'applicativo, potrebbe però risultare molto utile nella pratica.

9. **Comunicazione con dbms [priorità: media] [complessità: alta]**

Sarà necessario far interfacciare l'applicazione con un database, in modo tale da poter inserire i dati ottenuti dall'utilizzo dell'applicazione, ma anche per poter riusare dati già presenti su eventuali database disponibili all'interno dell'azienda (soprattutto per quanto concerne le anagrafiche dei dipendenti). Grazie all'uso del database si può supportare anche il salvataggio su file (usando ad esempio un formato come il Json).

Possiamo quindi espandere il punto 9.

9.1 Caricare un organigramma dal database [priorità: media] [complessità: alta]

9.2 Salvare un organigramma sul database [priorità: media] [complessità: alta]

Il nostro applicativo gestisce un organigramma alla volta, ma, inserendo la funzionalità relativa al salvataggio sul database, possiamo essere in grado di selezionare e caricare i singoli organigrammi presenti sul db dinamicamente. Il caricamento e salvataggio degli organigrammi su db, ovviamente, necessita anche il caricamento e salvataggio dei ruoli e dei dipendenti utilizzati.

9.3 Caricare ruoli e dipendenti dal database [priorità: bassa] [complessità: media]

9.4 Salvare ruoli e dipendenti sul database [priorità: bassa] [complessità: media]

Queste due ultime funzionalità, in realtà, sono implicitamente realizzate nella 9.1 e 9.2. Viene introdotta, in particolar modo, la possibilità di caricare e salvare i dipendenti ed i ruoli singolarmente in quanto all'utente potrebbe risultare utile gestire separatamente la parte relativa all'anagrafica e ai ruoli, rispetto agli organigrammi in toto.

10. Creazione di nuovi organigrammi [priorità: media] [complessità: media]

Sebbene nelle specifiche informali non venga specificato, potrebbe essere utile offrire all'utente la possibilità di creare un nuovo organigramma in modo dinamico.

Come già detto deve essere possibile aggiungere dinamicamente nuovi dipendenti.

11. Creare nuovi dipendenti [priorità: media] [complessità: media]

Deve inoltre essere possibile introdurre nuovi ruoli

12. Definire nuovi ruoli [priorità: media] [complessità: bassa]

A.2 Requisiti non Funzionali

Poiché l'applicazione verrà probabilmente utilizzata in un ambiente aziendale è necessario che il suo utilizzo risulti comprensibili anche per un utente distante dal dominio informatico. Abbiamo quindi bisogno di un'interfaccia grafica user-friendly, che faccia intuire facilmente all'utente il funzionamento dell'applicazione (questo problema potrebbe essere risolto parzialmente fornendo insieme al sistema un manuale d'uso).
Inoltre, è necessario ottenere delle prestazioni abbastanza elevate, per evitare dei ritardi che danneggino l'esperienza del cliente.
È anche possibile supporre che un requisito non funzionale per l'applicazione sia l'affidabilità, per quanto riguarda la gestione dei dati; ovvero, poter garantire all'utente che, oltre alla possibilità di rendere persistenti gli organigrammi tramite il salvataggio su file, questi siano disponibili all'interno di un database, per poter anche, eventualmente, utilizzarli per fini che vanno al di là della nostra applicazione.

A.3 Scenari d'uso dettagliati

I principali casi d'uso che si possono incontrare sono i seguenti:

- L'utente crea una nuova unità
- L'utente elimina una unità esistente
- L'utente cambia il nome ad un'unità
- L'utente crea un nuovo organigramma
- L'utente aggiunge un nuovo ruolo
- L'utente aggiunge un ruolo ammissibile ad una unità
- L'utente rimuove un ruolo ammissibile ad una unità
- L'utente assegna un ruolo, tra quelli ammissibili di quella unità, ad un dipendente
- L'utente rimuove un ruolo ad un dipendente in riferimento ad una certa unità
- L'utente carica da file un insieme di organigrammi
- L'utente salva su file un insieme di organigrammi
- L'utente carica su database un organigramma
- L'utente salva su database un organigramma
- L'utente carica dal database ruoli e dipendenti
- L'utente salva su database ruoli e dipendenti
- L'utente crea un nuovo dipendente
- L'utente aggiunge un nuovo ruolo

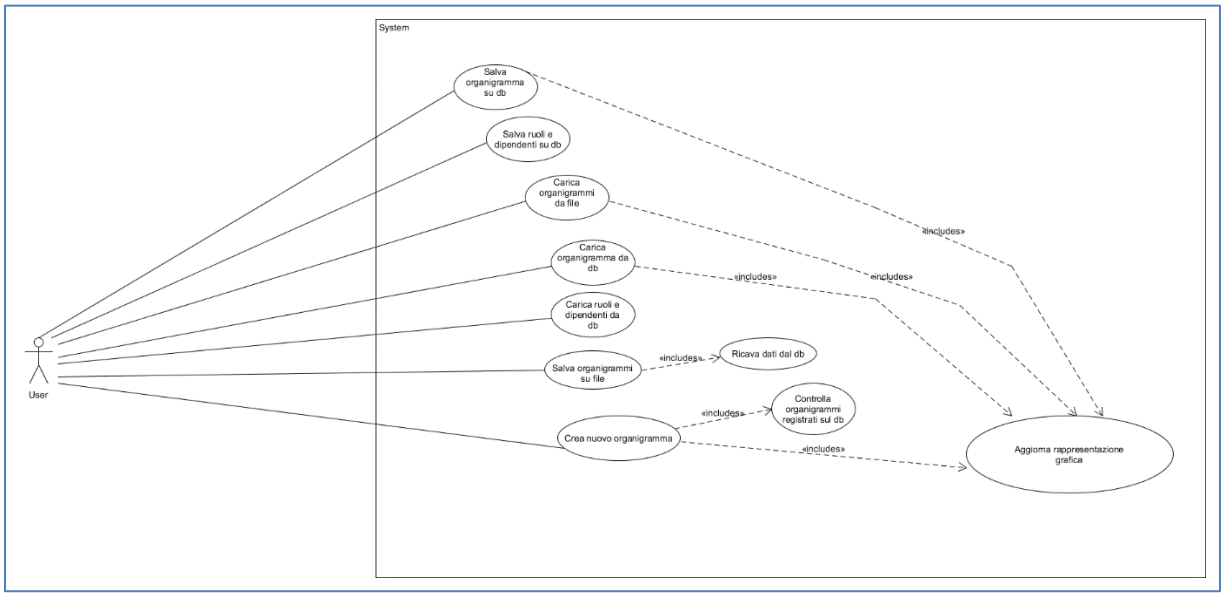
A.4 Excluded Requirements

Dei requisiti aggiungibili potrebbero essere:
gestire la comunicazione col db in modo tale che le modifiche siano eseguite in real-time. La realizzazione di un sistema del genere richiederebbe l'implementazione di un'applicazione basata sul design pattern "repository". Per la realizzazione di questo progetto si è preferito optare per una soluzione basata sulla creazione di un meccanismo di traduzione CRUD, di più semplice realizzazione. La creazione del sistema basato su "repository" potrebbe essere implementata in una successiva versione dell'applicazione. Per quanto si sia cercato di raggiungere una certa efficienza nell'esecuzione delle operazioni sull'applicazione, la priorità nella realizzazione del progetto è stata data al soddisfacimento delle specifiche sopra riportate, specialmente quelle funzionali. Un'operazione di ottimizzazione generale, almeno per quanto concerne una versione successiva del sistema è quindi possibile.

A.5 Assunzioni

Le assunzioni più importanti effettuate durante la realizzazione del progetto sono relative alla rappresentazione dei dati scelta per modellare l'organigramma. Ad esempio, in assenza di ulteriori informazioni, si è supposto che un dipendente sia caratterizzato, nell'azienda: da un nome, cognome, e-mail e identificato tramite un "sn".

.6 Use Case Diagrams



Caso d'uso	Nuova unità
Precondizione	L'utente è in grado di selezionare un'unità dell'organigramma corrente
Svolgimento normale	1. Viene selezionata un'unità 2. Una nuova unità viene aggiunta come unità figlia di quella selezionata 3. Aggiorna rappresentazione grafica
Svolgimento alternativo	Nel caso in cui l'utente non specifica l'unità madre l'aggiunta non può avvenire, l'esito dell'operazione è negativo
Postcondizione	La rappresentazione dell'organigramma riflette i cambiamenti effettuati
Descrizione	L'utente vuole aggiungere una nuova unità all'organigramma. Seleziona l'unità madre e poi aggiunge la figlia

Caso d'uso	Elimina unità
Precondizione	L'utente deve essere in grado di selezionare almeno un'unità (diversa dalla radice dell'organigramma)
Svolgimento normale	1. L'utente seleziona l'unità da eliminare 2. L'unita selezionata viene eliminata 3. Vengono eliminati tutti i ruoli assegnati ai dipendenti, relativi all'unità rimossa 4. Vengono eliminati tutti i ruoli ammessi, relativi all'unità rimossa 5. Aggiorna rappresentazione grafica
Svolgimento alternativo	L'utente non seleziona alcuna unità, l'operazione fallisce
Svolgimento alternativo	L'unità selezionata è la radice dell'organigramma: l'operazione di rimozione fallisce
Postcondizione	La rappresentazione dell'organigramma riflette i cambiamenti effettuati
Descrizione	L'utente che desidera eliminare un'unità la seleziona e poi sceglie di eliminarla

Caso d'uso	Cambia nome ad unità
Precondizione	L'utente deve essere in grado di selezionare almeno un'unità nell'organigramma corrente
Svolgimento normale	1. L'utente seleziona l'unità di cui vuole cambiare il nome 2. L'utente inserisce il nuovo nome dell'unità 3. Il nome dell'unità viene cambiato con quello inserito 4. Aggiorna rappresentazione grafica
Svolgimento alternativo	Se l'utente inserisce una stringa non valida il nome non viene aggiornato
Postcondizione	L'unita mostra il nome aggiornato
Descrizione	L'utente richiede di cambiare il nome dell'unità

Caso d'uso	Crea nuovo organigramma
Svolgimento normale	1. L'utente inserisce il nome del nuovo organigramma 2. Controlla organigrammi registrati sul db 3. Il nuovo organigramma viene creato 4. Aggiorna rappresentazione grafica
Svolgimento alternativo	Il nome dell'organigramma è già presente sul db, l'operazione fallisce
Postcondizione	L'applicazione mostra l'organigramma corrente
Descrizione	L'utente desidera creare un nuovo organigramma; inserisce il nome e richiede al sistema di crearlo. Se questo nome è però già in uso, l'organigramma non può essere creato, l'operazione fallisce; altrimenti, l'organigramma è creato e visualizzato

Caso d'uso	Aggiungi nuovo ruolo
Svolgimento normale	1. L'utente inserisce il nome del nuovo ruolo 2. Controlla ruoli esistenti 3. Il nuovo ruolo viene aggiunto 4. Aggiorna rappresentazione grafica

Svolgimento alternativo	L'utente inserisce un nome non valido o già presente, il ruolo non è aggiunto
Postcondizione	La lista dei ruoli è aggiornata
Descrizione	L'utente desidera aggiungere un nuovo ruolo, se questo nuovo nome è già presente nella lista dei ruoli, o se la stringa non è valida, l'aggiunta non può avvenire. Nel caso contrario, il nuovo ruolo viene aggiunto e mostrato.

Caso d'uso	**Aggiungi ruolo ammissibile ad unità**
Precondizione	L'utente deve poter selezionare almeno un ruolo e un'unità
Svolgimento normale	1. L'utente seleziona un ruolo tra quelli disponibili 2. L'utente seleziona un'unità 3. Controlla ruoli ammissibili sull'unità 4. Il nuovo ruolo ammissibile viene aggiunto all'unità 5. Aggiorna rappresentazione grafica
Svolgimento alternativo	Se l'utente non seleziona alcun ruolo o alcuna unità, o se il ruolo è già presente nell'unità, l'aggiunta non può avvenire
Postcondizione	La lista dei ruoli ammissibili sull'unità è aggiornata
Descrizione	L'utente desidera aggiungere un nuovo ruolo ammissibile ad un'unità. Per fare ciò seleziona l'unità e il ruolo e poi esegue l'operazione. Se non seleziona alcuna unità o ruolo l'operazione fallisce. Se il ruolo è già presente nei ruoli ammissibili dell'unità l'operazione non ha alcun effetto

Caso d'uso	**Rimuovi ruolo ammissibile ad unità**
Precondizione	L'utente deve poter selezionare almeno un ruolo e un'unità
Svolgimento normale	1. L'utente seleziona un'unità 2. L'utente seleziona un ruolo 3. Il ruolo ammissibile viene eliminato dall'unità 4. Aggiorna rappresentazione grafica

Svolgimento alternativo	L'utente non seleziona alcuna unità o alcun ruolo, oppure, il ruolo non è presente nell'insieme dei ruoli ammissibili; in tal caso nessun cambiamento viene effettuato
Postcondizione	I ruoli ammissibili dell'unità selezionata sono aggiornati
Descrizione	L'utente desidera rimuovere un ruolo ammissibile, seleziona il ruolo, l'unità e poi esegue l'operazione. Se nessun ruolo o unità è selezionata l'operazione non può essere eseguita

Caso d'uso	**Assegna ruolo a dipendente**
Precondizione	L'utente deve essere in grado di selezionare almeno un'unità, un ruolo e un dipendente
Svolgimento normale	1. L'utente seleziona un'unità 2. L'utente seleziona un dipendente 3. L'utente seleziona un ruolo 4. Controlla ruoli ammissibili sull'unità 5. Controlla i ruoli assegnati ai dipendenti nell'unità 6. Aggiungi il ruolo al dipendente in relazione all'unità 7. Aggiorna rappresentazione grafica
Svolgimento alternativo	L'utente seleziona un ruolo non appartenente a quelli ammissibili sull'unità, oppure, non seleziona alcuna unità, ruolo, o dipendente, oppure, tale ruolo è stato già assegnato al dipendente in quell'unità, in questi casi l'operazione non può essere eseguita
Postcondizione	I ruoli assegnati sull'unità sono aggiornati

Caso d'uso	**Rimuovi ruolo a dipendente**
Precondizione	L'utente deve essere in grado di selezionare almeno un'unità, un ruolo e un dipendente
Svolgimento normale	1. L'utente seleziona un'unità 2. L'utente seleziona un dipendente

	3. L'utente seleziona un ruolo 4. Rimuovi il ruolo assegnato al dipendente nell'unità 5. Aggiorna rappresentazione grafica
Svolgimento alternativo	L'utente non seleziona alcun ruolo, unità o dipendente, oppure, l'assegnamento non è presente nella lista dei ruoli occupati nell'unità, in tal caso l'operazione non viene eseguita
Postcondizione	I ruoli assegnati sull'unità sono aggiornati

Caso d'uso	**Salva su file**
Precondizione	L'utente desidera salvare il contenuto del database (che contiene l'insieme degli organigrammi salvati in esso) su file
Svolgimento normale	1. L'utente inserisce il nome del file 2. Ricava dati dal db 3. Il contenuto del database viene salvato sul file
Svolgimento alternativo	Il salvataggio sul file non avviene correttamente, l'operazione non viene eseguita
Postcondizione	Il file col nome scelto è creato e contiene al suo interno il contenuto del database

Caso d'uso	**Carica organigrammi da file**
Precondizione	L'utente deve essere in grado di selezionare un file
Svolgimento normale	1. L'utente seleziona il file 2. Il contenuto attuale del database viene rimosso 3. Il contenuto del file viene caricato sul database 4. La lista degli organigrammi disponibili sul database viene aggiornata 5. Aggiorna rappresentazione grafica
Svolgimento alternativo	In caso di file corrotto o di errore nel caricamento lo stato precedente del database viene ripristinato

Postcondizione	Il database contiene adesso l'insieme degli organigrammi presenti sul file

Caso d'uso	**Carica organigramma da db**
Precondizione	L'utente deve essere in grado di selezionare almeno uno degli organigrammi salvati sul db
Svolgimento normale	1. L'utente seleziona l'organigramma che desidera caricare dal db 2. L'organigramma corrente viene rimosso 3. L'organigramma selezionato viene caricato dal db 4. Aggiorna rappresentazione grafica
Postcondizione	L'utente visualizza adesso l'organigramma caricato dal database

Caso d'uso	**Salva organigramma su db**
Precondizione	L'utente ha, nella sessione corrente, un organigramma
Svolgimento normale	1. L'organigramma corrente viene salvato sul database 2. Aggiorna rappresentazione grafica
Svolgimento alternativo	Se l'organigramma che si sta cercando di salvare è già presente sul db, allora questo viene semplicemente aggiornato, la lista degli organigrammi non viene quindi modificata
Postcondizione	La lista degli organigrammi disponibili sul database è aggiornata

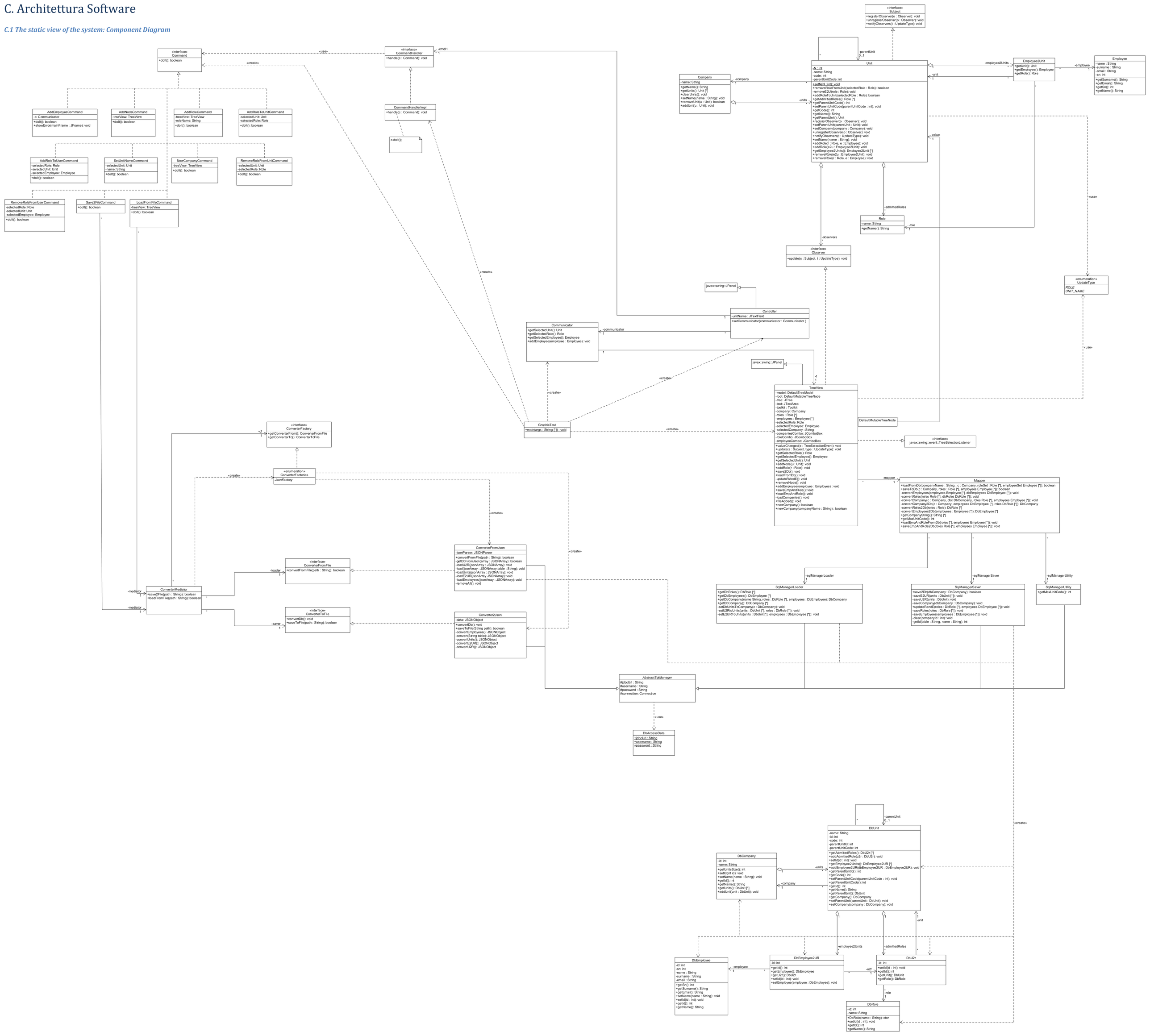
Caso d'uso	**Carica ruoli e dipendenti da database**
Precondizione	L'utente desidera caricare i dipendenti e i ruoli presenti sul database
Svolgimento normale	1. I clienti e i ruoli sono caricati dal database
Postcondizione	La lista dei dipendenti e dei ruoli è aggiornata con le informazioni prese dal db

Caso d'uso	Salva dipendenti e ruoli su database
Precondizione	L'utente desidera salvare i dipendenti e i ruoli disponibili sulla sessione attuale sul database
Svolgimento normale	1. I dipendenti e i ruoli sono salvati sul database
Postcondizione	Il database viene aggiornato con i nuovi dipendenti e ruoli

Caso d'uso	Nuovo dipendente
Precondizione	L'utente deve essere in grado di inserire i dati del nuovo dipendente
Svolgimento normale	1. L'utente inserisce i dati del dipendente (nome, cognome, e-mail, identificativo…) 2. Controlla dipendenti esistenti 3. Il nuovo dipendente viene aggiunto 4. Aggiorna rappresentazione grafica
Svolgimento alternativo	Se il dipendente è già presente all'interno della lista dei dipendenti disponibili, allora l'aggiunta non può avvenire.
Postcondizione	La lista dei dipendenti viene aggiornata

