Child Care Tech

Prova Finale di Ingegneria del Software

Realizzato da:

VITTORIO DENTI
RAFAEL MOSCA

Laurea in Ingegneria Informatica
POLITECNICO DI MILANO

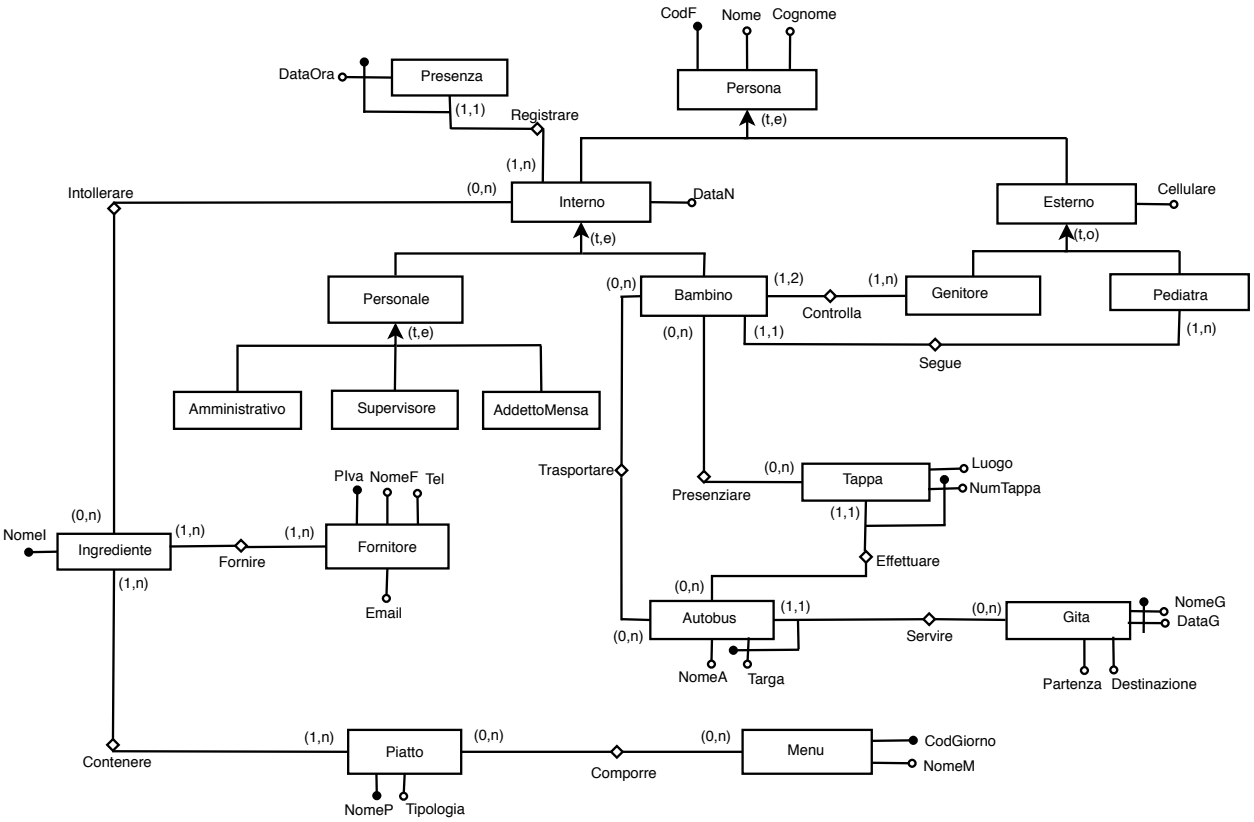
Progetto realizzato sotto la supervisione del
PROF. ALESSANDRO CAMPI.