C. Architettura Software

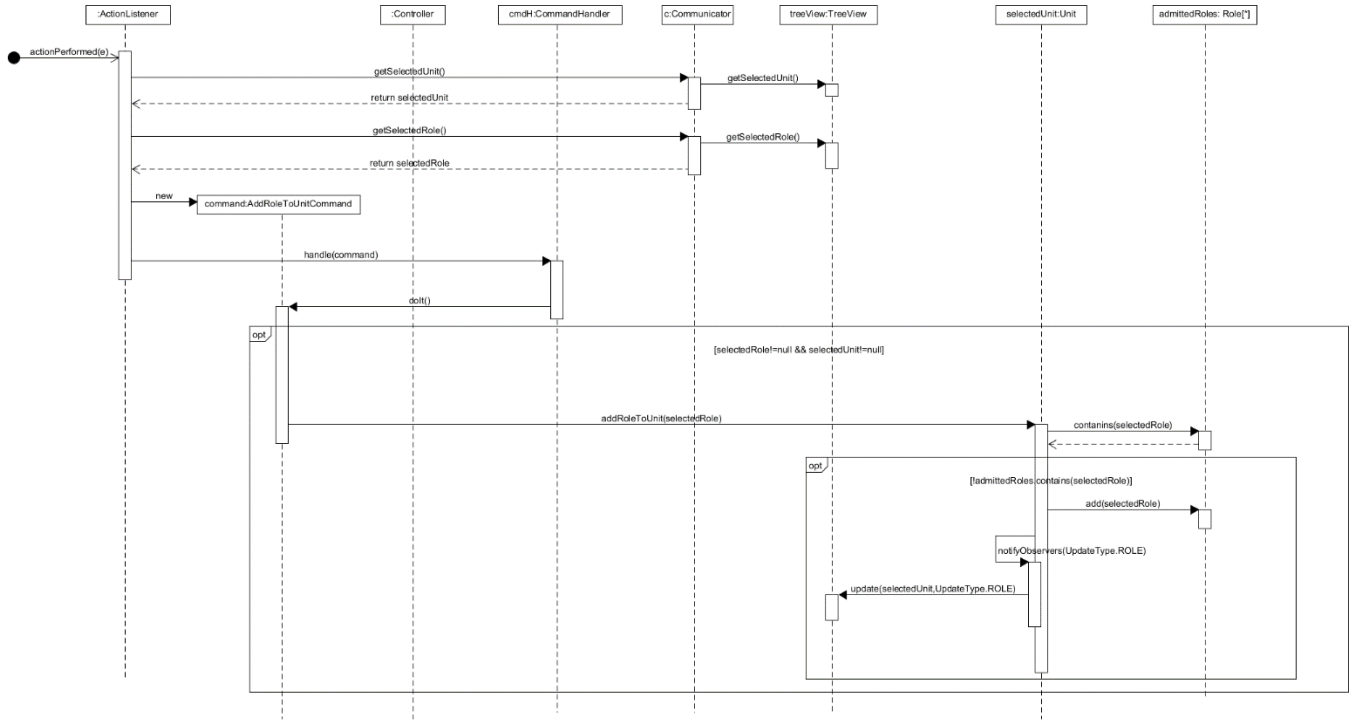
C.1 The static view of the system: Component Diagram



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

-Add role to unit

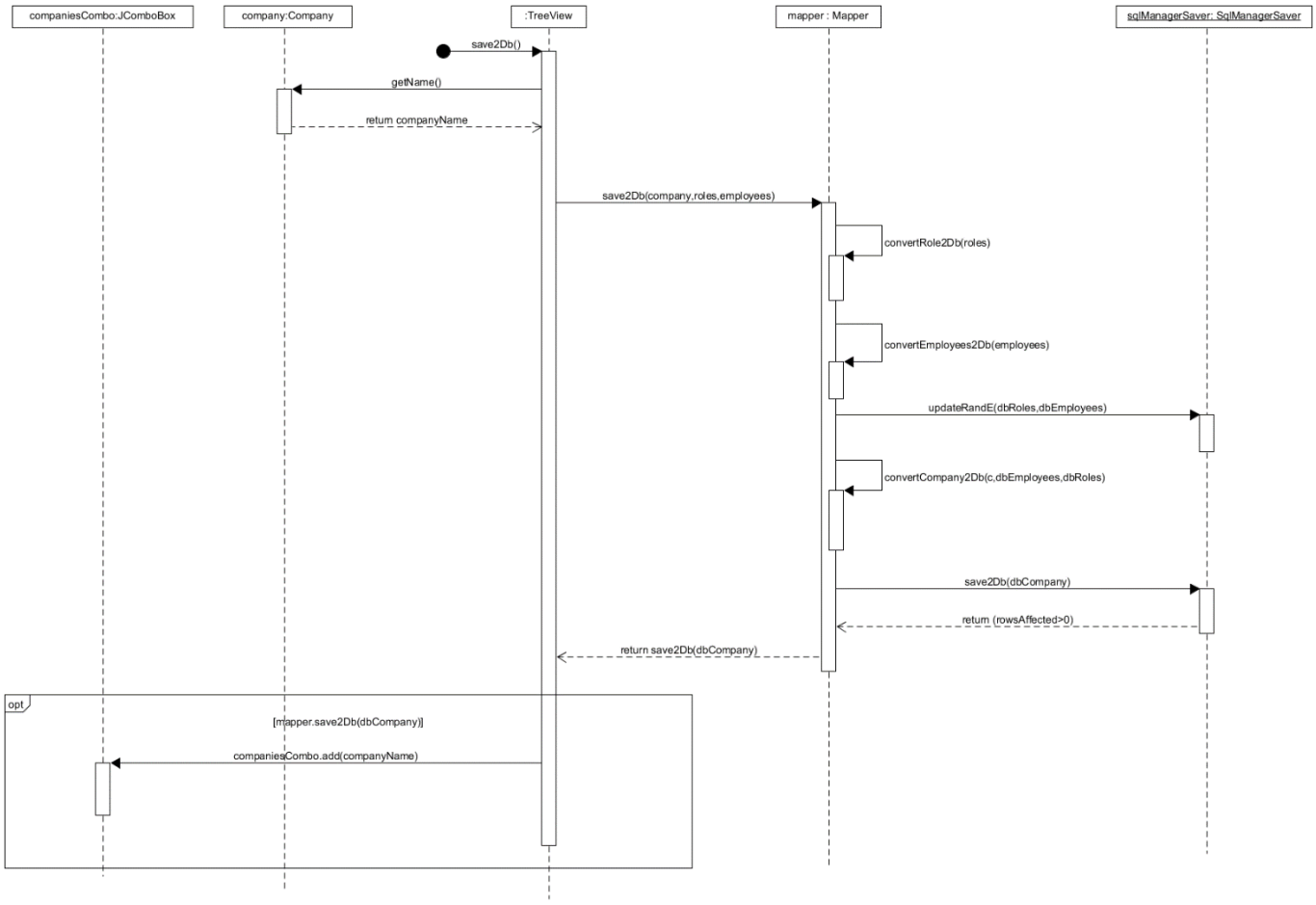
Il primo sequence diagram da mostrare è relativo all'inserimento di un ruolo ammissibile ad un'unità.



L'evento di click viene catturato dall'ActionListener scelto come listener del JButton che l'utente usa per aggiungere il ruolo. Tramite l'ausilio del Communicator l'ActionPerformed ricava l'unità e il ruolo selezionati, dopodiché crea il Command adibito all'esecuzione dell'operazione di aggiunta del ruolo (l'AddRoleToUnitCommand), passandogli il ruolo e l'unità selezionati. Il comando viene affidato al command handler che esegue semplicemente il comando richiamando su di esso il metodo doIt(). A questo punto, se sia il selected role, che la selectedUnit sono diversi da null si può procedere all'aggiunta del ruolo. Il comando per aggiungere il ruolo ammesso viene richiamato sull'unità; se quindi tra i ruoli ammissibili dell'unità non vi è già il ruolo selezionato questo viene inserito e l'Observer (i.e. il treeView) viene notificato dell'evento.

-Save to db

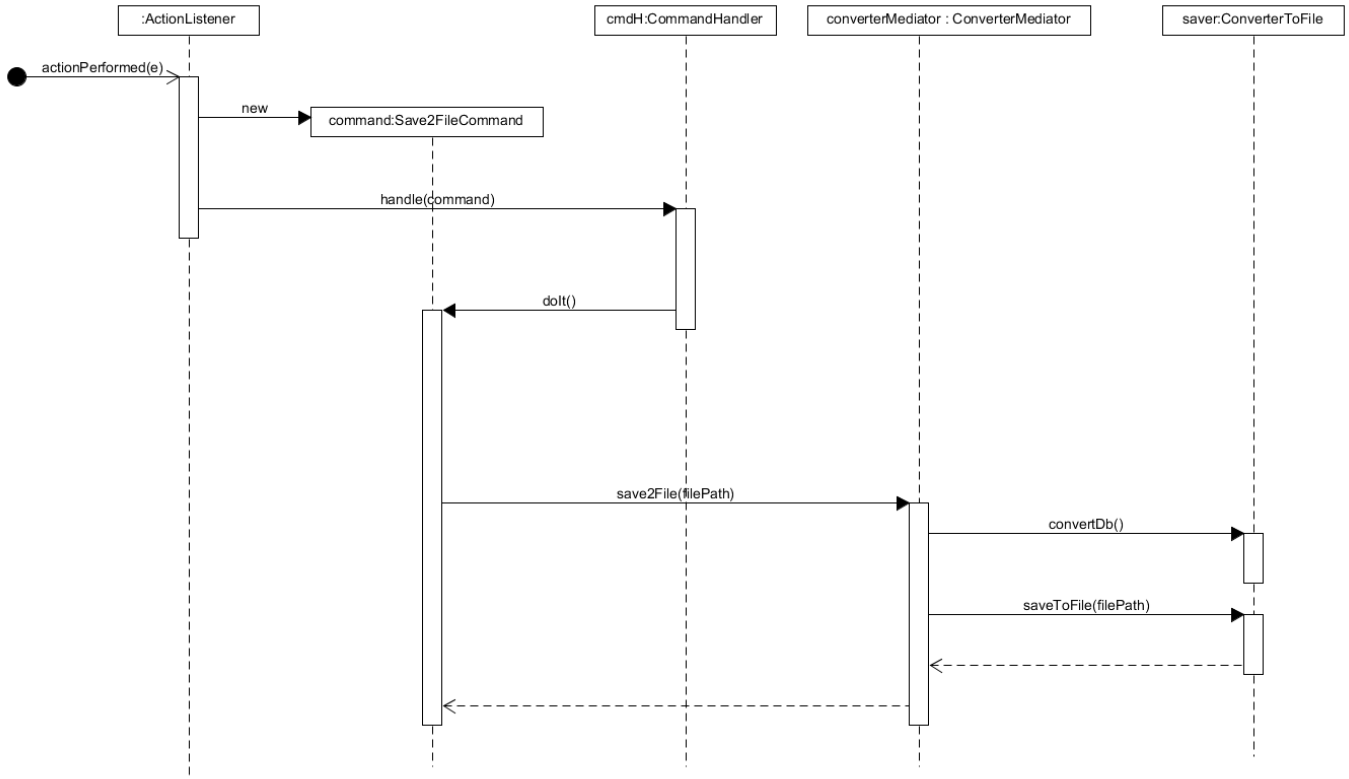
Un altro sequence diagram degno di nota è quello relativo al salvataggio dell'organigramma corrente su db.



Sul treeView viene richiamato il comando save2Db(). Il treeView ricerca la companyName richiedendolo alla company corrente. Fatto ciò, richiede al mapper di salvare l'organigramma corrente sul database. Il mapper esegue le operazioni per convertire il view model in db model, dopodiché inserisce ruoli, dipendenti e lo stesso organigramma sul database, utilizzando il sqlManagerSaver. Una volta eseguite queste operazioni, il sqlManager risponde al mapper comunicandogli se la company salvata era già presente all'interno del database; questa informazione viene inoltrata al treeView che, in caso di risposta affermativa non fa nulla, altrimenti aggiunge al comboBox delle company la stringa col nome della company, segnalando, in tal modo, che nel database è presente un nuovo organigramma che, eventualmente, può essere caricato e mostrato.

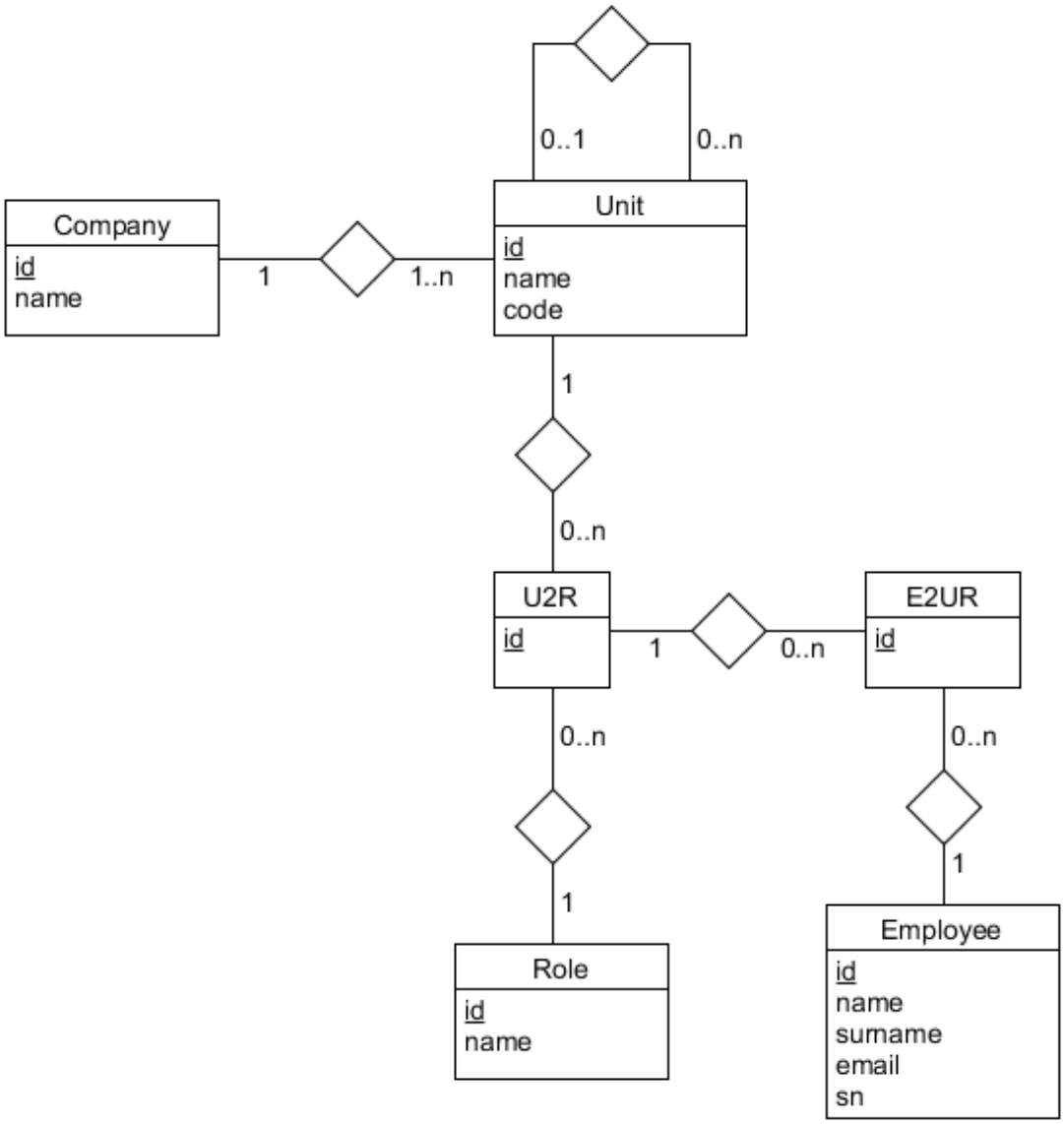
-Save to file

Il seguente sequence diagram è relativo al salvataggio del contenuto del database su file.



L'action performed scelto come listener del JMenuItem, responsabile per il salvataggio del db su file, riceve l'evento, crea perciò il comando che sarà responsabile per il salvataggio del db su file. Affida il comando al commandHandler che richiama sul primo il metodo doIt(); il comando viene perciò mandato in esecuzione. Il command, dopo aver ottenuto il path del file su cui effettuare la save, delega le operazioni al converterMediator. Questo a sua volta utilizza un ConverterToFile; prima richiama su di esso il convertDb, che converte il contenuto del db in un formato adatto per essere poi scritto su un file, e poi richiama il saveToFile, inserendo il path del file stesso.

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



Lo schema utilizzato è il seguente

Company(id,name)
Unit(id,name,companyId,parentUnit,code)
U2R(id,unitId,roleId)

Role(id,name)
E2UR(id,empId,u2rId)
Employee(id,name,surname,email,sn)

Sono state definite le seguenti foreign key:

Unit[parentUnit] $\subseteq_{FK}$Unit[id]
Unit[companyId] $\subseteq_{FK}$ Company[id]

U2R[unitId] $\subseteq_{FK}$ Unit[id]
U2R[roleId] $\subseteq_{FK}$ Role[id]

E2UR[u2rId] $\subseteq_{FK}$ U2R[id]
E2UR[empId] $\subseteq_{FK}$Employee[id]

Inoltre, oltre alle chiavi primarie "id" scelte per ogni entità sono stati definiti le seguenti chiavi secondarie.

Per Unit: UNIQUE{code}
Per Employee: UNIQUE{sn}
Per U2R: UNIQUE{unitId, roleId}
Per U2UR: UNIQUE{u2rId, empId}

Per la gestione delle foreign key è stato scelto di aggiungere l'opzione "DELETE ON CASCADE".
Come dbms è stato scelto MySql.

L'idea di base è stata quella di considerare una company come costituita da un certo numero di unità. Poiché l'organigramma ha una struttura gerarchica è stato necessario pensare a come rappresentare questo aspetto in relazione alla modellazione dati su db. Ogni unità può presentare una unità padre, si è scelta quindi una relazione ricorsiva. Come già detto, ogni unità può presentare un'unità padre (solo l'unità radice non lo ha) e può essere padre di un numero qualunque di figli. In tal modo siamo in grado di rappresentare strutture con una profondità qualsiasi.
I ruoli ammissibili di un'unità sono modellati come elementi di U2R. Ogni U2R è associato, evidentemente, a una sola unità e a un solo ruolo (la coppia ruolo-unità identificano la U2R).
L'assegnamento di un ruolo a un dipendente è modellato tramite la E2UR. In questo caso una E2UR ha una relazione con un U2R e un dipendente; in tal modo stiamo dichiarando che quel particolare dipendente ha un ruolo (ammissibile) assegnato in relazione alla coppia ruolo-unità definita dall'U2R.
Infine, abbiamo i ruoli e i dipendenti.

-Struttura del database su MySql

Seguendo la struttura sopra indicata le varie table sono state così strutturate:

companies:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
name	VARCHAR(45)	☐	☑	☑	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	☐	

employees:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
name	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
surname	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
email	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
sn	INT	☐	☑	☑	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	☐	

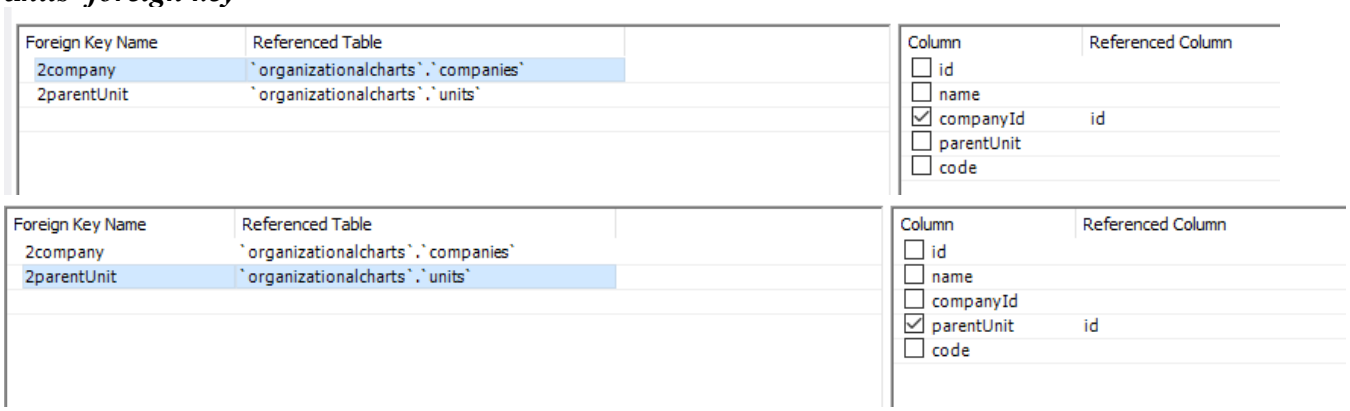
roles:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
name	VARCHAR(45)	☐	☑	☑	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	☐	

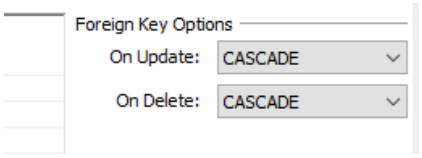
units:

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
name	VARCHAR(45)	☐	☑	☐	☐	☐	☐	☐	☐	
companyId	INT	☐	☑	☐	☐	☐	☐	☐	☐	
parentUnit	INT	☐	☐	☐	☐	☐	☐	☐	☐	NULL
code	INT	☐	☑	☑	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	☐	

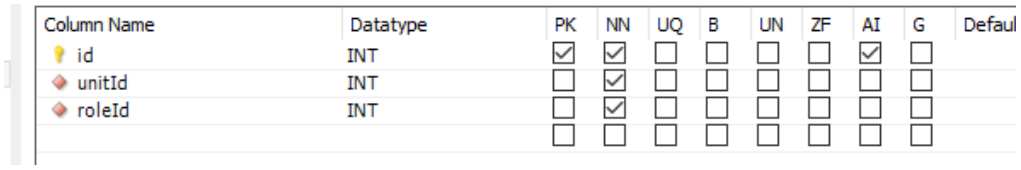
units' foreign key



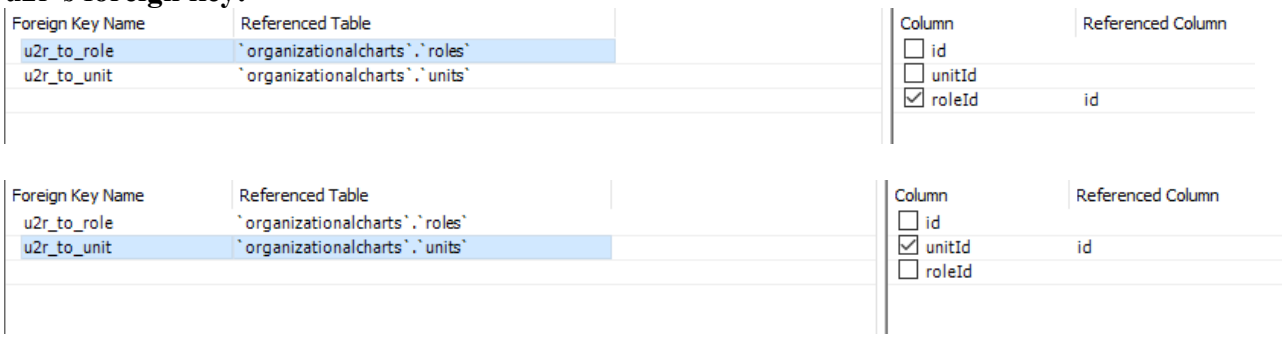
Per entrambe è stato scelto:



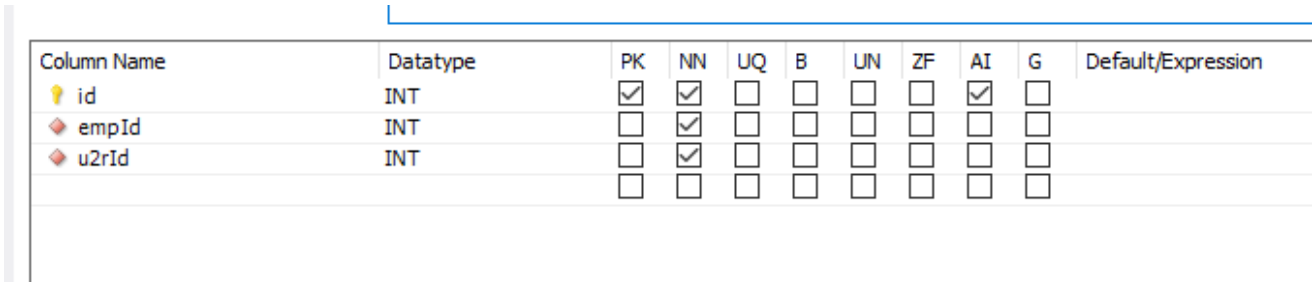
u2r:



u2r's foreign key:



e2ur:

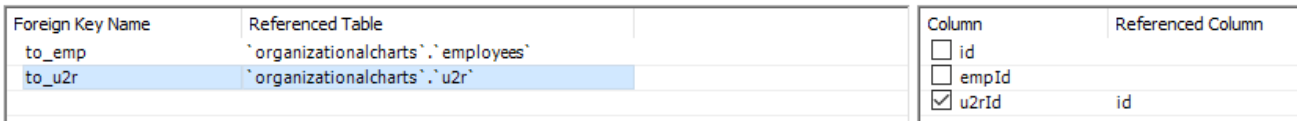


Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
id	INT	☑	☑	☐	☐	☐	☐	☑	☐	
empId	INT	☐	☑	☐	☐	☐	☐	☐	☐	
u2rId	INT	☐	☑	☐	☐	☐	☐	☐	☐	
		☐	☐	☐	☐	☐	☐	☐	☐	

e2ur's foreign key:

Foreign Key Name	Referenced Table
to_emp	`organizationalcharts`.`employees`
to_u2r	`organizationalcharts`.`u2r`

Column	Referenced Column
☐ id	
☑ empId	id
☐ u2rId	



Foreign Key Name	Referenced Table
to_emp	`organizationalcharts`.`employees`
to_u2r	`organizationalcharts`.`u2r`

Column	Referenced Column
☐ id	
☐ empId	
☑ u2rId	id

E. Scelte Progettuali (Design Decisions)

-Design pattern

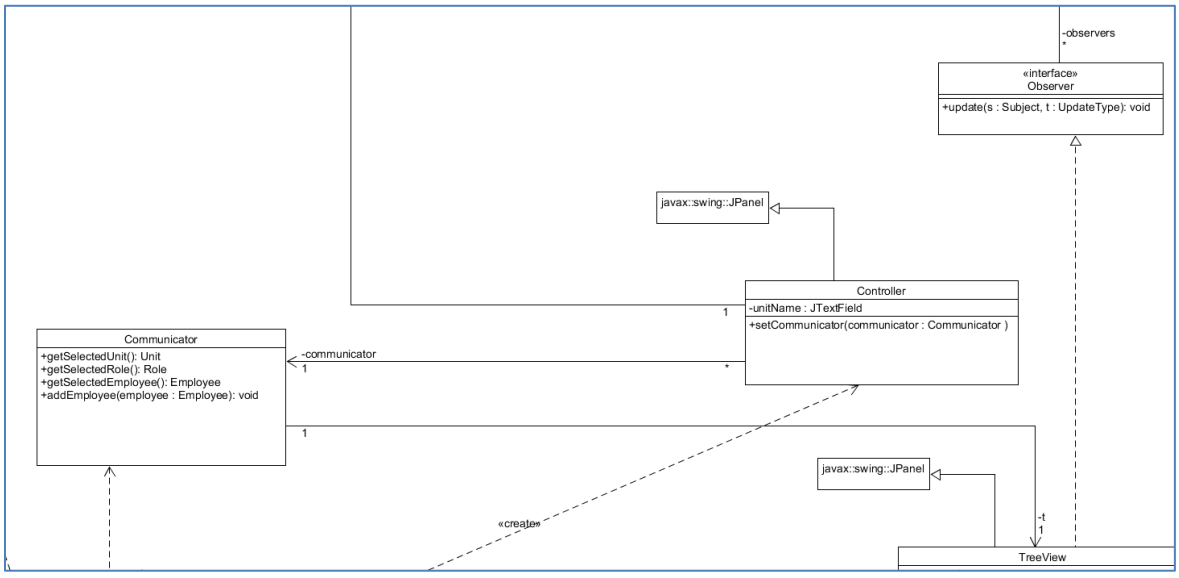
Una parte molto importante nel design del sistema è stata relativa alla scelta dei design pattern da implementare.

La struttura principale del sistema è stata realizzata tramite l'applicazione del pattern **MVC** (model view controller). I dati sui quali si basa l'organigramma, difatti, costituiscono il model che rappresenta lo stato dell'applicazione; la rappresentazione grafica è la view, mentre il pannello che permette di modificare l'organigramma costituisce il controller. Le modifiche vengono eseguite a partire dal controller, questo modifica direttamente i dati sul modello, tali cambiamenti si ripercuotono successivamente sulla view.

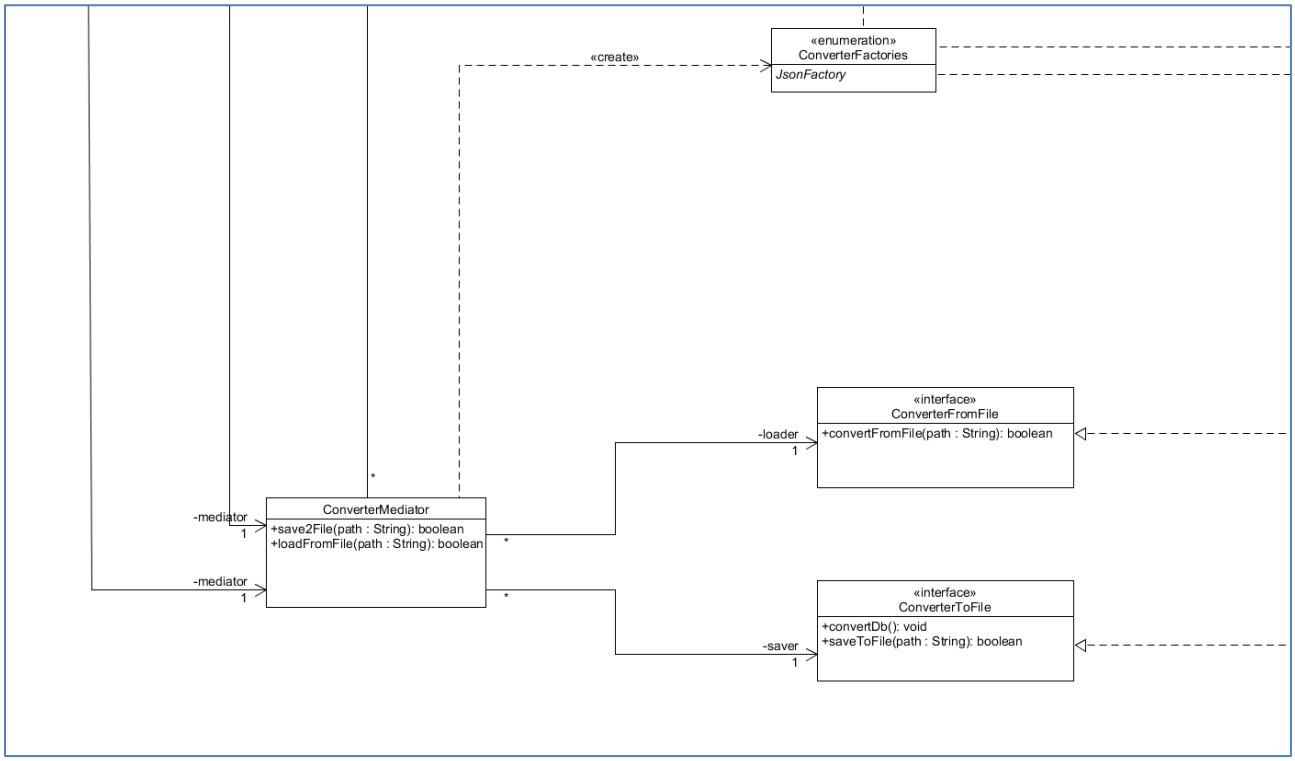
Grazie a questo pattern e ad altri, è stato possibile disaccoppiare la view dal controller e perciò, sarebbe potenzialmente possibile recuperare e riusare il codice delle singole parti del modello MVC.

Nella realizzazione dell'MVC sono stati usati altri pattern molto importanti.

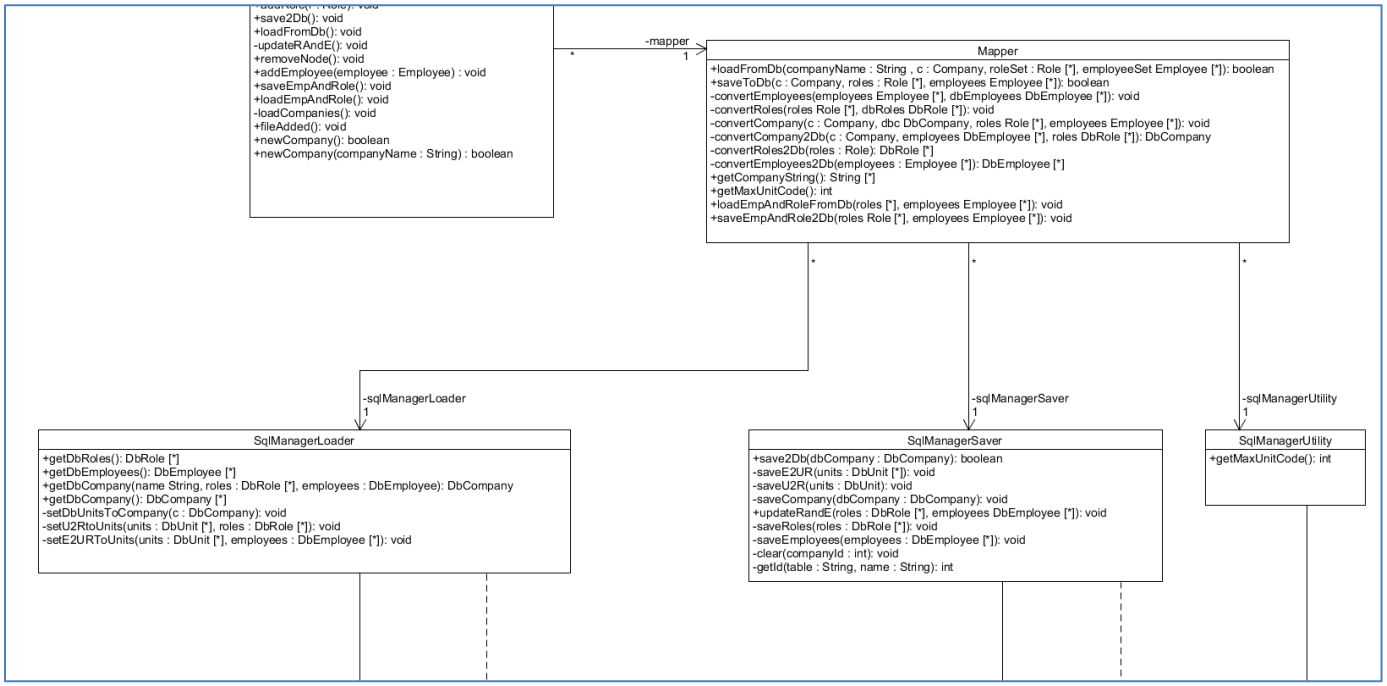
Façade: è stato necessario costruire una façade che permettesse al controllore di capire, di volta in volta, quali fossero i dipendenti, ruoli o unità selezionate senza perdere, però, il disaccoppiamento tra controllore e view (si tratta della classe Communicator).



Mediator: il pattern mediator è stato applicato per permettere di gestire le operazioni di salvataggio e caricamento da e per i file (il ConverterMediator). In questo caso l'applicazione di questo pattern consente di disaccoppiare l'MVC dalla parte del sistema dedicata al management dei file. Il mediator infatti, in questo caso, permette di combinare le operazioni offerte dai converter per gestire le operazioni da e per i file.

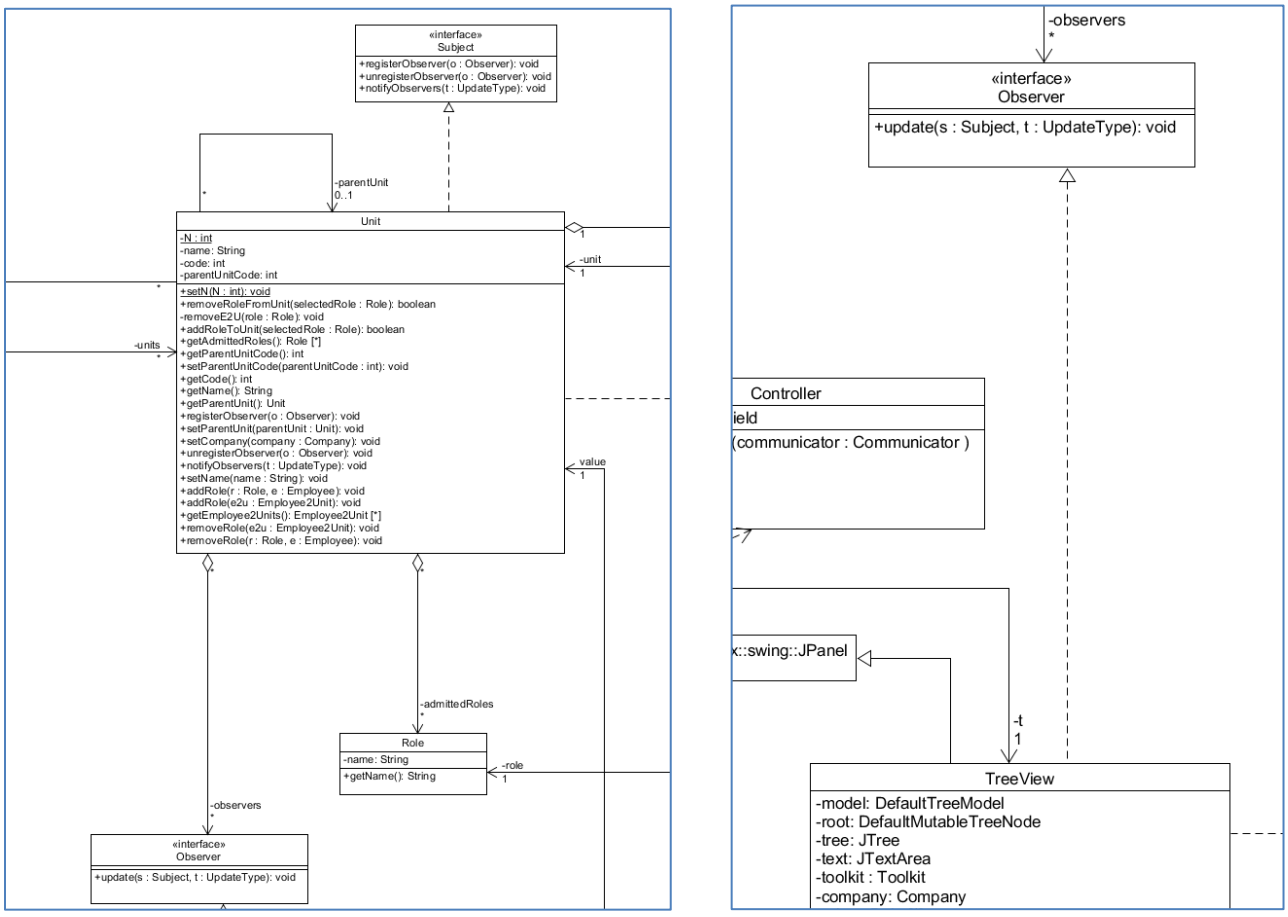


Vi è un ulteriore utilizzo di questo pattern, e ciò riguarda la gestione delle operazioni di salvataggio da e per il database. Si è difatti utilizzata la classe Mapper, la quale, oltre a permettere la comunicazione tra MVC e SqlManager (disaccoppiando le due parti al contempo) esegue le operazioni di conversione tra il view model e db model, combinando le operazioni offerte dai SqlManager.

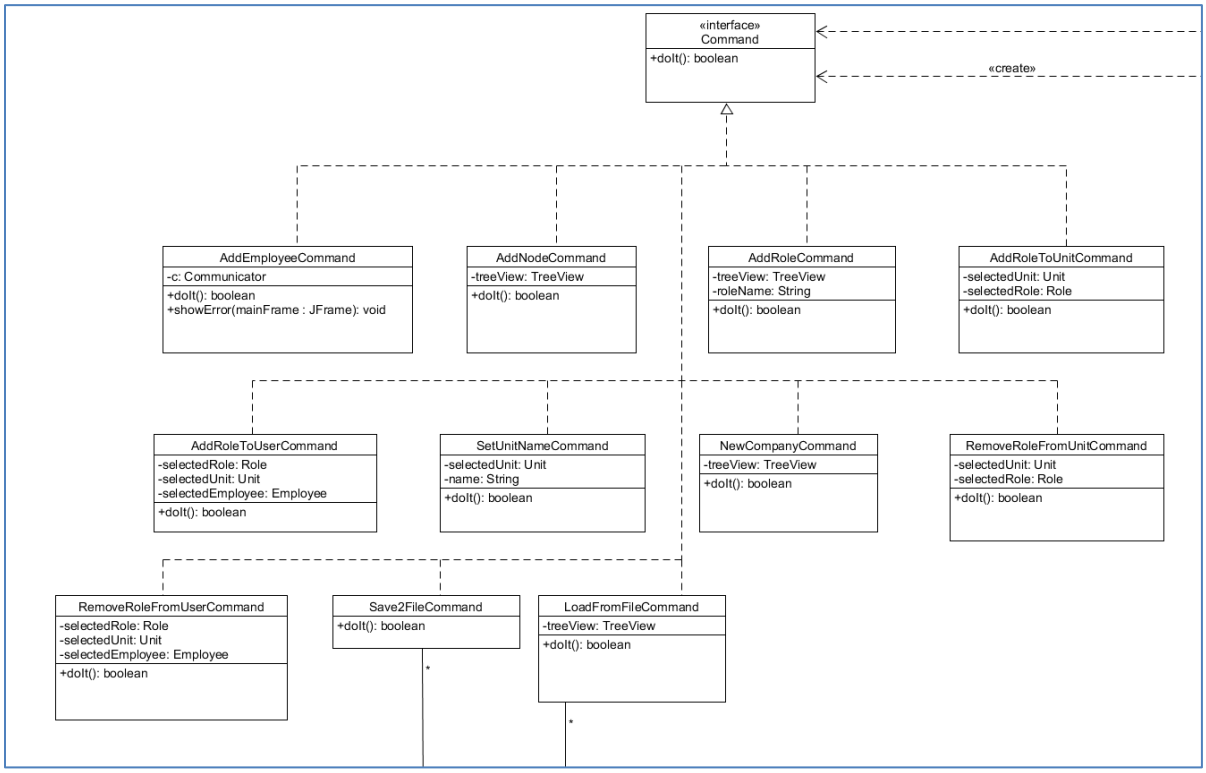


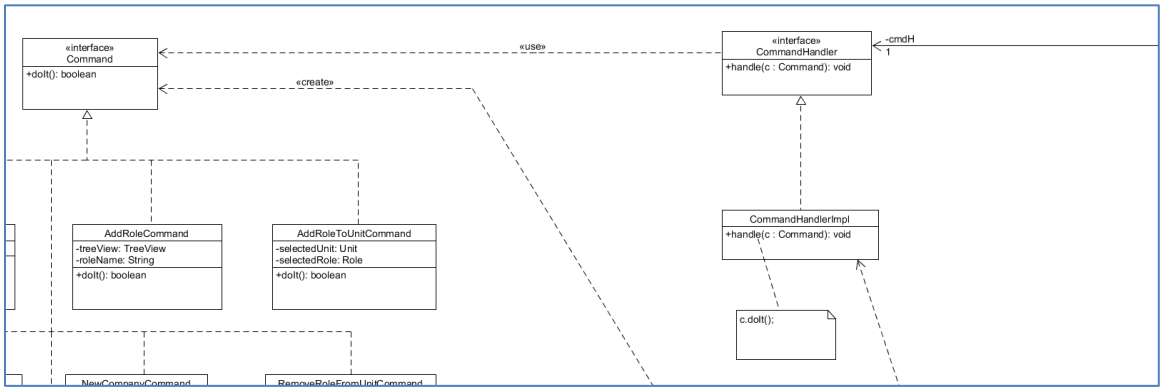
Observer: per permettere alla vista di aggiornarsi automaticamente nei casi di modifica del modello è stato implementato il pattern Observer. Le unità rappresentano i Subject, mentre il TreeView rappresenta l'Observer. Ogni volta che un'unità cambia stato, se questa modifica è significativa, viene riportata agli observer, utilizzando il metodo update offerto dall'interfaccia Observer. Il subject comunica anche agli observer il tipo di modifica eseguita (si è seguito l'approccio push). Questo è stato realizzato tramite l'introduzione di una enumeration "UpdateType". Le unità mantengono quindi una lista di observer che notificheranno in caso di cambiamenti. L'observer, in questo caso è rappresentato dalla classe TreeView, ove il modello dell'organigramma viene rappresentato (si noti come subject e observer appartengano a livelli diversi dell'applicazione; il modello si trova ad un livello più basso, mentre la vista ad un livello più alto).