MAGGIO 2018

INDICE

CAPITOLO 1

DATABASE



Il diagramma ER riportato è stato progettato volendo supportare tutte le funzionalità desiderate per l'applicazione "Child Care Tech": a livello di anagrafica si vogliono memorizzare tutti i dati dei soggetti interagenti con il sistema; a livello di funzionalità per la gestione dei pasti si vogliono invece poter memorizzare le informazioni riguardanti i diversi menu, i piatti e le intolleranze alimentari degli operatori addetti alla supervisione dei bambini e dei bambini stessi. Infine, per quanto riguarda la gestione delle gite, si è deciso oltre che supportare funzionalità legate alla pianificazione delle gite e tutte le opzioni a esse connesse, anche di tener traccia delle varie tappe all'interno di un viaggio e memorizzarle insieme alle presenze dei bambini in modo da capire agevolmente quale bambino fra quelli iscritti al momento dell'appello non risulti presente alla fermata.

Tutte le associazioni fra le entità, in riferimento all'autoesplicatività del diagramma ER, risultano ben evidenti, tuttavia nel seguito si riportano le ipotesi progettuali alla luce delle quali si è giunti alla realizzazione di tale diagramma:

- L'applicativo, in seguito al login, verrà customizzato in base ai permessi associati a tale operatore: l'utente assegnato alla mensa potrà occuparsi solamente della gestione della mensa e delle allergie, l'utente assegnato alla supervisione dei bambini potrà gestire le gite ed infine l'utente amministrativo, oltre a far ciò che gli altri utenti già fanno, può registrare nuovi bambini, nuovi fornitori e nuove persone entranti nello staff;
- Al fine di facilitare la scalabilità dell'applicazione si è deciso di associare a ciascun ingrediente il fornitore da cui proviene: al momento tale funzionalità non è richiesta e si è giudicata non rilevante per il modo in cui è strutturata correntemente, tuttavia in fase di progettazione è già stata considerata prevedendo un'eventuale richiesta futura;
- Tramite l'entità "Presenza" si tiene traccia dei bambini presenti e dei membri dello staff al momento presenti;
- A livello di codice Java si è effettuata una mappatura 1:1 fra le entità del database e le classi Java: in questo modo si facilitano la manutenibilità e la scalabilità dell'applicazione nel corso del tempo.

Entrando nello specifico, le query sono state realizzate usando dei `PreparedStatement` anziché degli `Statement` normali, poiché i primi evitano il problema del SQL Injection.

Si è dunque usato:

```
"INSERT INTO PERSONALE"
 + "(CodF, Nome, Cognome, Username, Password, DataN, TypeFlag) " +
        "VALUES(?,?,?,?,?,?,?);";

stmt = conn.prepareStatement(query);

stmt.setString(1,s.getCodiceFiscale());
stmt.setString(2,s.getNome());
stmt.setString(3,s.getCognome());
stmt.setString(4,s.getUsername());
stmt.setString(5,s.getPassword());
stmt.setString(6,s.getDataNascita());
stmt.setString(7,s.getTipo());
```