Possiamo infatti osservare dal class diagram che Unit realizza l'interfaccia Subject e mantiene una lista di "observers"; mentre TreeView realizza l'interfaccia Observer

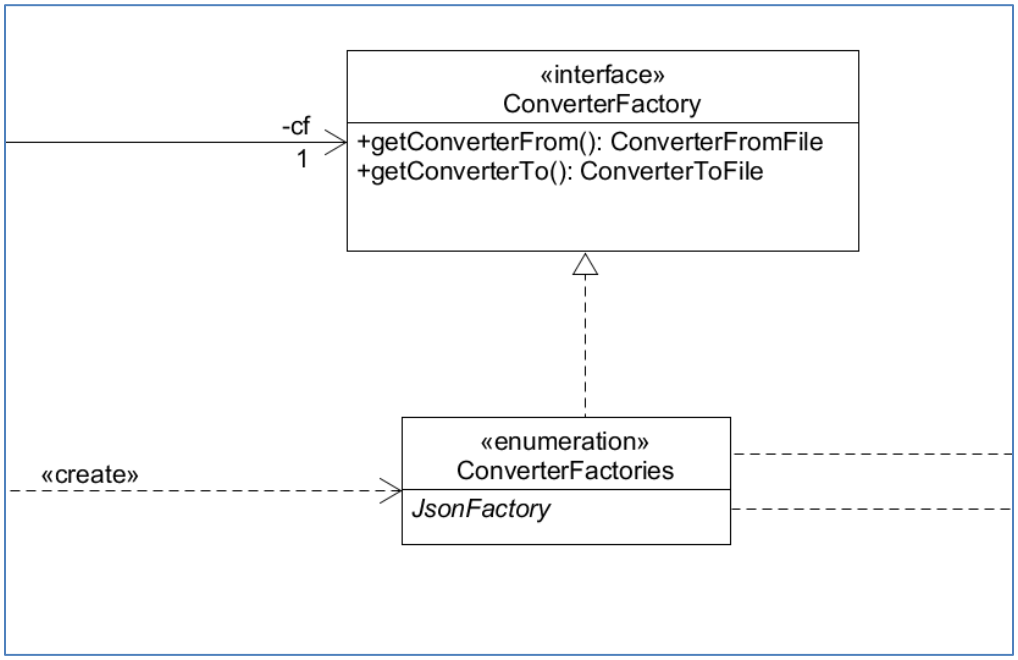


Command: Poiché le funzionalità richieste sono molte, si è deciso di strutturare la loro esecuzione implementando il pattern Command, incapsulando quindi l'esecuzione del comando all'interno di un oggetto (i comandi specifici) e disaccoppiando chi richiede l'esecuzione del task (il controller) da chi sa effettivamente eseguirlo (il command stesso ed eventuali receiver). Abbiamo quindi realizzato un CommandHandler in grado di mandare in esecuzione i comandi. Sebbene la scelta di realizzare un CommandHandler non fosse strettamente necessaria ai nostri obiettivi, l'utilizzo di un invoker permetterebbe in futuro di implementare dei meccanismi di esecuzione più elaborati, possibilmente, anche con la possibilità di eseguire le "undo" e "redo" sulle operazioni eseguite. La presenza di numerose funzioni da realizzare si traduce in una moltitudine di classi che realizzano l'interfaccia Command.

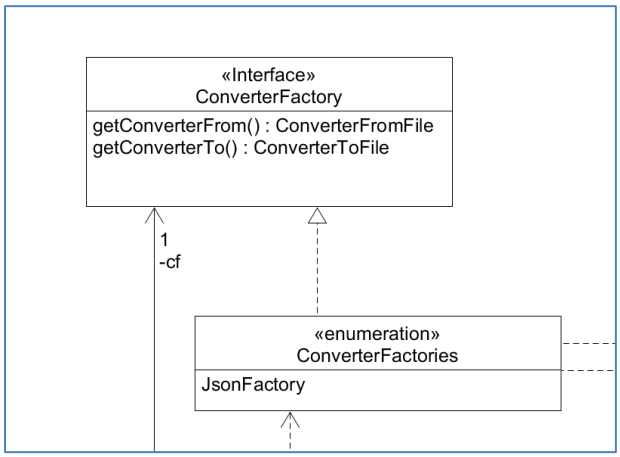


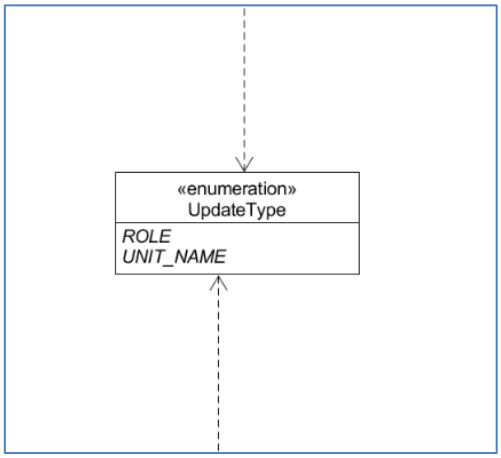


Abstract factory: come sappiamo, il pattern abstract factory è un design pattern utilizzato per offrire al client un'interfaccia per la creazione di oggetti correlati o dipendenti. In questo caso noi vorremmo avere un'interfaccia per la creazione delle classi concrete che implementano le interfacce per la gestione dei file. Poiché quindi per ogni tipo di file avremmo bisogno di una classe che realizzi ConverterFromFile e una che realizzi ConverterToFile creeremo un'interfaccia abstractFactory che ci offra i metodi (factory method) per ottenere questi converter, e poi, per ogni formato di file da gestire, creeremo una factory concreta apposita.



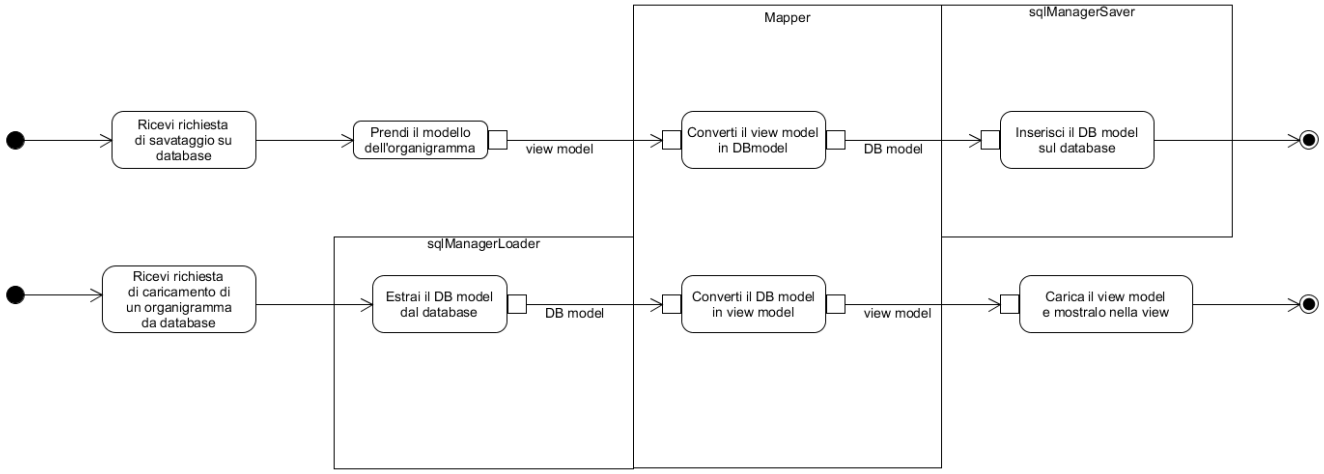
Singleton: come già detto il pattern singleton è stato utilizzato per permettere di comunicare alle unit il tipo di modifica effettuato, vi è però un ulteriore utilizzo; difatti, nella realizzazione del pattern abstractFactory, in corrispondenza del package "fileManager", si è deciso di realizzare la factory concreta JsonFactory come enumeration; in tal modo saremo sicuri che una sola istanza di ogni factory concreta verrà istanziata.





-View model e db model

La necessità di comunicare con un database introduce delle ulteriori difficoltà. È stato necessario separare infatti il view model dell'MVC dal database. Per realizzare ciò si è deciso di implementare un modello relativo al database (con le DbUnit, DbRole, ecc…), ovvero, un db model. La presenza di un mapper che effettua la conversione (dal view model al db model e viceversa) contribuisce al disaccoppiamento tra MVC e database. Il funzionamento di base è il seguente:

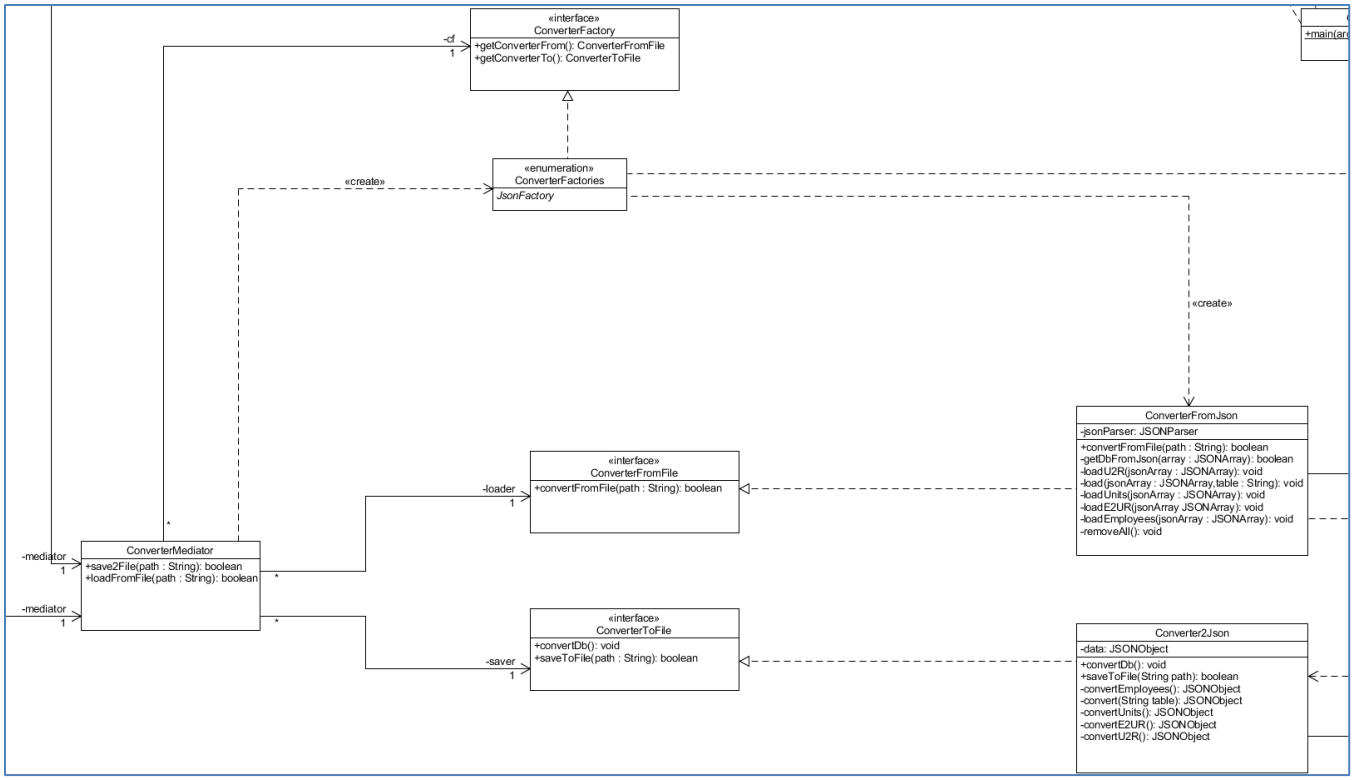


Il Mapper effettua la conversione dal view model verso il db model e viceversa, mentre il sqlManagerSaver inserisce il db model sul database. Il sqlManagerLoader invece estrae il db model dal database.

Grazie a questo tipo di progettazione il model view è totalmente disaccoppiato rispetto al db model; in tal modo sarà possibile modificare il *database schema* senza che la rappresentazione grafica debba necessariamente essere cambiata; basterà infatti mutare solo i SqlManager ed il Mapper.

-Gestione dei file

Un'altra scelta importante di progettazione è stata fatta in relazione al salvataggio su file. In questo caso si è infatti deciso di salvare i dati del database in formato Json. Volendo però aggiungere la possibilità, nelle successive versioni, di inserire facilmente nuovi formati di salvataggio si è così strutturato il codice.



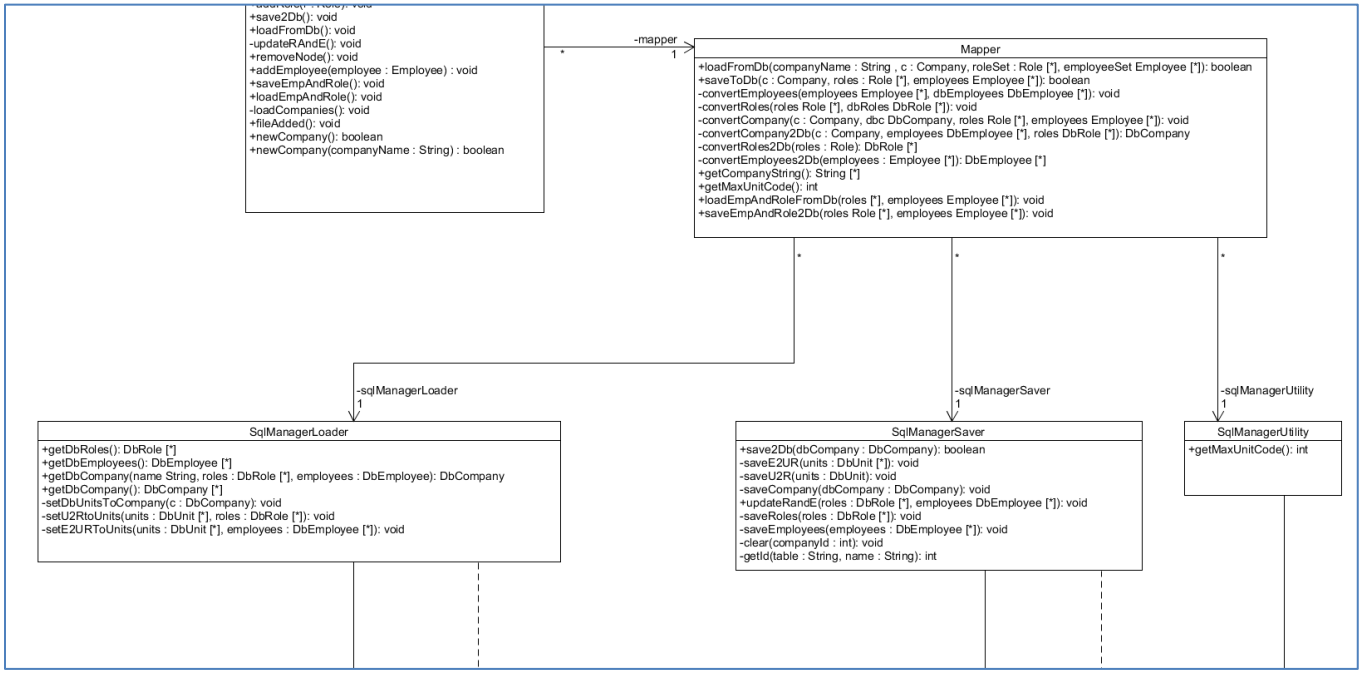
Il converter mediator utilizzato sia da LoadFromFileCommand che da Save2FileCommand mantiene un riferimento ai: ConverterFromFile, ConverteToFile e ad una ConverterFactory. Sia ConverterFromFile che ConverterToFile sono due interfacce che mettono a disposizione delle operazioni per caricare dati da file e inserire dati su file. Il Converter mediator non conosce le classi che poi implementeranno queste funzioni. La scelta sulle classi concrete è incapsulata nella ConverterFactory scelta. Si è deciso di implementare le factory concrete come istanze di una enumeration (in tal modo garantiamo che per ogni factory concreta esista una e una sola istanza). Nel progetto qui mostrato, ConverterMediator utilizza una JsonFactory, quindi i converter salveranno il contenuto del db in formato Json. In una successiva versione basterebbe aggiungere altri converter che implementino le interfacce per poi costruire la relativa concrete factory come nuova istanza della enumeration ConverterFactories. Cambiando la concrete factory utilizzata nel conveterMediator saremo in grado di salvare il contenuto del database in altri formati. Grazie all'abstract factory siamo in grado, infatti, di

disaccoppiare l'oggetto che desidera costruire i converter (il mediator) da quello che effettivamente li crea (la factory).

Tramite quindi le operazioni messe a disposizione da ConverterFromFile e ConverterToFile possiamo salvare il db o caricare il db da file.

-Comunicazione col database

La gestione del database viene effettuata a partire dal TreeView, il quale, ricevendo i comandi che necessitano un'interazione col database rimanda queste operazioni al Mapper, di cui TreeView mantiene il riferimento ad un'istanza.



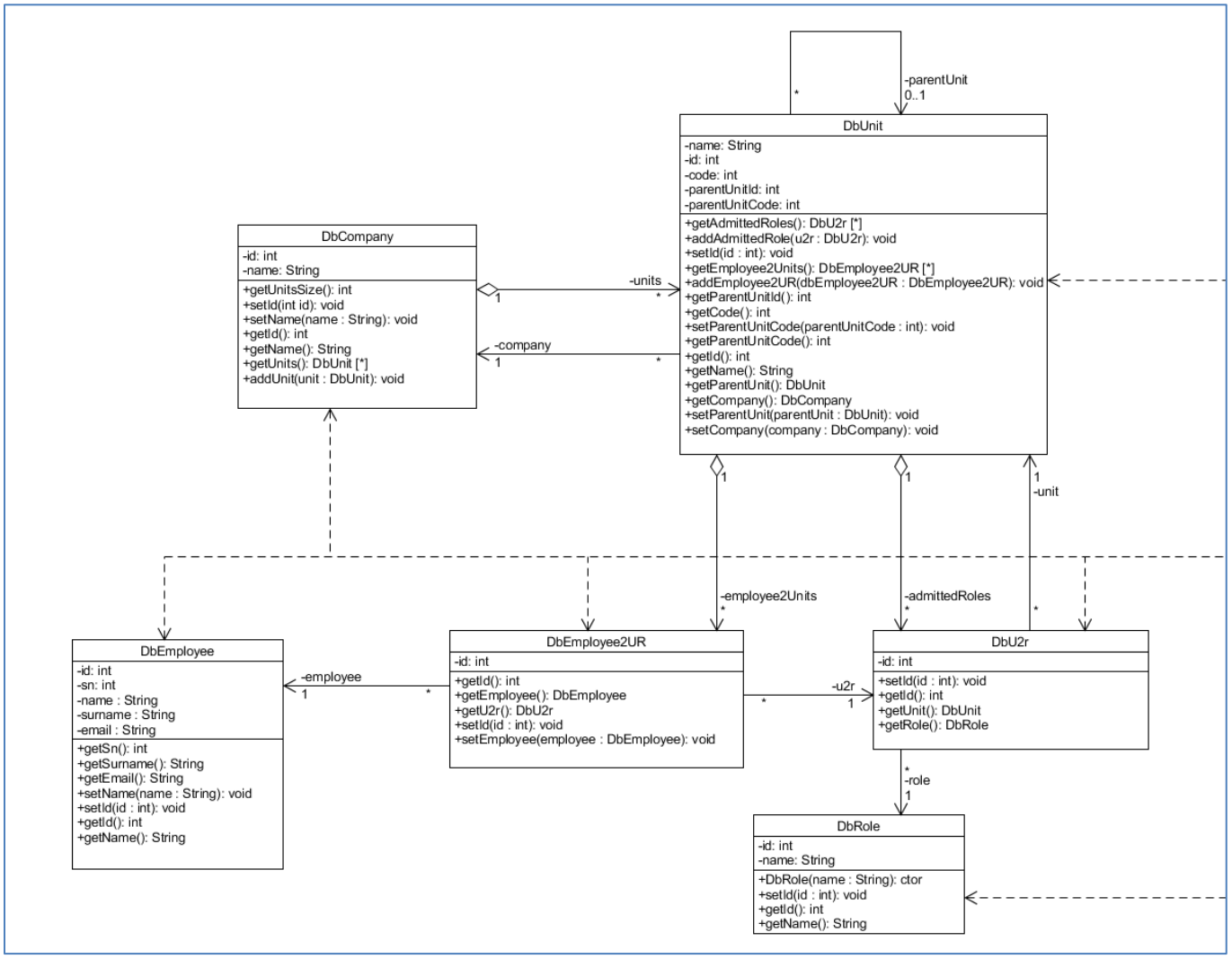
Il Mapper offre dei metodi per caricare/salvare un particolare organigramma (anche ruoli e dipendenti) da/per il database, ma anche operazioni di utilità, come: ricavare i nomi delle company presenti sul database in modo da permettere all'utente di scegliere una di esse; ricavare il massimo valore del "code" relativo alle unità per settare il valore del campo statico "N" in Unit in modo tale da evitare duplicati sui codici (che identificano le unità).

Il mapper esegue prettamente delle operazioni di conversione dal view model al db model e viceversa; il compito di ricavare il db model dal database o immetterlo sul db è dato a delle classi ausiliarie, SqlManagerLoader e SqlManagerSaver che offrono tutti i metodi necessari per il mapper, per eseguire caricamenti o salvataggi.

Il mapper si pone inoltre, in tal modo, da intermediario tra la parte che gestisce il database e l'MVC, riuscendo in tal modo a disaccoppiare le due parti.

-Discussione sul db model

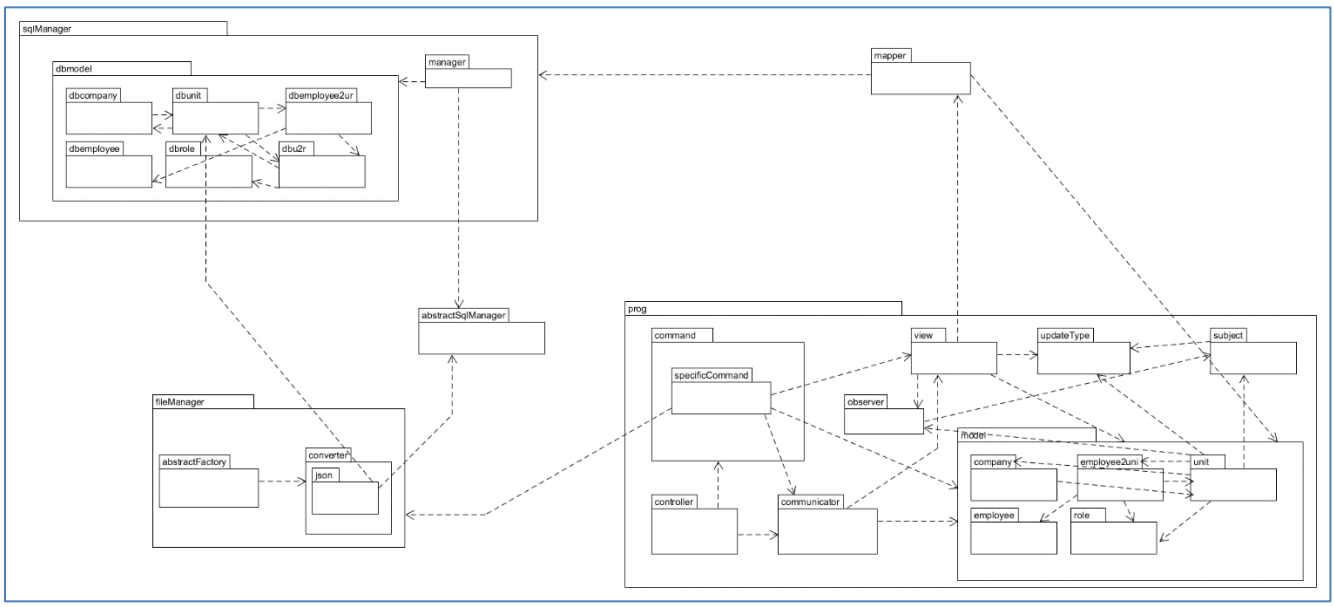
Come già anticipato, il db model viene utilizzato per permettere il disaccoppiamento tra l'MVC e il database, in questo modo infatti, la rappresentazione grafica del model può rimanere invariata anche nell'eventualità che dei cambiamenti sulla struttura del database schema possano avvenire. Ovviamente, in caso di effettive modifiche sarà necessario mutare sia il Mapper che i vari SqlManager.



Per quanto riguarda prettamente la struttura del db model, questo cerca di riprodurre quanto più fedelmente possibile la struttura del db schema, in modo tale da far sì che il passaggio dei dati da database a db model possa essere il più lineare possibile. Questo è un aspetto non necessario, ad esempio, sul view model, che essendo disaccoppiato non dipende strettamente dalla struttura del database.

-Analisi sul diagramma di package

L'analisi sulle scelte progettuali può essere effettuata anche andando ad analizzare la situazione che il sistema rivela in relazione al diagramma di package ottenuto.



Notiamo innanzitutto un'alta coesione interna in relazione ai principali package del sistema, ovvero, prog, sqlManager e fileManager; mentre è anche evidente un basso accoppiamento in relazione alle dipendenze esterne dei pacchetti. Si tratta di un aspetto molto importante in quanto favorisce la modularità del sistema. All'interno di prog è stato realizzato l'MVC che implementa le funzionalità principali di gestione degli organigrammi. Il pacchetto model contiene il modello dell'organigramma. Come già anticipato, vi è il package prog::communicator che permette di mantenere la view e il controller disaccoppiati, ciò è reso ancora più evidente all'interno del diagramma. Le dipendenze che il pacchetto prog presenta verso l'esterno sono relative alla dipendenza tra view e mapper e tra specificCommand e converter; la view, infatti, utilizza il mapper come intermediario per comunicare col sqlManager (effettuando in tal modo le operazioni di trasferimento dei dati da/verso il database. Grazie a questa scelta di progettazione, difatti, l'MVC non è consapevole della presenza del sqlManager, il che ci permetterebbe di riutilizzare lo stesso MVC indipendentemente dal sqlManager scelto.

La relazione tra specificCommand e fileManager è invece legata all'utilizzo dei comandi per salvare il contenuto del db su file e di caricare il contenuto del db dal file. Anche in questo caso, tuttavia, la dipendenza non è diretta, difatti, utilizzando il ConverterMediator è possibile disaccoppiare le classi responsabili per la conversione dei dati del db da/per il file dal comando responsabile per l'esecuzione della richiesta dell'utente.

Vi è un'ulteriore dipendenza degna di nota, relativa alla relazione tra json, manager e abstractSqlManager. I primi due pacchetti infatti comunicano col database, per questo motivo devono condividere le modalità di accesso a quest'ultimo. La definizione di queste modalità è incapsulata dall'AbstractSqlManager, inserito all'interno dell'omonimo package.

F. Progettazione di Basso Livello

- **Il modello**

Come già detto, la modellazione dell'organigramma è stata effettuata tramite la realizzazione del package prog::model, in cui, per ogni entità significativa dell'organigramma è stata introdotta una classe per modellarla.

Company

Una company rappresenta un singolo organigramma, essa è costituita da una lista di unità e da un nome che la identifica.
La classe offre la possibilità di accedere al nome della company e alla lista delle unità (ovviamente, restituita in modo tale da evitare di poterla modificare). La classe mette anche a disposizione la possibilità di eliminare tutte le unità, di settare il nome dell'unità e di rimuovere/aggiungere dinamicamente particolari unità. Questo permette la gestione dinamica della company; l'applicazione dovrà infatti essere in grado di poter gestire dinamicamente il modello dell'organigramma.

Employee

Un dipendente viene rappresentato tramite la classe Employee. Questa è caratterizzata da un name, surname, e-mail e un sn, quest'ultimo permette di identificare il dipendente all'interno dell'azienda. La classe mette a disposizione la possibilità di accedere ai campi della classe tramite dei getter.

Role

Un ruolo è modellato tramite la classe Role, caratterizzata e identificata dal nome del ruolo. È possibile accedere al nome tramite un metodo getter.

Unit

Un'unità viene modellata tramite la classe Unit. Un'unità è caratterizzata dal proprio nome, da un codice (che la identifica), dalla company alla quale è relativa (un riferimento all'oggetto company), da una lista di ruoli ammissibili, da una lista di ruoli assegnati ai dipendenti (una lista di istanze di Employee2Unit) e dal riferimento al parentUnit, che nel caso della radice dell'organigramma è nullo. Viene inoltre mantenuto il codice della parentUnit. La classe Unit realizza l'interfaccia Subject, per rendere possibile l'implementazione del pattern Observer, e per tale motivo, oltre a concretizzare i metodi della suddetta interfaccia, mantiene una lista di oggetti di tipo Observer, che poi andrà a notificare con l'apposito tipo di aggiornamento tra quelli resi disponibili nell'enumeration UpdateType.
La classe Unit mantiene anche un intero N come campo statico della classe, questo per permettere che il campo "code", identificativo per gli oggetti di tipo Unit, sia auto-incrementale. Se infatti non ci troviamo in una fase di caricamento dal database (caso in

cui invece il valore del "code" è già definito. La situazione è distinta dall'uso di un booleano nel costruttore: "load") il valore di code viene settato sulla base di N, e poi N stesso viene incrementato. Per permettere di rendere il valore di N coerente coi codici presenti all'interno del database la classe mette a disposizione un metodo setN(N : int) che permette di settare il valore di N (si sceglie solitamente il max(code)+1, dove max(code) è il valore massimo di code all'interno del database).

La classe mette poi a disposizione dei metodi come "removeRoleFromUnit" e "addRoleToUnit" per permettere l'aggiunta e la rimozione dei ruoli ammissibili dall'unità. Nel caso di rimozione del ruolo ammissibile, tutti i dipendenti assegnati col ruolo eliminato (elementi della lista employee2units) devono anche essere eliminati dall'unità.
È poi possibile rimuovere e aggiungere i ruoli assegnati tramite "addRole" e "removeRole".
Oltre ai getter (che nel caso di proprietà a più valori restituiscono la versione non modificabile della proprietà) che permettono di accedere ai campi della classe vi sono anche metodi setter, come "setParentUnitCode" per settare il codice della parentUnit e "setParentUnit" che setta il riferimento alla parentUnit e aggiorna automaticamente il parentUnitCode (questi metodi sono in particolar modo importanti nella fase di comunicazione col database); vi è anche "setName" che permette di cambiare il nome dell'unità.
Sia nel caso dell'aggiunta/rimozione del ruolo ammissibile che nel caso dell'aggiunta/rimozione dell'Employee2Unit, come anche nel caso del cambiamento del nome dell'unità, vengono notificati gli observer tramite il metodo "notifyObservers". Nel caso del cambiamento del nome dell'unità si usa l'UpdateType "*UNIT_NAME*", negli altri casi si fa uso di "*ROLE".*

Employee2Unit

Per modellare l'assegnamento di un dipendente ad un ruolo è stata usata la classe Employee2Unit. Un Employee2Unit è caratterizzato da un'unità, da un ruolo e da un dipendente ed è identificato dalla tripla di questi tre elementi.
Tramite dei getter è possibile accedere ai campi dell'oggetto.

UpdateType

Si tratta di un'enumerazione che viene utilizzata per permettere ai Subject (le Unit) di comunicare all'Observer la tipologia del cambiamento avvenuto.

- **Observer**

Per realizzare il pattern Observer è necessario mettere a disposizione un'interfaccia che gli effettivi observer dovranno implementare. Questa interfaccia mette a disposizione l'operazione per permettere al subject di notificare gli observer del cambiamento, vi è infatti il metodo "update".

- **Controller**

Per realizzare il pattern MVC è stato necessario realizzare l'oggetto che incapsula, per l'appunto, il ruolo di controllore, ovvero, quell'oggetto che modifica, in base alle decisioni dell'utente, il modello. Il controller estende JPanel, quindi potrà essere utilizzato direttamente per fornire all'utente un'interfaccia grafica per eseguire le operazioni desiderate.
Il controller mantiene il riferimento a un CommandHandler, al quale darà il compito di gestire i comandi relativi ad ogni operazione messa a disposizione all'utente. Mantiene poi un riferimento a un oggetto istanza di Communicator, utilizzato per ricavare le informazioni in merito ai ruoli/dipendenti/unità ecc… attualmente selezionate dall'utente sulla view. Vi è anche un JTextField utilizzato per permettere all'utente di inserire il nuovo nome dell'unità.
È possibile anche settare dinamicamente il communicator, tramite il setter "setCommunicator".

Nella costruzione del Controller vengono definiti dei JButton, uno per ogni operazione che dovrà essere eseguita dal controllore.

```
JButton addRoleToUser=new JButton( text: "Add role to user");
JButton removeRoleFromUser=new JButton( text: "Remove role from user");
JButton setUnitTitle=new JButton( text: "Set title to unit");
JButton addEmployee = new JButton( text: "Add Employee");
JButton addRole2Unit=new JButton( text: "Add role to unit");
JButton removeRoleFromUnit=new JButton( text: "Remove role from unit");
```

Ad ognuno di essi verrà poi assegnato un actionListener che non fa altro che affidare al commandHandler il comando appena creato, relativo all'operazione offerta dal button corrente.

```
addRoleToUser.addActionListener((e)-> cmdH.handle(new AddRoleToUserCommand(communicator.getSelectedRole(), communicator.getSelectedUnit(), communicator.getSelectedEmployee())));
removeRoleFromUser.addActionListener((e)-> cmdH.handle(new RemoveRoleFromUserCommand(communicator.getSelectedRole(), communicator.getSelectedUnit(), communicator.getSelectedEmployee())));
setUnitTitle.addActionListener((e)-> { cmdH.handle(new SetUnitNameCommand(communicator.getSelectedUnit(),unitName.getText().trim())); unitName.setText(""); });
addEmployee.addActionListener((e)->cmdH.handle(new AddEmployeeCommand(communicator)));
addRole2Unit.addActionListener((e)->cmdH.handle(new AddRoleToUnitCommand(communicator.getSelectedUnit(), communicator.getSelectedRole())));
removeRoleFromUnit.addActionListener((e)->cmdH.handle(new RemoveRoleFromUnitCommand(communicator.getSelectedUnit(), communicator.getSelectedRole())));
```

Il pannello del controllore viene quindi arricchito dei bottoni e del text field.

- **Communicator**

Per permettere al controllore di eseguire le sue operazioni ed al contempo mantenere quest'ultimo e la view disaccoppiati è stato realizzato un Communicator (implementazione del pattern Façade) il quale non fa altro che inoltrare eventuali ricerche di utenti, ruoli e dipendenti selezionati (anche l'aggiunta dei dipendenti) direttamente alla view.

- **TreeView**

La classe TreeView rappresenta la componente della vista relativa al pattern MVC. La vista estende JPanel, come nel caso del Controller può quindi essere applicata direttamente al frame che conterrà tutte le componenti dell'applicazione. TreeView implementa inoltre l'interfaccia Observer, questo per realizzare il pattern Observer.

Per realizzare la rappresentazione grafica dell'Organigramma si è deciso di utilizzare la classe JTree di javax::swing.
Un TreeView presenta come campi di istanza il "DefaultTreeModel" e il "JTree" per la gestione dell'albero. Viene inoltre mantenuto il riferimento alla radice dell'albero come "DefaultMutableTreeNode".
Per mostrare il contenuto di un'unità in termini di company di appartenenza, lista di ruoli ammessi, di ruoli assegnati ecc… viene utilizzata una JTextArea non modificabile.
Per mantenere inoltre la corrispondenza tra nodi e unità viene utilizzata una mappa. Un Toolkit viene utilizzato per segnalare eventuali errori come segnali sonori all'utente.
Viene anche mantenuto il riferimento alla Company attualmente rappresentata dalla vista.
Sono inoltre mantenuti la lista dei ruoli e dipendenti attualmente disponibili, rappresentati all'interno di due JComboBox. Si mantiene anche il riferimento al ruolo/dipendente selezionato.
Per la gestione della comunicazione col database viene mantenuto un riferimento a un oggetto istanza della classe Mapper (implementazione del pattern mediator, che permette di eseguire le operazioni di caricamento/salvataggio).
Viene mantenuta inoltre una lista dei nomi della Company attualmente salvate all'interno del database, e un JComboBox è utilizzato per mostrarle all'utente.

TreeView implementa inoltre l'interfaccia "TreeSelectionListener", ovvero, ogni volta che l'utente effettua la selezione di un nodo viene richiamato il metodo "valueChanged": viene identificato il nodo selezionato, tramite la mappa si risale alla Unit corrispondente e si mostra nella textArea la rappresentazione sottoforma di stringa dell'unità stessa (che contiene tutte le sue informazioni come: nome, company di appartenenza, ruoli ammessi, ecc…).

Come già detto, TreeView realizza Observer, quindi deve mettere a disposizione il metodo "update". Se l'UpdateType sarà di tipo ROLE, allora basterà settare la textArea con la rappresentazione attuale, sottoforma di stringa, dell'unità correntemente selezionata; altrimenti, se quindi l'UpdateType è di tipo UNIT_NAME sarà anche necessario cambiare il nome del nodo del JTree tramite il metodo "setUserObject".

I metodi getSelectedRole, getSelectedEmployee e getSelectedUnit vengono utilizzati dal Communicator, come già detto.

Per realizzare la funzionalità di aggiunta di una nuova unità all'organigramma è stato definito un metodo apposito in TreeView, ovvero, addNode(u : Unit). Questo comando fa uso del DefaultTreeModel. In particolare, quando si aggiunge una nuova unità all'organigramma è necessario settare il parentUnit, la company di appartenenza, è necessario registrare TreeView come observer della nuova unità, aggiungere l'unità alla lista della company e inserire la relazione nodo-unità nella mappa.

È stato definito un metodo per aggiungere un nuovo ruolo, "addRole": questo metodo va semplicemente a controllare se il ruolo che si vuole aggiungere è già presente nei ruoli disponibili, in caso contrario lo aggiunge sia all'insieme dei ruoli che al comboBox che li visualizza.

Per permettere all'utente di rimuovere nodi è stato realizzato il metodo "removeNode". Questo metodo verifica se il nodo selezionato è valido (i.e. non è la radice), in caso affermativo non elimina solo il nodo selezionato ma anche tutti quelli che presentano quello selezionato come antenato. L'eliminazione consiste nella rimozione dell'unità dalla company e dalla mappa.

Per permettere l'aggiunta di un dipendente si è implementato il metodo addEmployee. Per prima cosa si verifica se il dipendente che si vuole aggiungere è già presente o meno tra i dipendenti disponibili, in caso affermativo l'aggiunta non può avvenire, un messaggio di errore viene mostrato all'utente. Altrimenti, il nuovo dipendente viene aggiunto e visualizzato.

Per permettere la creazione di una nuova company si è realizzato il metodo newCompany. Esistono due versioni di questo metodo, una senza parametri in input e una con il companyName come stringa di parametro. Quella senza parametro richiama il metodo col parametro passando come nome "WorkBench". Il metodo con parametro verifica innanzitutto se all'interno del database vi è già una company con quel nome, in caso affermativo non procede con la creazione. Dopodiché ripulisce la mappa, crea una nuova unità radice (alla quale si registra come observer), elimina tutti i figli dal nodo radice e lo setta con l'unità radice, aggiungendolo nella mappa, infine, ricarica il modello.

Per permettere di salvare l'organigramma corrente sul database è stato realizzato il metodo "save2Db". Poiché si è deciso che una company con nome "WorkBench" non possa essere salvata, come prima cosa si verifica questa condizione, in tal caso si mostra un messaggio di errore all'utente. Altrimenti, tramite l'uso del mapper, l'organigramma corrente viene salvato. Il metodo "saveToDb" del mapper restituisce un booleano che ci permette di verificare se quella company era già presente sul database o meno; in caso non fosse presente si aggiunge il nome della company al comboBox delle company sul database. Si noti come, dal punto di vista del TreeView, le operazioni effettuate sul database, almeno in questo caso, sono molto semplici, la presenza del mapper permette di nascondere alla view, infatti, i dettagli implementativi delle operazioni effettuate sul database.

Per permettere di caricare un organigramma da database è stato implementato il metodo "loadFromDb". Il metodo verifica come prima cosa che la company selezionata sia valida, in tal caso richiama sul mapper l'operazione "loadFromDb", passando come parametri company, selectedCompany, i ruoli e gli employees. Se il caricamento non è stato effettuato con successo il metodo termina qui, altrimenti è necessario, come prima cosa, aggiornare i comboBox di ruoli e dipendenti. Quando il mapper esegue l'operazione "loadFromDb", ciò che fa è aggiornare le variabili passate come parametro in modo tale che queste contengano, una volta terminata l'operazione, l'organigramma caricato dal database. Vengono quindi aggiunti i vari nodi e vengono legati i nodi alle unità all'interno della mappa.