anziché

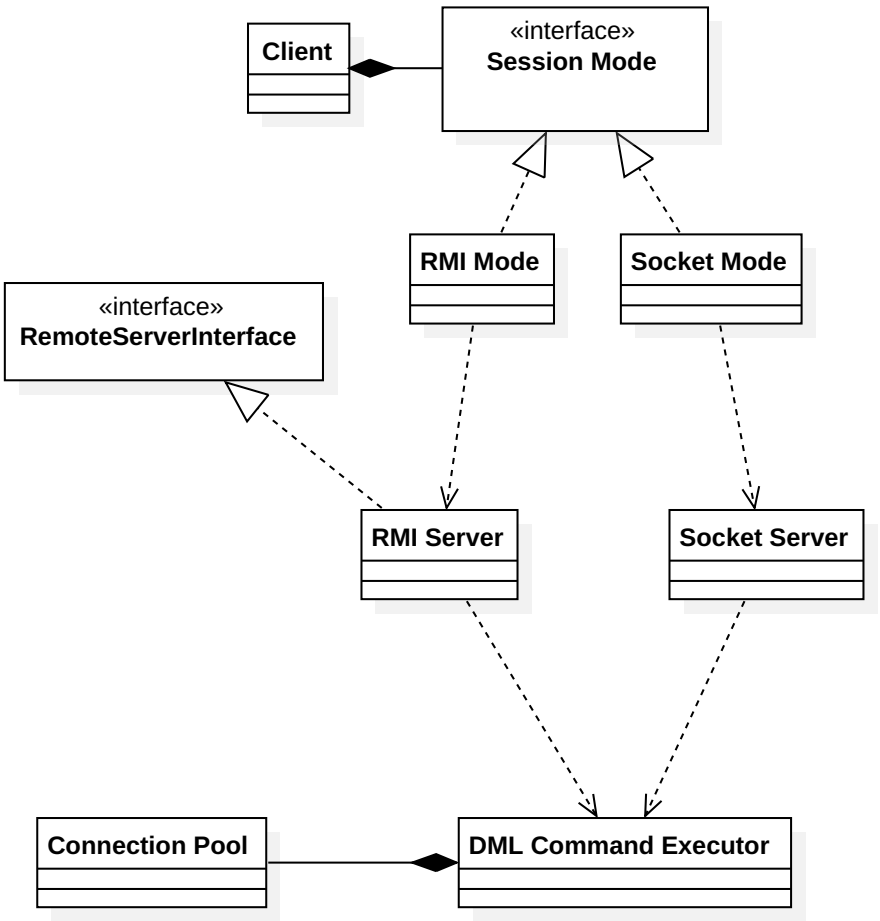
```
"INSERT INTO PERSONALE"
 + "(CodF, Nome, Cognome, Username, Password, DataN, TypeFlag) " +
 "VALUES('" + s.getCodiceFiscale() + "','" + s.getNome() + "','" +
 s.getCognome() + "' ,'" + s.getUsername() + "','"
 + s.getPassword() + "','" +
 s.getDataNascita() + "','" + s.getTipo() + "');";
```

Inoltre, come doppio controllo, si è deciso di prima dell'esecuzione della query utilizzare il metodo replaceAll di String per sostituendo l'apice per un carattere molto simile a un apice, la virgoletta singola.

```
string.replaceAll("'", "’");
```

CAPITOLO 2

CLASSI



Il diagramma delle classi riportato mette in evidenza la struttura a strati che si è deciso adottare per l'applicativo. In questa struttura ogni strato, anche noto come layer, è indipendente dal resto dei moduli del programma al fine di realizzare un basso accoppiamento fra essi. La progettazione è stata principalmente orientata al garantire sia la riusabilità dei moduli che gli aggiornamenti modulari; ad esempio se si dovesse decidere di cambiare interfaccia grafica questa potrebbe essere sostituita senza dover alterare la struttura e il codice degli altri moduli, lo stesso qualora si inventasse una nuova modalità di connessione o si decidesse di cambiare tipologia di database.
L'intefaccia Session Mode e le rispettive implementazioni realizzano il layer di connessione.
L'implementazione RMI Mode non fa altro che invocare i metodi dell'intefaccia remota.

```
server = (RemoteServerInterface) Naming.lookup("rmi://127.0.0.1/stub");
...
//dentro all'implemetazione del Metodo di session Mode
server.nomeMetodo( parametri );
```