Per permettere che le operazioni di caricamento e salvataggio di suoli e dipendenti possano essere effettuate separatamente da quelle dell'organigramma sono stati realizzati i metodi saveEmpAndRole e loadEmpAndRole. Questi, come principale operazione, rimandano la richiesta al mapper, ed eventualmente si occupano di aggiornare la view. Nel caso del caricamento di ruoli e dipendenti, viene prima chiesto all'utente se desidera procedere, infatti, in caso affermativo, l'organigramma corrente viene eliminato.

Ogni qual volta un file viene caricato, è necessario eseguire delle operazioni per aggiornare la vista. Per tale motivo è stato implementato il metodo fileAdded, questo si occupa di caricare dipendenti, ruoli e company dal database e di settare, coerentemente a ciò che è contenuto nel database, il valore di N all'interno di Unit. Infine, viene cancellato l'organigramma precedentemente visualizzato.

- ## Command

Per la realizzazione del pattern Command sono state usate diverse classi.
L'interfaccia command permette di definire i comandi come oggetti che mettono a disposizione il metodo doIt(). È stata inoltre definita l'interfaccia del CommandHandler, che definisce l'operazione di handle sul comando. Il commandHandler svolge il ruolo di invoker nel pattern. Come commandHandler concreto è stato realizzato il CommandHandlerImpl, che si limita ad eseguire la doIt() sul comando che gli viene passato come parametro.

- ## I comandi specifici

Possiamo quindi osservare le classi che implementano l'interfaccia command.

AddEmployeeCommand
AddEmployeeCommand Incapsula l'operazione per l'aggiunta di un nuovo dipendente. Ogni oggetto mantiene il riferimento al communicator che utilizzerà per far sì che la view aggiunga il nuovo dipendente.
Tramite un'apposita finestra l'utente potrà inserire i dati relativi al dipendente che desidera aggiungere. Una volta che questi dati sono inseriti, se ben formattati, viene creato l'oggetto Employee, e poi inviato alla view tramite il communicator.

AddNodeCommand
Questo comando specifico incapsula l'operazione per l'aggiunta di un nuovo nodo nell'organigramma (la nuova unità). L'oggetto mantiene un riferimento al treeView, che userà per comunicare l'aggiunta del nodo. Viene creata una unità di default (con nome: "*"), poi l'unità viene aggiunta alla view tramite il metodo "addNode".

AddRoleCommand
Questo comando specifico incapsula l'aggiunta di un nuovo ruolo. L'oggetto mantiene il riferimento al treeView, ma mantiene anche il nome del ruolo che bisogna aggiungere, passato in fase di costruzione dell'oggetto (il nome sarà ricavato da una textArea).
Il ruolo viene creato, e poi viene comunicata la richiesta di aggiunta tramite il metodo "addRole" di TreeView.

AddRoleToUnitCommand
Questo comando specifico incapsula l'operazione di aggiunta di un ruolo ammissibile ad un'unità. L'oggetto mantiene un riferimento all'unità alla quale aggiungere il ruolo e al ruolo stesso.
Il ruolo viene direttamente aggiunto all'unità, saranno poi i meccanismi del pattern Observer a permettere l'aggiornamento automatico della view, nel caso l'aggiunta sia stati portata a termine.

AddRoleToUserCommand
Questo comando specifico incapsula l'operazione per l'assegnamento di un ruolo ad un dipendente in relazione ad un'unità. L'oggetto mantiene il riferimento al ruolo selezionato, all'unità selezionata e al dipendente selezionato.
L'assegnamento del ruolo avviene direttamente applicandolo all'unità; come nel caso precedente saranno i meccanismi del pattern Observer a permettere di aggiornare la view.

LoadFromFileCommand
Questo comando specifico incapsula l'operazione per il caricamento del contenuto del database da file. L'oggetto mantiene il riferimento al ConverterMediator (quest'ultimo permette di disaccoppiare la gestione dell'MVC dalla parte dell'applicazione che gestisce le operazioni sui file), e il riferimento al treeView.
Il comando si assicura di ottenere dall'utente il path del file da cui caricare il contenuto del database, se lo si ottiene si procede comunicando al ConverterMediator il path, richiamando su di esso l'operazione di "loadFromFile". Sarà questo oggetto a gestire tutte le operazioni per eseguire il caricamento e, nell'eventualità di file corrotto anche di ristabilire lo stato del database. Se quindi il caricamento avviene con successo (lo si può capire dal booleano restituito da loadFromFile) allora si comunica al treeView che un file è stato caricato tramite il metodo "fileAdded".

NewCompanyCommand
Questo comando specifico incapsula l'operazione per la creazione di una nuova company. L'oggetto mantiene il riferimento al treeView.
Il comando richiede all'utente di inserire il nome della nuova company, una volta ottenuto tale nome richiede al treeView di creare la nuova company con il metodo "newCompany". Se l'operazione non avviene correttamente vuol dire che la company esiste già nel database, un messaggio di errore viene mostrato all'utente.

RemoveRoleFromUnitCommand
Questo comando specifico incapsula l'operazione per la rimozione di un ruolo ammissibile da un'unità. L'oggetto mantiene il riferimento all'unità selezionata e al ruolo selezionato.
Il comando inoltra la richiesta di eliminazione del ruolo ammissibile all'unità, lasciando che siano i meccanismi del pattern Observer ad aggiornare la view.

RemoveRoleFromUserCommand
Questo comando specifico incapsula l'operazione per il de-assegnamento di un ruolo ad un dipendente, in relazione ad un'unità. L'oggetto mantiene il riferimento al ruolo selezionato, all'unità selezionata e al dipendente selezionato.
Il comando, come visto in altri casi, inoltra semplicemente la richiesta di rimozione all'unità.

Save2FileCommand
Questo comando specifico incapsula l'operazione per il salvataggio del contenuto del database su file. L'oggetto mantiene il riferimento al ConverterMediator che userà per gestire le operazioni sul file.
Il comando richiede all'utente il path del file sul quale salvare il db content. Una volta ottenuto richiede al ConverterMediator di salvare il file tramite il metodo "save2File". L'esito dell'operazione viene mostrato all'utente.

SetUnitNameCommand
Questo comando specifico incapsula l'operazione per il cambiamento del nome di un'unità. L'oggetto mantiene il riferimento all'unità selezionata e mantiene il nuovo nome dell'unità (ottenuto precedentemente tramite una textArea).
La richiesta di cambiare il nome viene semplicemente inoltrata all'unità stessa.

- **Il mapper**

Il mapper incapsula le logiche di trasferimento dei dati dall'MVC al database e viceversa. Il ruolo del mapper è quello di convertire il view model in db model e viceversa, garantendo in tal modo il disaccoppiamento tra le due parti

Un oggetto di tipo mapper mantiene il riferimento ad un SqlManagerSaver, che gestisce le operazioni di salvataggio dei dati sul database; un SqlManagerLoader che invece gestisce il caricamento dei file dal database verso il view model e un SqlManagerUtility, utilizzato invece per la gestione di operazioni di utilità.

Il mapper mette a disposizione un'operazione per il caricamento di un organigramma da database, tramite il metodo "loadFromDb". Questo metodo richiede che si specifichi: il nome della company che si vuole caricare; la company stessa; l'insieme dei ruoli e dipendenti nel quale inserire i dati provenienti dal database.
Come prima cosa, tramite il sqlManagerLoader si ricavano i DbRole e i DbEmployee, poi si ottiene la DbCompany richiesta. Una volta ottenuti questi elementi si possono convertire i DbRole in Role, i DbEmployee in Employee e la DbCompany in Company; queste ultime operazioni sono gestite da metodi ausiliari presenti nella classe Mapper. Per quanto riguarda dipendenti e ruoli il mapping è molto più diretto.
Per quanto concerne invece la DbCompany devono essere convertiti al contempo tutti le unità (DbUnit presenti nella DbCompany), ruoli ammissibili (DbU2r) e ruoli assegnati (DbEmployee2UR). Il mapping deve essere effettuato ricordando anche di settare i vari parentUnit.

L'operazione di save, realizzata tramite il metodo saveToDb, richiede come parametri la company che si vuole salvare e l'insieme di ruoli e dipendenti relativi.
Come prima cosa si esegue il mapping inverso da Role a DbRole e da Employee a DbEmployee. Infine, si converte la Company in DbCompany (seguendo il procedimento inverso di prima. Bisogna tener conto della presenza di unità, ruoli ammissibili e ruoli assegnati presenti nella company, che devono essere convertiti). Una volta ottenuti i DbRole, i DbEmployee e la DbCompany si può richiedere al sqlManagerSaver di salvare quest'ultima sul database. Il save dei DbRole e DbEmployee può essere eseguito prima della DbCompany utilizzando il metodo "updateRandE" del saver.

Il mapper permette di ricavare anche il codice massimo, tramite il metodo "getMaxUnitCode" (questo per mantenere il valore di N di Unit coerente con i codici presenti sul database). Il metodo è realizzato inoltrando la richiesta al sqlManagerUtility.

Infine, sono resi disponibili i metodi per caricare e salvare separatamente ruoli e dipendenti. Il procedimento seguito è il medesimo di quello visto per quanto riguarda l'intera company.

- **AbstracteSqlManager**

Per la comunicazione col database si è definita una classe astratta che contiene i dati necessari per connettersi al database. In particolar modo questi dati sono presi dalla classe DbAccessData, nella quale questi campi come "jdbcUrl", "username" e "password" sono campi statici. Nella classe astratta, in più, è stato inserito un campo, anch'esso protetto, di tipo "Connection", saranno poi le classi concrete a inizializzare tale valore, connettendosi al database.

- **sqlManager**

Il package sqlManager si divide in due parti, una parte chiamata "manager" e una detta "dbmodel"; all'interno della prima sono presenti tutte le classi che permettono di caricare, salvare e gestire i dati da e per il database (per questo estendono tutte AbstractSqlManager), nel pacchetto "dbmodel" sono invece contenute le classi responsabili per la modellazione dei dati del db; modello che, come si è precedentemente accennato, permette di disaccoppiare la gestione del database dall'MVC.

-manager

SqlManagerLoader

All'interno di questa classe sono fornite tutte le operazioni per caricare i dati dal database.
È stata definita un'operazione "getDbRoles" utilizzata per ricavare i ruoli del database, in formato DbRole. Un'operazione simile, per ricavare i dipendenti è stata definita con "getDbEmployee". All'interno di questi metodi si utilizzano gli statement eseguiti sul database per ricavare i dati delle tabelle (si è usata la sintassi definita da MySql).

Per ricavare la DbCompany è stato definito il metodo getDbCompany, che richiede come parametro la stringa contenente il nome della company da caricare, la lista dei DbRole e dei DbEmployee. Come prima cosa si verifica se la company è davvero presente sul db, in tal caso si ricavano le unità tramite dei metodi appositi, le si assegnano i relativi ruoli ammissibili (oggetti istanza di DbU2r) e i ruoli assegnati (oggetti istanza di DbEmployee2UR). Per ricavare le unità si cercano le righe della tabella units che hanno come companyId quello della company che si sta caricando.

SqlManagerSaver

All'interno di questa classe sono inserite le operazioni necessarie per salvare una particolare company all'interno del database.
La classe offre all'esterno solo la possibilità di salvare una company o di aggiornare separatamente dipendenti e ruoli sul db, gli altri metodi sono privati e servono per realizzare i metodi pubblici sopra citati.

Per quanto concerne il salvataggio di una company, per prima cosa si verifica se questa è già presente o meno sul db, il metodo save2Db restituisce infatti un booleano per segnalare questa informazione al chiamante. L'approccio che si è scelto di seguire per salvare la company è quello del *kill and fill*; una tecnica diversa avrebbe richiesto un appesantimento eccessivo sulle operazioni da seguire sul database. Non è escluso tuttavia, in una successiva versione dell'applicazione di poter implementare questa diversa tecnica; basterebbe in tale caso andare a cambiare solo il codice presente in questa classe. Dopo aver ripulito il database dalla company corrente (avendo imposto alle chiavi esterne l'opzione DELETE ON CASCADE, basta eliminare la riga di units che rappresenta il nodo radice dell'organigramma per far sì che tutte le unità, ruoli assegnati e ruoli ammissibili vengano eliminati).
Il metodo "saveCompany" come prima cosa salva tutte le unità, eseguendo questa operazione in batch. Dopodiché, ricavati gli id delle unità inserite, si settano i parentUnitId, eseguendo questa operazione sul db sempre in batch.
"saveU2R" permette invece di salvare i ruoli ammissibili; anche questa operazione viene effettuata in batch. Si ricavano infine gli id dei ruoli ammissibili aggiunti; questi serviranno nella fase di salvataggio dei ruoli assegnati.
"saveE2UR" permette di salvare invece i ruoli assegnati; l'operazione è eseguita in batch. In questo caso non è necessario ricavare le chiavi generate.

Le operazioni per il salvataggio di ruoli e dipendenti vengono effettuate a parte, nel metodo updateRandE, che a sua volta richiama saveRoles e saveEmployees. Questi metodi, date le relative liste di DbRole e DbEmployee li caricano sul db, si ricavano le chiavi generate, assegnandole poi ai ruoli e dipendenti passati come parametri. Bisogna qui notare che per eseguire il salvataggio della company è necessario, prima, conoscere gli id dei dipendenti e dei ruoli, il mapper, come abbiamo già visto, difatti, nel salvataggio sul db esegue prima una updateRandE.

Vi è infine un metodo "getId" che permette di ricavare l'id di una qualsiasi tabella dove la riga può essere identificata tramite una stringa (come nel caso delle company o dei ruoli). In particolar modo, questo metodo è stato utilizzato per ricavare l'id della company che si sta salvando.

SqlManagerUtility
Questa classe contiene i metodi di utilità nella gestione del database. Sebbene, almeno in questa versione, è presente solo il metodo getMaxUnitCode, per ricavare il massimo valore di code presente nella table "units" (utilizzato per mantenere il valore di N della classe Unit coerente), la classe potrebbe essere arricchita di nuove funzionalità, se dovesse essere necessario.

-dbmodel

Come già anticipato, per disaccoppiare l'MVC dal database, si è utilizzato un db model. Il db model è molto simile, almeno in questa applicazione, al model già discusso in precedenza. Per garantire però una maggior facilità nel mapping dei dati del database al db model, si è dovuto introdurre un id per ogni classe. Inoltre, sempre per lo stesso motivo, si è fatto sì che ad ogni table del database corrispondesse una classe; e si è fatto sì che le relazioni tra le varie table fossero traposte nel modello in modo quanto più simile, in modo tale da ottenere una conversione il più lineare possibile.
Per tale motivo, come principale differenza dal "model", oltre alla presenza dell'id, si sono definite le classi DbU2r e DbEmpployee2UR, che corrispondono alle table nel database, rispettivamente, u2r e e2ur. Come nel database, ad un'unità è associato un numero qualsiasi di ruoli ammissibili (righe di u2r), caratterizzate dall'unità alla quale si riferiscono e dal ruolo. Un e2ur invece, definisce la relazione tra i dipendenti e i ruoli ammissibili delle unità; ogni riga è infatti caratterizzata dal ruolo ammissibile al quale si riferisce (l'u2r) e dal dipendente a cui è stato assegnato il ruolo. Un'unità ha quindi una lista di ruoli ammissibili, ovvero DbU2r e di ruoli assegnati, ovvero DbEmployee2UR.

- **fileManager**

Il pacchetto fileManager contiene invece le classi utilizzate per gestire i file. Questo è diviso in tre parti, il pacchetto abstractFactory che contiene un'implementazione del pattern omonimo, il pacchetto converter che contiene le classi responsabili per la gestione del caricamento/salvataggio dei file; vi è infine il ConverterMediator che svolge il ruolo di mediatore tra l'entità richiedenti le operazioni sui file e chi effettivamente esegue tali operazioni, permettendo di mantenere le due parti disaccoppiate.

-ConverterMediator

Il converterMediator agisce da intermediario tra chi richiede la gestione dei file e le entità che sono effettivamente in grado eseguire queste operazioni, ovvero, i converter. Il mediator mantiene il riferimento a un converterFromFile e ad un ConverterToFile, in più, mantiene il riferimento a una ConverterFactory, alla quale richiederà i Converter in fase di costruzione.
Le operazioni che il ConverterMediator mette a disposizione sono la saveFile e la loadFromFile, in entrambi i casi l'oggetto richiede che venga specificato il path del file che si sta considerando.
Nel save2File, prima si chiede al saver di convertire il db, poi si può usare il metodo saveToFile del saver per eseguire l'operazione.
Per quanto concerne il metodo loadFromFile, questo va semplicemente a inoltrare la richiesta al loader, sul quale è richiamato il metodo convertFromFile.

-abstractFactory

Seguendo il pattern si è definita una interfaccia ConverterFactory che definisce due factory method, uno per ogni prodotto che la factory deve realizzare: un ConverterFromFile e un ConverterToFile. Poiché si è deciso di salvare i file in formato Json si è creato una apposita JsonFactory che crea i converter relativi al formato Json. La factory in questione (che implementa ConverterFactory) è un valore di una enumeration, si è infatti deciso di inserire le varie factory che si andranno a realizzare nella enum "ConverterFactories" realizzando così il pattern Singleton, permettendo quindi di avere una sola factory per tipologia e permettendo di avere un unico punto di accesso alle varie istanze, ovvero, l'enumeration ConverterFactories.
Per ottenere una nuova factory, che permetta di creare dei converter per altri formati basterà aggiungere un nuovo valore alla enumeration, che restituisca i converter voluti. Si noti che con questa tecnica siamo in grado di isolare le classi concrete; il mediator, infatti, che utilizza la factory, non sa di che tipo concreto sono gli oggetti Converter che sta utilizzando, per questo motivo siamo in grado di riutilizzare lo stesso ConverterMediator indipendentemente dal formato in cui stiamo gestendo i file. Per ottenere una diversa gestione dei file (in altri formati) basterà cambiare la factory utilizzata dal ConverterMediator.

-converter

Nel package conveter sono definite le interfacce: ConverterFromFile e il ConverterToFile, la prima offre solo il metodo convertFromFile, al quale deve essere passato il path; la seconda offre convertDb e poi saveToFile.

Nel sottopackage json invece, è fornita un'implementazione alle due interfacce; come già detto, i file sono prodotti in formato json. Poiché entrambi queste classi dovranno operare col database, estendono abstractSqlManager.

Converter2Json

Converter2Json implementa l'interfaccia ConverterToFile.
L'oggetto presenta un campo d'istanza di tipo JSONObject. In questo oggetto sarà inserito il contenuto del database.
Il metodo convertDb esegue la conversione del contenuto del database in un JSONObject. Per eseguire questa operazione, per ogni table del database viene creato un metodo apposito (privato) che estrae i dati e li inserisce in dei JSONArray, a loro volta inseriti in dei JSONObject. Una volta ottenuto un JSONObject per ogni table, viene creato un JSONArray, vengono aggiunte la table; l'array viene infine inserito in un ulteriore JSONObject utilizzando come chiave "organizationalcharts". Questo oggetto sarà proprio la variabile d'istanza "*data*".
Il metodo saveToFile crea un file al path passato e ci scrive sopra la rappresentazione in forma di stringa del JSONObject "data".

ConverterFromJson

Questa classe mette a disposizione le operazioni per caricare un file in formato Json e immetterne il contenuto sul database.
L'interfaccia pubblica dell'oggetto è composta solo dal metodo convertFromFile, che richiede il path dove trovare il file.
Poiché si suppone che il file sia in formato Json, si può effetturare un parsing su di esso, usando un oggetto apposito di tipo JSONParser. Se il parsing fallisce l'operazione di caricamento non può continuare.
Una volta effettuato il parsing del file il contenuto attuale del database viene eliminato (vi è il metodo removeAll). Per evitare però problemi dovuti a file corrotti, viene temporaneamente tolta la autocommit. Nel caso si dovesse verificare un errore nelle operazioni verrà eseguita la rollback, altrimenti, alla fine del caricamento si esegue una commit e l'autocommit viene nuovamente abilitata.

Per il caricamento viene fornito un metodo apposito di load per ogni table del database. L'inserimento dei dati è abbastanza lineare quasi per tutte le table, i dati vengono presi dal file e inseriti sul database (gli inserimenti vengono effettuati in batch per motivi di efficienza). L'unica eccezione vale per le unità; essendo infatti queste provviste del riferimento al padre (relazione garantita dal vincolo di chiave esterna) è necessario, per ogni company, inserire le unità a partire dall'unità padre e quindi a scendere nella struttura dell'organigramma. Per questa ragione viene prima effettuata una conversione delle unità dal file al formato DbUnit, e poi, per ogni company, si parte dalla radice e si

immettono le unità nel database (le operazioni di inserimento vengono sempre effettuate in batch).