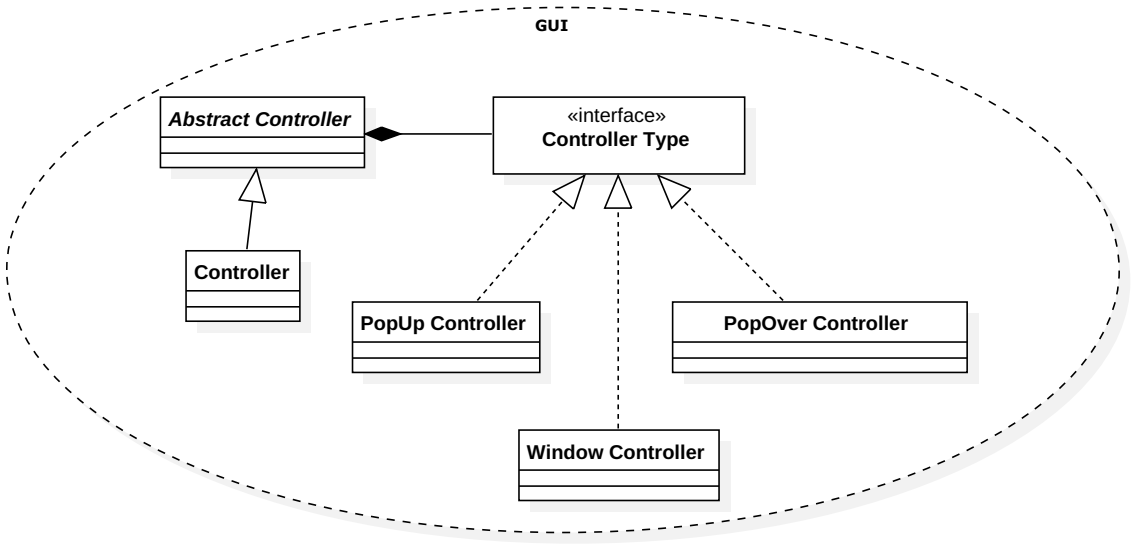
Socket Mode realizza la scrittura dei comandi e parametri sullo stream in output e restituisce il risultato dallo stream in input.

```
socket = new Socket(address, PORT);
socketObjectOut = new ObjectOutputStream(socket.getOutputStream());
            socketObjectIn = new ObjectInputStream(socket.getInputStream());
...
//dentro all'implemetazione del Metodo di Session Mode
socketObjectOut.writeObject( "StringaComando" );
socketObjectOut.flush();
socketObjectOut.writeObject(parametro1);
socketObjectOut.flush();
...
return socketObjectIn.readBoolean();
```

La classe DML (Data Manipulation Language) Command Executor è il layer dedicato all'accesso alla base di dati, in particolare esso contiene dei metodi che eseguono i seguenti step:

1. Ottengono una connessione dal Connection Pool
2. Eseguono le query
3. Rilasciano la connessione
4. Elaborano il result set
5. Inviano un oggetto, una lista di oggetti o un valore di ritorno opportuno.

Per quanto concerne l'interfaccia grafica si è utilizzato JavaFx e si è seguito il paradigma MVC (Model View Controller). Ogni scena ha dunque associato un controller, inoltre nel nostro caso ogni controller utilizzato estende la classe astratta Abstract Controller, la quale fattorizza metodi comuni a tutti i controller (si noti che è una classe astratta al fine di evitare l'istanziazione della classe, in quanto non avrebbe alcun senso).



Ogni controller nella GUI ha un attributo Controller Type che specifica la tipologia di finestra e viene inizializzato nel metodo `Initialize` che è una sorta di costruttore per i Controller (questo perché non si invoca mai il costruttore di una classe controller).

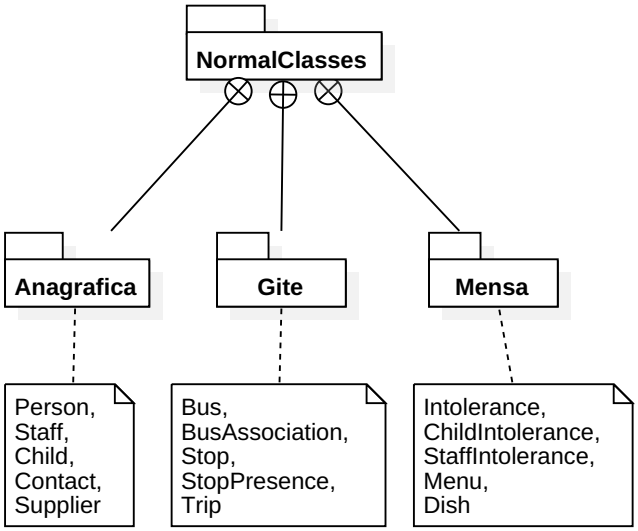
```
controllerType = new PopUpController();
```

Questa struttura è molto simile al design pattern *Strategy* con l'unica differenza che l'attributo non può cambiare a runtime in quanto non avrebbe senso farlo.
Per esempio la chiusura della finestra avviene:

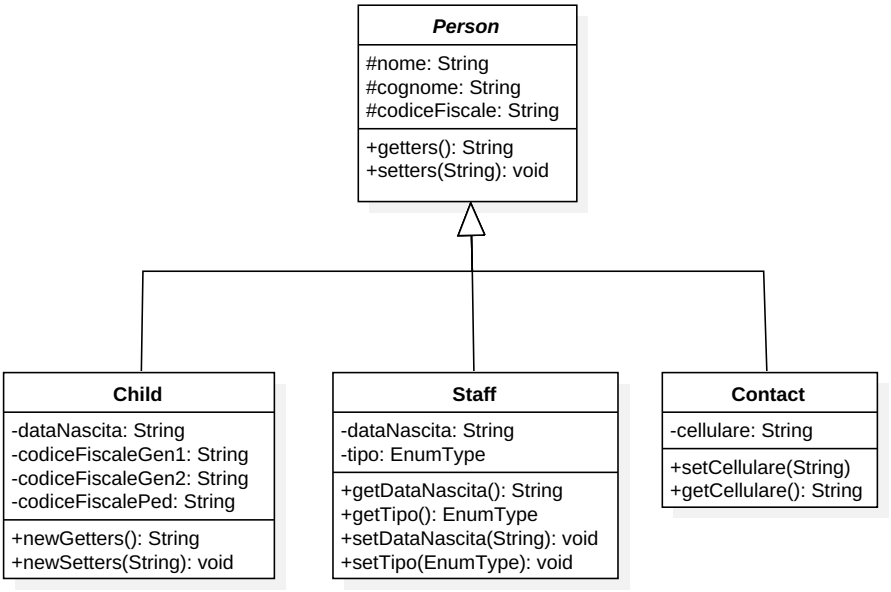
```
controllerType.close();
```

Il metodo da eseguire viene scelto in base alla tipologia di Controller specificata nel costruttore.

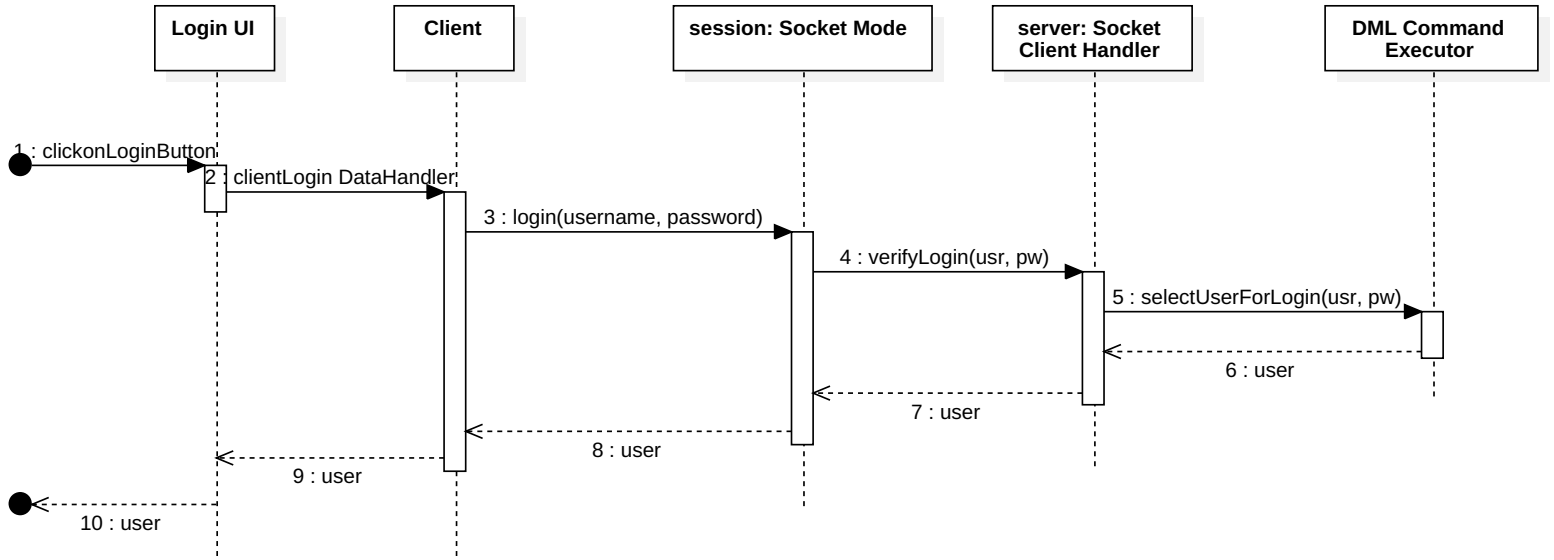
Di seguito si riporta come si sono strutturate le classi dati.