-Utilizzo di jdbc

Per la comunicazione col database si è usata la libreria jdbc. Si è usato come dbms MySql. L'utilizzo di jdbc per gestire il database prescrive innanzitutto l'instaurazione di una connessione col database stesso.

```
try {
    connection = DriverManager.getConnection(jdbcUrl, username, password);
} catch (SQLException throwables) {
    throwables.printStackTrace();
}
```

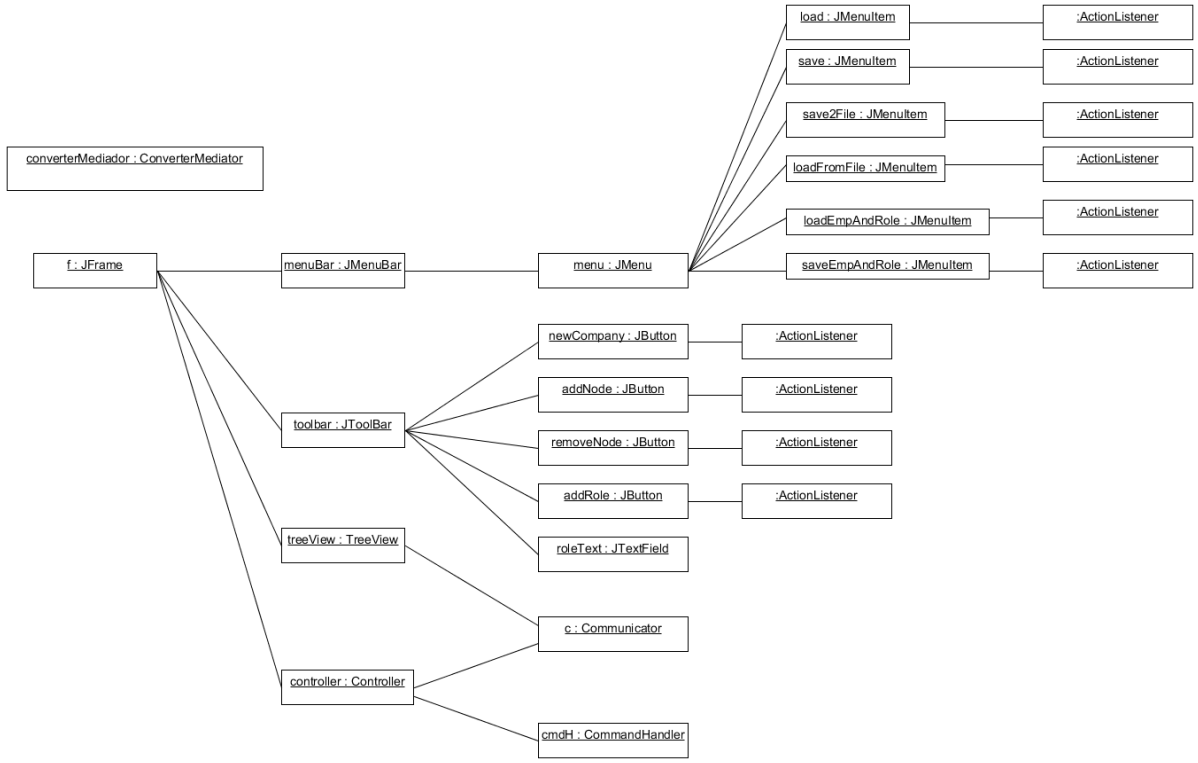
L'instaurazione della comunicazione necessita quindi la conoscenza dell'url jdbc, dell'username e della password per il database. Una volta eseguito l'accesso è possibile, tramite la creazione di statement e preparedStatement (ottenibili sempre tramite l'utilizzo di connection), scrivere (come stringhe) delle query o statement di aggiornamento (seguendo la sintassi del dbms scelto) come se ci si stesse interfacciando direttamente col database stesso.

Le informazioni per l'accesso al database sono contenute nella classe DbAccessData.

- **GraphicTest**

La classe GraphicTest contiene il main che permette all'applicazione di essere eseguita. Il main permette di creare e aggiungere alla finestra principale il Controller e il TreeView. Oltre a questo, crea il Communicator e il CommandHandler e li passa al Controller.
GraphicTest è anche responsabile per la creazione dei pulsanti e MenuItem nella parte alta della finestra dell'applicazione, ovvero, per ciò che riguarda i menu item: "Load company from db", "Save company to db", "Save db content to file", "Load db content from file", "Load employees and roles from db", "Save employees and roles to db", e per quanto concerne i pulsanti: "New company", "Add child", "Remove node" e "Add role" (con la relativa TextArea). Per ognuno di questi elementi GraphicTest aggiunge un ActionListener che, eseguendo il metodo actionPerformed affida il comando relativo al pulsante/menuItem cliccato all'apposito commandHandler (i.e. viene creato il relativo comando specifico che viene poi passato al commandHnalder). Il tutto è aggiunto alla finestra e mostrato all'utente.

Può essere visualizzato l'object diagram di GraphicTest al momento dell'avvio dell'applicazione.

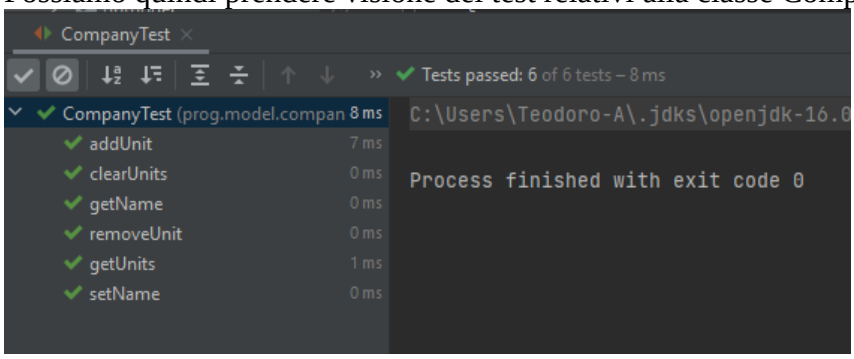


Il converter mediator non è ancora legato ad alcun oggetto, in quanto non è stato eseguito alcuno comando per salvare o caricare file. Ad ognuno dei pulsanti o menuItem è associato un action listener (realizzato nel codice tramite una lambda expression), il quale, al momento del click sull'elemento, nella maggior parte dei casi creerà un command e lo affiderà al commandHandler per eseguirlo.

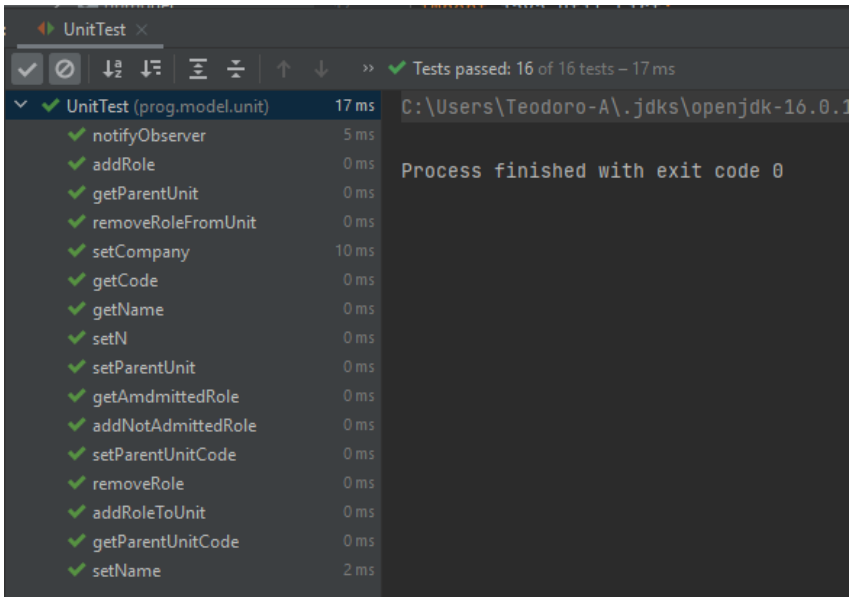
-Test

Si è deciso di eseguire dei test sull'applicazione realizzata, in particolar modo, sul package prog::model, contente le classi utilizzate per modellare l'organigramma su Java; si tratta, quindi, di un modulo molto importante ai fini del funzionamento dell'applicazione.
Il testing è stato effettuato utilizzando JUnit4.
Si sono testate le principali funzionalità offerte dalle varie classi.

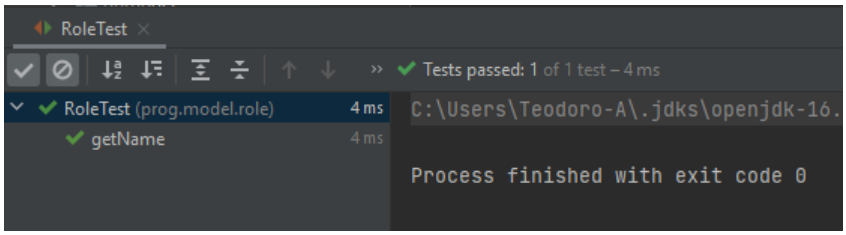
Possiamo quindi prendere visione dei test relativi alla classe Company.



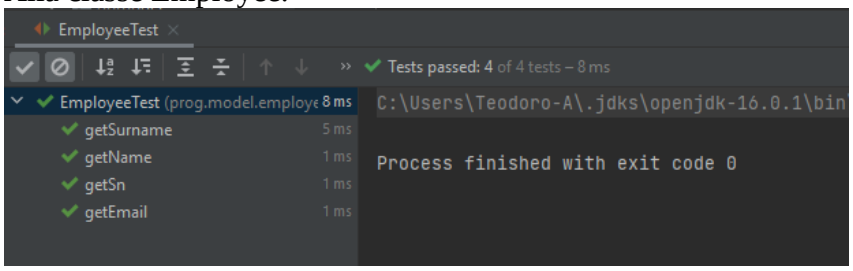
Relativamente alla classe Unit.



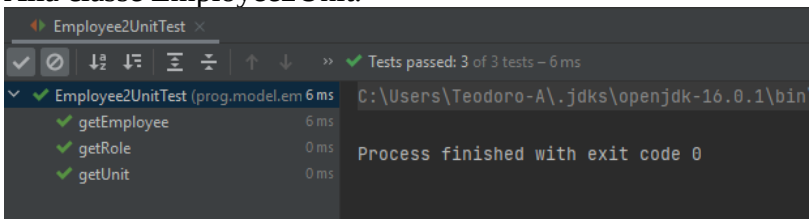
Alla classe Role.



Alla classe Employee.



Alla classe Employee2Unit.



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

- **Requisiti funzionali**

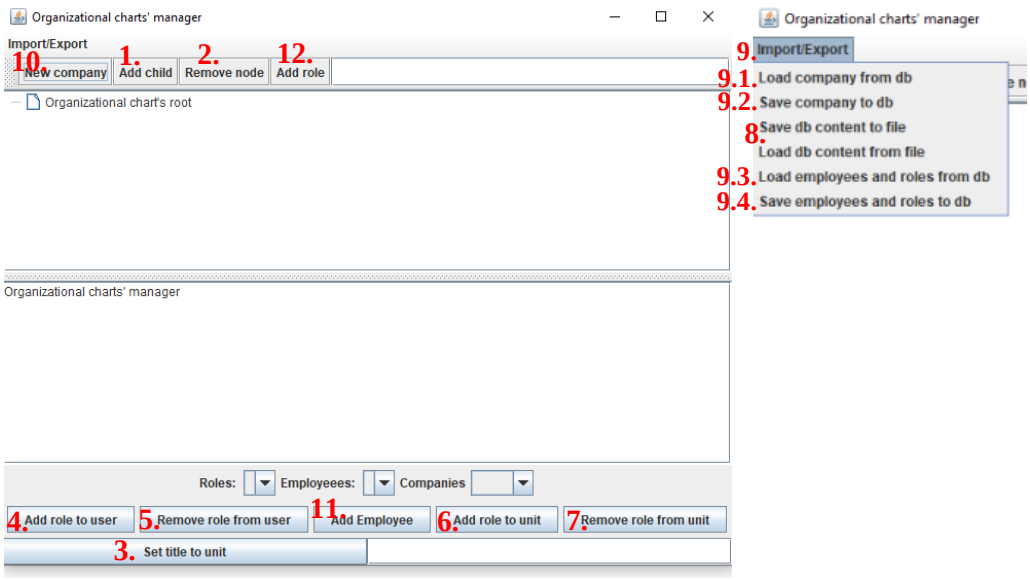
Come abbiamo visto sia nella HLD che nella LLD, per ognuno dei requisiti funzionali individuati in fase di analisi e specifica dei requisiti, è stato fornito all'utente un modo per eseguire tali funzioni; ciò è stato realizzato tramite pulsanti o menuItem.

1. Aggiunta di nuove unità
2. Rimozione di unità esistenti
3. Aggiornamento nome unità
4. Aggiunta ruolo ammissibile ad un dipendente, relativamente ad un'unità
5. Rimozione ruolo ammissibile ad un dipendente, relativamente ad un'unità
6. Aggiungere un ruolo ammissibile ad una unità
7. Rimuovere un ruolo ammissibile ad una unità
8. Salvataggio su file
9. Comunicazione con dbms

9.1 Caricare un organigramma dal database
9.2 Salvare un organigramma sul database
9.3 Caricare ruoli e dipendenti dal database
9.4 Salvare ruoli e dipendenti sul database

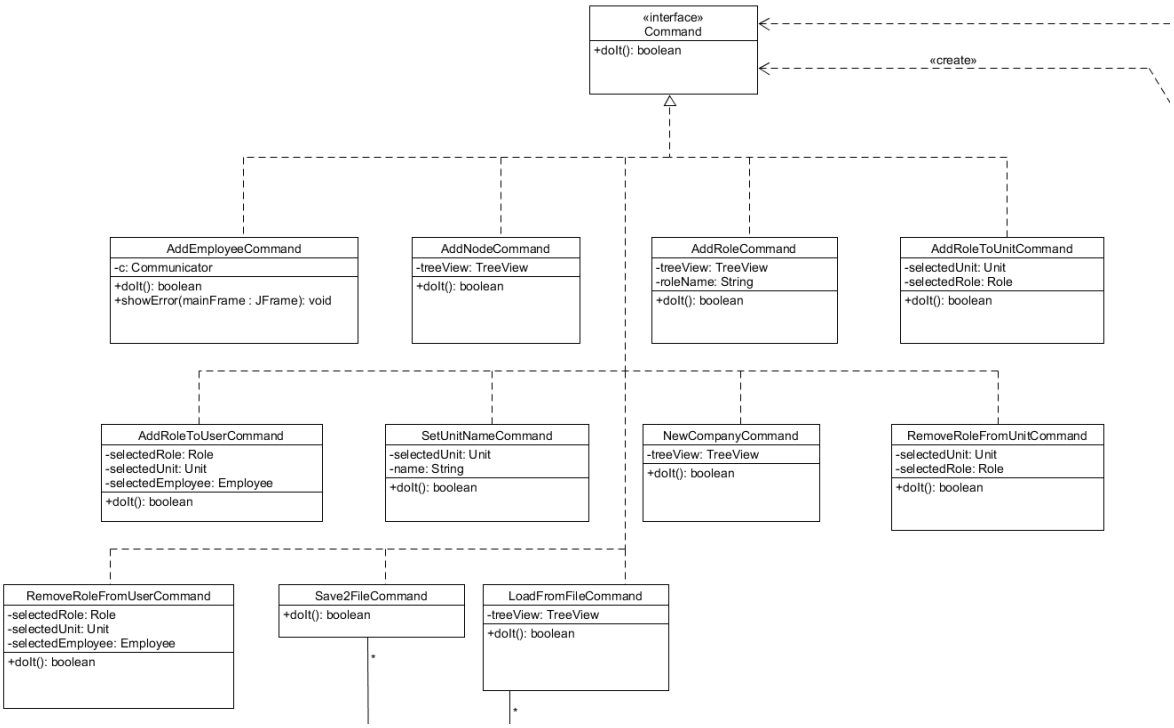
10. Creazione di nuovi organigrammi
11. Creare nuovi dipendenti
12. Creare nuovi ruoli

Possiamo quindi osservare come queste funzionalità sono garantite, almeno graficamente all'utente tramite l'interfaccia grafica.



Ovviamente, salvare i dati su file, richiede anche un meccanismo che ci permetta di leggerli, il punto 8 è stato quindi esteso tra la possibilità di salvare il contenuto del database su file e quello di caricare il contenuto del database da un file.

Come si è visto nella HLD e poi si è approfondito durante la LLD, gran parte di queste funzionalità sono quasi sempre state realizzare tramite l'ausilio di oggetti che implementano il pattern Command.



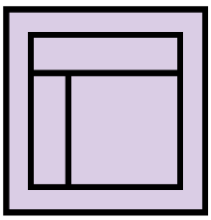
Dal diagramma delle classi è stato infatti possibile osservare che, quasi ad ognuna di queste classi corrisponde una delle funzionalità sopra elencate. Nei casi in cui l'operazione è risultata più banale, oppure non richiede la presenza di un comando si è deciso di realizzare l'operazione in modo diverso. Come, ad esempio: la rimozione di un nuovo nodo. Questa operazione viene demandata direttamente alla TreeView.
In più, si è visto che, in particolar modo per la gestione dei file, è stato dedicato un intero package/modulo (il fileManager), come anche per la comunicazione col database (il sqlManager), all'interno dei quali, come si è visto, sono realizzate le funzionalità richieste.
La gestione dinamica dell'organigramma (con l'aggiunta/rimozione/aggiornamento degli elementi che lo costituiscono) è stata realizzata tramite l'applicazione del pattern MVC, ed anche del pattern Observer, che è risultato molto utile per la gestione della parte grafica e per il suo aggiornamento. L'MVC ci ha permesso anche di dare la possibilità all'utente di eseguire le operazioni di modifica desiderate sul modello visualizzato dalla view. L'MVC è stato anche utile per gestire la rappresentazione grafica dell'applicazione; come visto in particolar modo nella LLD, infatti, sia il controller che la view estendono JPanel, sono stati quindi aggiunti direttamente alla finestra principale dell'applicazione.

Infine, da come visto nella LLD e nella HLD, la modellazione dell'organigramma è stata effettuata in modo tale da permettere una maggiore facilità nella comunicazione col database, sebbene queste due entità siano disaccoppiate dal meccanismo di conversione viewModel-dbModel.

- **Requisiti non funzionali**

È evidente che, ottenendo l'intercomunicazione della main application con il database, siamo stati in grado di far sì che i dati inseriti dall'utente potessero essere memorizzati all'interno di un dbms. Questa scelta progettuale non solo ha permesso un miglioramento in termini di affidabilità dell'applicazione, ma è stata anche di supporto per quanto concerne la possibilità di salvare e caricare file, in quanto si è così potuto utilizzare lo schema del database per generare dei file Json.
Per quanto concerne invece l'utilizzabilità dell'applicazione e le sue prestazioni, almeno in questa fase di rilascio si è deciso di non dare priorità all'efficienza, ma di porre l'attenzione sul raggiungimento dei requisiti funzionali, in modo tale da fornire all'utente una versione quanto più completa del sistema software richiesto. L'utilizzabilità è una proprietà soggettiva, quindi non è possibile offrire un esito certo sul raggiungimento di questo obiettivo, tuttavia, la presenza dell'interfaccia grafica rappresenta un buon punto di partenza per permettere una gestione user friendly dell'applicazione.
Per quanto concerne l'utilizzabilità nei termini di ritardi osservati durante l'utilizzo dell'applicazione non ne sono stati osservati; sarà tuttavia possibile ottenere risultati migliore dopo un'opportuna opera di ottimizzazione. La suddivisione del sistema in moduli dà la possibilità di eseguire queste operazioni senza dover stravolgere la struttura generale del progetto; una conseguenza del principio di modularità sta proprio nel fatto che è possibile modificare il sistema tramite il cambiamento di una sua parte, ovvero, tramite la variazione dei dettagli implementativi dei singoli moduli. Poiché inoltre questi sono stati progettati in modo tale da non permettere ad eventuali client si dipendere da dettagli interni di tali componenti, l'ottimizzazione potrà essere effettuata senza dover cambiare radicalmente le componenti software che dipendono da quelle soggette al cambiamento.

Appendix. Prototype



Il sistema implementato consiste in un'applicazione in grado di gestire dinamicamente la struttura di un'organizzazione tramite la realizzazione di organigrammi aziendali. Ciò che è stato realizzato non riguarda solo la definizione di nuovi nodi e relazioni gerarchiche tra le unità, ma anche l'introduzione dei ruoli ammissibili per le unità e l'assegnamento di tali ruoli ai dipendenti dell'organizzazione. Come richiesto, è stato anche implementato un sistema, facente parte dell'applicazione, in grado di salvare (e caricare) i dati degli organigrammi da file.

Supponendo di lavorare su un sistema destinato ad un ambiente aziendale, si è creduto utile includere anche delle funzionalità per la comunicazione dell'applicazione con un dbms.

Come già ampiamente discusso nella HDL e LLD, le funzionalità sopra descritte sono state rese disponibili tramite un'interfaccia grafica. Per ognuno dei requisiti funzionali individuati è stato realizzato una componente interattiva dell'interfaccia in grado di prendere in carico la richiesta del client ed eseguire la data operazione.

L'interfaccia grafica permette di definire nuove unità (anche rimozione di unità esistenti e modifica del nome dell'unità), nuovi ruoli, nuovi dipendenti, e, sebbene non richiesto esplicitamente nei requisiti informali, anche di creare nuovi organigrammi (rendendone disponibile quindi, la gestione di più di uno).

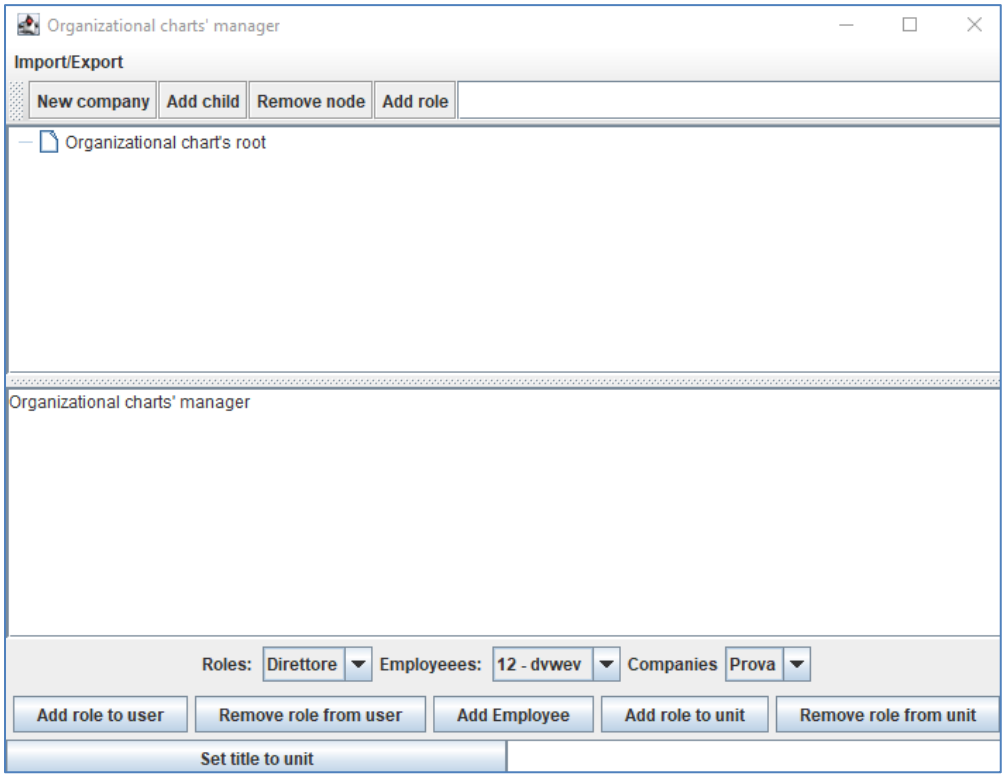
È stata anche resa possibile la navigazione interattiva sulla struttura, basterà infatti cliccare l'unità interessata sull'organigramma visualizzato per prenderne visione delle informazioni contenute.

Tramite appositi pulsanti è stato reso possibile anche aggiungere ruoli ammissibili, assegnare ruoli, rimuovere ruoli ammissibili e de-assegnare ruoli ai dipendenti.

La richiesta sulla comunicazione col database è stata resa possibile all'interno del menu a tendina "Import/Export". All'interno sono state inseriti i comandi per gestire i file, caricare e salvare organigrammi da e per il database (oppure, soltanto dipendenti e ruoli).

Queste funzionalità sono state ovviamente realizzate tramite l'utilizzo di diversi design pattern; questi strumenti garantiscono delle soluzioni già collaudate per problemi ricorrenti, ed aiutano anche a rendere il codice realizzato riutilizzabile.

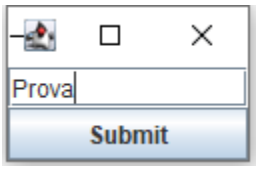
Di seguito è mostrata l'interfaccia grafica dell'applicazione.



Di seguito sono mostrati alcuni esempi di uso dell'applicazione.

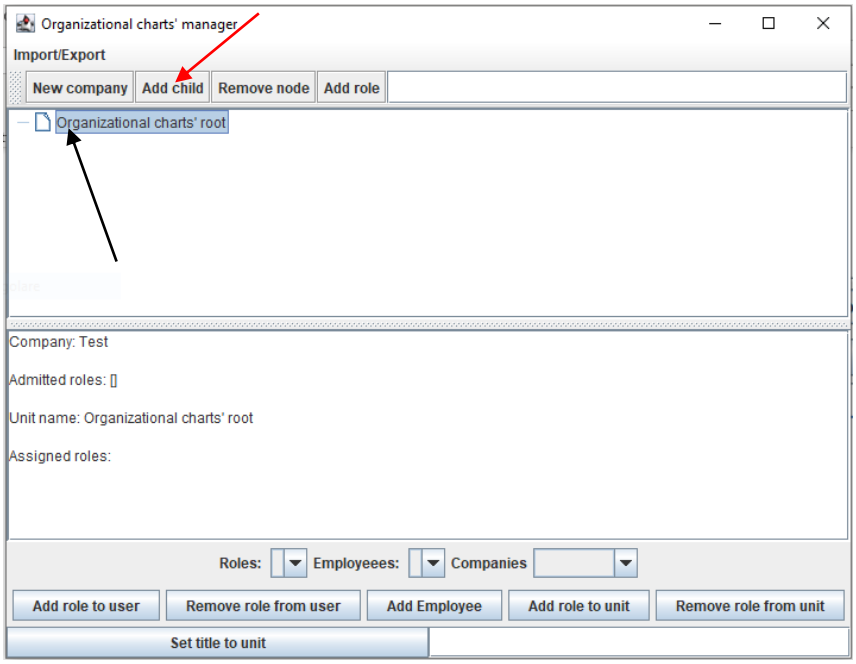
Poiché desideriamo salvare il nostro organigramma su database, per prima cosa creiamo una nuova company cliccando sul pulsante "New company".

Ci viene mostrata la seguente finestra, dove possiamo inserire il nome desiderato (diverso da WorkBench).

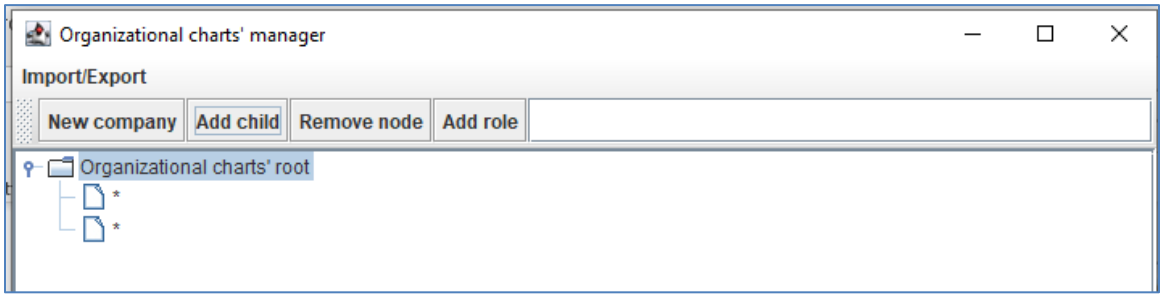


Il nuovo organigramma è pronto per essere modificato.

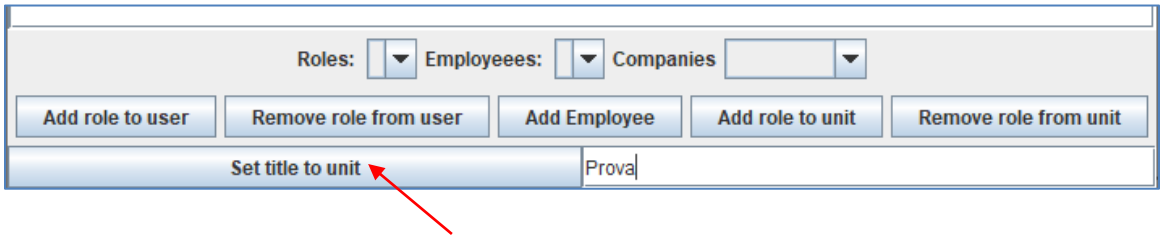
Selezionando una particolare unità è possibile aggiungere dei nodi figli premendo "Add child".



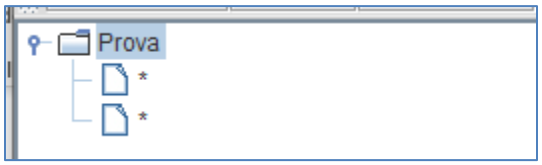
Difatti, cliccando:



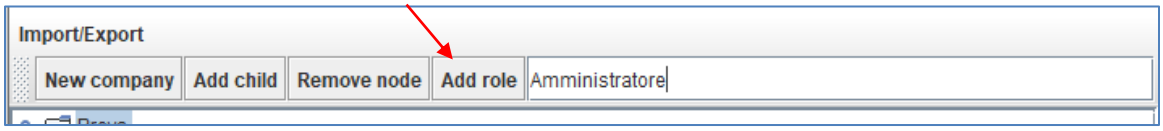
Inserendo un nome nell'area di testo in fondo alla finestra e selezionando Set title to unit è possibile cambiare il nome dell'unità selezionata.



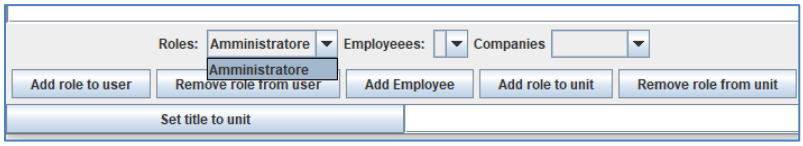
Dopo aver cliccato:



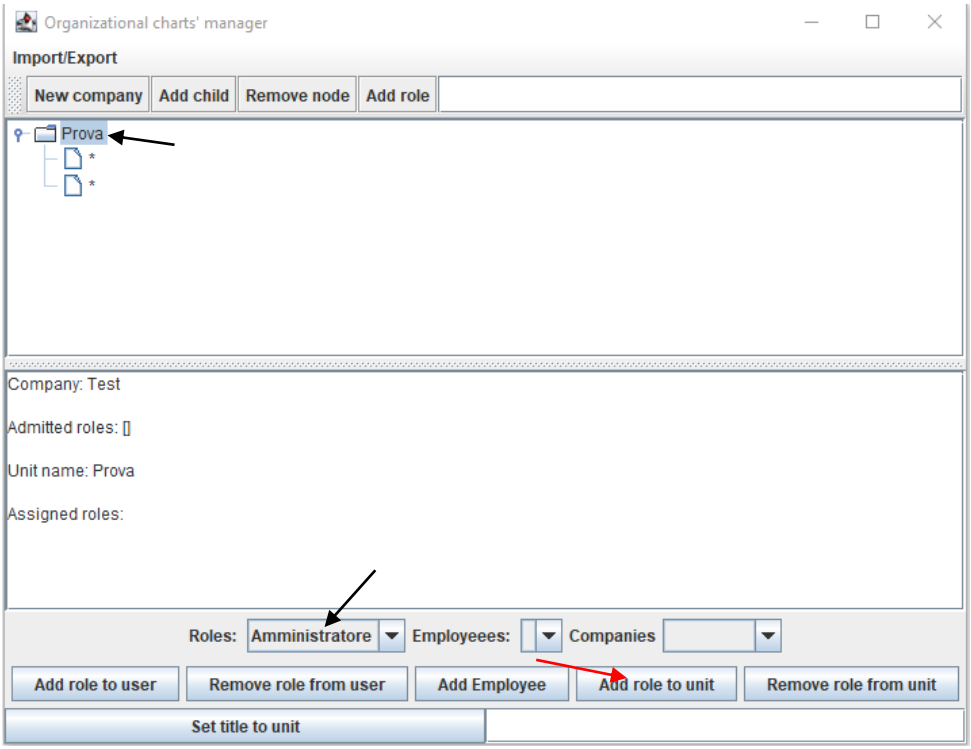
Possiamo definire un nuovo ruolo inserendo il nome nella text area in alto nella finestra dell'applicazione e cliccando il pulsante "Add role".



Il nuovo ruolo viene aggiunto alla lista dei ruoli disponibili.

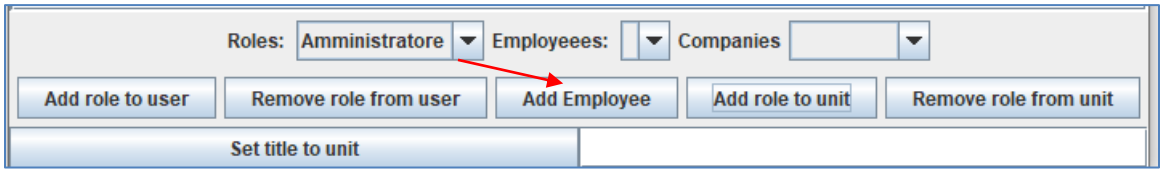


Possiamo quindi aggiungere un ruolo ammissibile ad un'unità; selezionando il ruolo e l'unità.

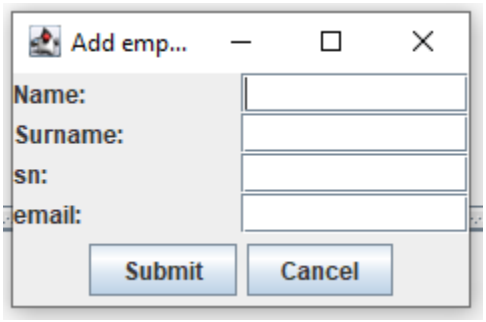


La lista dei ruoli ammissibili dell'unità viene quindi aggiornata.

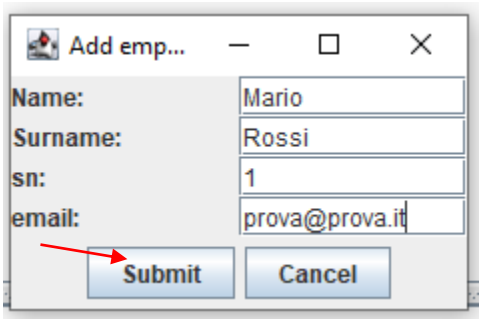
Possiamo quindi creare un nuovo dipendente.



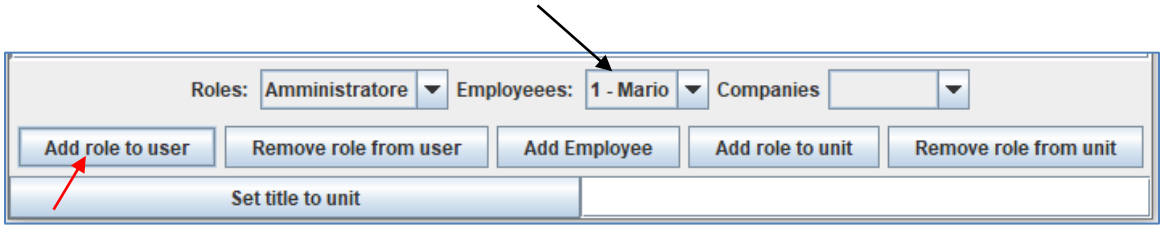
Il pulsante apre una finestra nella quale possiamo inserire i dati del nuovo dipendente.



Inseriamo i dati e clicchiamo submit.



La lista dei dipendenti disponibili viene aggiornata.

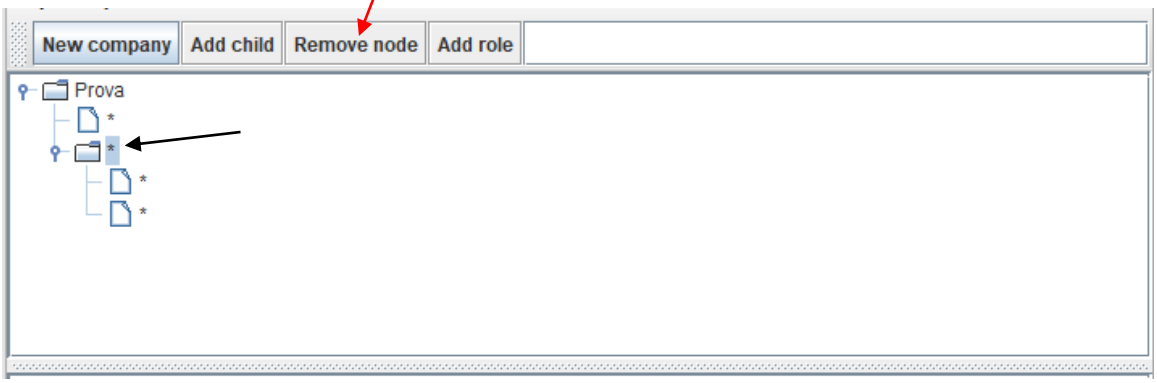


Possiamo quindi assegnare il ruolo di “Amministratore” al dipendente “Mario” in corrispondenza dell’unità che abbiamo già selezionato.

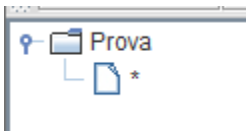
I ruoli assegnati nell'unità vengono aggiornati.

Se provassimo ad assegnare il ruolo in un'unità in cui questo non è stato aggiunto tra i ruoli ammissibili l'aggiunta non sarebbe possibile.

Volendo eliminare un nodo.



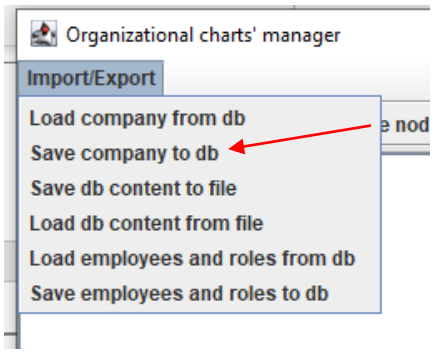
Il nodo e i suoi figli vengono quindi eliminati



Possiamo ora, quindi, salvare l'organigramma creato su database.

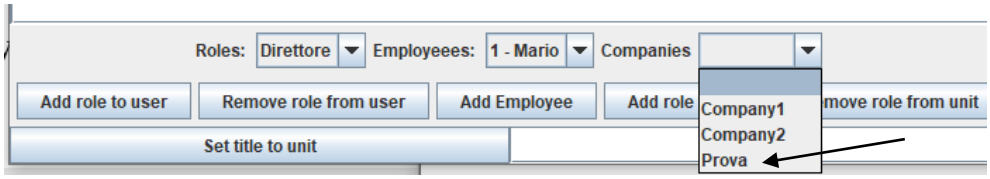
Andando nel menu Import/Export è possibile visualizzare le seguenti operazioni.

Clicchiamo quindi su “Save company to db”



Viene mostrato un messaggio che ci segnala l’avvenuto caricamento.

La lista delle company disponibili su database viene aggiornata.



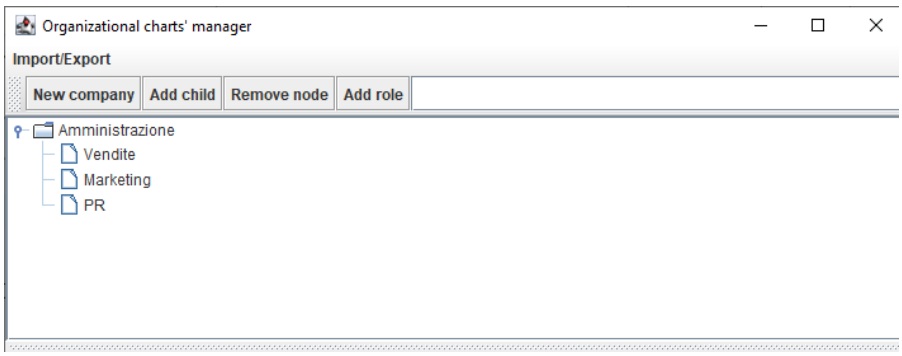
Andando sul dbms è infatti possibile visualizzare nella table units:

	id	name	companyId	parentUnit	code
▶	9	Prova	3	NULL	11
	10	*	3	9	12

Possiamo caricare un organigramma da database.

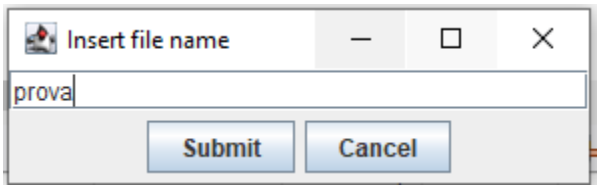
Scegliamo, ad esempio, Company1.

L’organigramma viene caricato e mostrato.

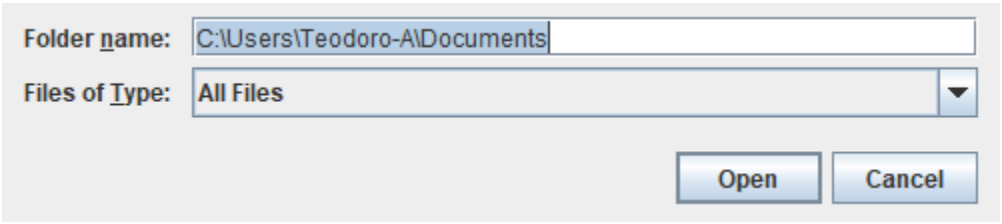


È anche possibile salvare il contenuto del database su di un file, selezionando l'operazione in Import/Export "Save db content to file".

Scegliamo il nome del file:



E poi la cartella nella quale sarà salvato.



Il file json contenten i dati del db viene quindi creato.

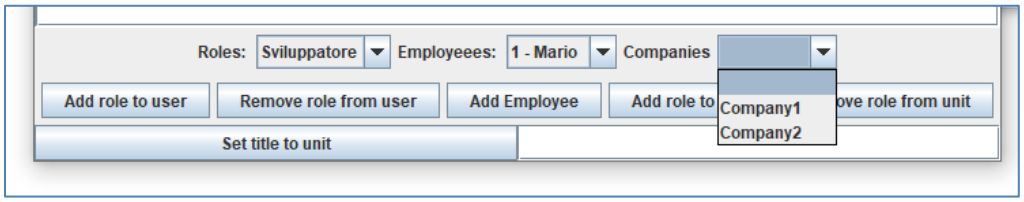
Sarà possibile, in un secondo momento, prendere i dati dal file e caricarli sul database.

Ad esempio, prima dell'esecuzione di questa prova, era stato salvato un file contente lo stato del database precedente al salvataggio della company "Prova".

Possiamo quindi caricate questo file usando il comando "Load db content from file".

Ci viene chiesto di selezionare il file

Una volta scelto il suo contenuto verrà caricato.



Sono infatti disponibili le company precedentemente create.