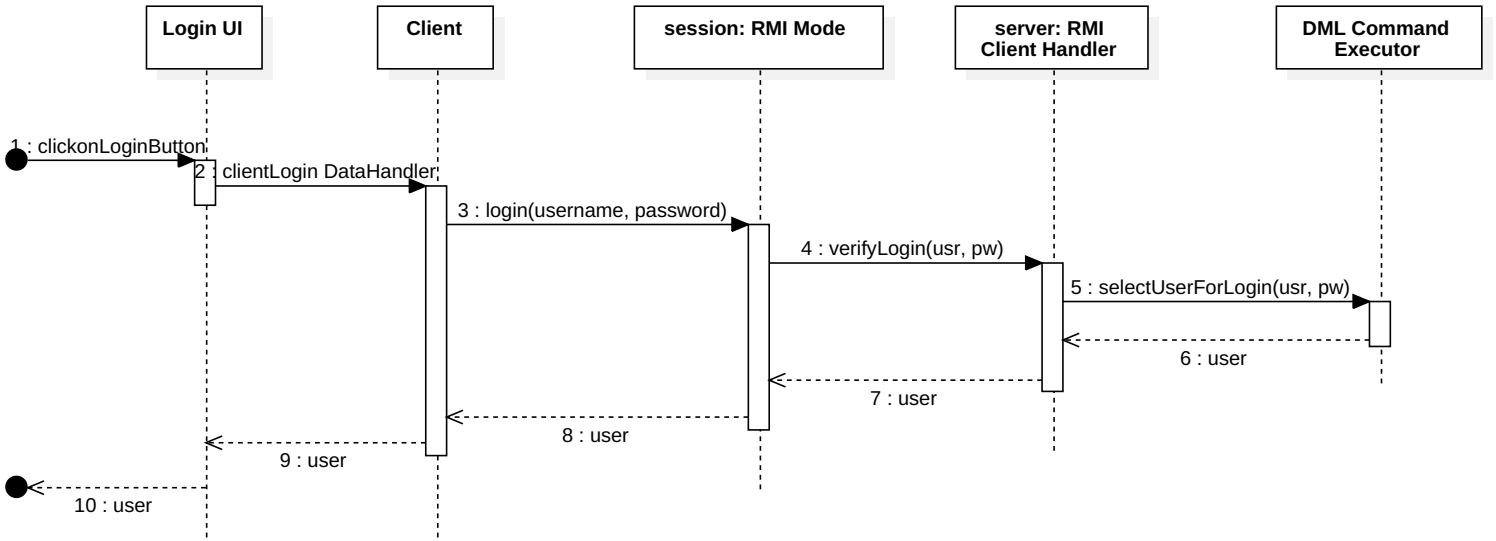
Di seguito si include anche una piccola gerarchia di classi. Si sono omesse le altre classi in quanto il diagramma delle classi risulta superfluo in quanto non sono altro che la mappatura del ER.



SEQUENCE DIAGRAM



SEQUENCE DIAGRAM



Si noti come entrambi i sequence diagram siano praticamente uguali, infatti logicamente in entrambi si ha lo stesso flusso di esecuzione. Si noti come per altri metodi come inserimento ingrediente si esegue lo stesso flusso.

1. Un bottone oppure un'immagine viene cliccata, a questa immagine è associata un'azione corrispondente nel FXML che invoca il metodo associato nel Controller.

   ```
   <Button fx:id="insertIngredient" onAction="#handleInsertIngredientAction" .../>
   ```

2. A questo punto il metodo del controller non fa altro che prendere le informazioni inserite nella GUI, validarle e passarle alla classe Client.

3. La classe Client invoca su un oggetto session un metodo della interfaccia SessionMode, che a seconda del tipo di connessione scelta, può chiamare un metodo remoto del server (RMI):

   ```
   @Override
   public boolean insertIngredientIntoDishIntoDb(String nomeP, String nomeI) {
      try {
           return server.insertIngredientIntoDishIntoDbExecution(nomeP, nomeI);
       } catch (RemoteException e) {
           return false;
       }
   }
   ```

   oppure inviare un comando al server per identificare il metodo richiesto, seguito dai parametri per l'invocazione dello stesso (Socket).

   ```
   @Override
   public boolean insertIngredientIntoDishIntoDb(String nomeP, String nomeI) {
       try {
            socketObjectOut.writeObject("cmd_insertIngredientIntoDish");
            socketObjectOut.flush();
            socketObjectOut.writeObject(nomeP);
            socketObjectOut.flush();
            socketObjectOut.writeObject(nomeI);
            socketObjectOut.flush();
            return socketObjectIn.readBoolean();
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
   ```

```
}
```

4. Il server ottiene una istanza del *singleton* DML Command Executor che come precedentemente chiarito ottiene una connessione al DB, esegue le query e elabora il risultato.

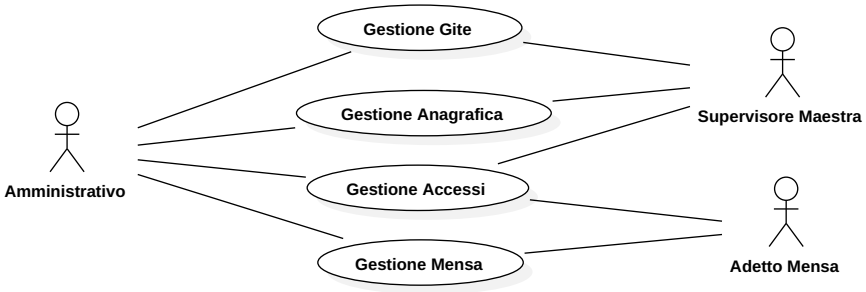
```
public boolean insertIngredientIntoDishIntoDb(String nomeP, String nomeI){
        PreparedStatement stmt = null;
        boolean status = false;
        Connection conn = myPool.getConnection();
        String sql = "INSERT INTO COMPOSIZIONEPIATTO (NomeI, NomeP)" +
         "VALUES(?,?) ON DUPLICATE KEY UPDATE NomeP=?;";

        try {
            stmt = conn.prepareStatement(sql);
            stmt.setString(1,nomeI);
            stmt.setString(2,nomeP);
            stmt.setString(3,nomeP);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        try {
            if (stmt.executeUpdate(sql) == 1)
                status = true;
        } catch (SQLException ex) { //Handle a double key exception
            //ex.printStackTrace();
            System.out.println(ex);
        } finally {
            myPool.releaseConnection(conn);
            return status;
        }
    }
```

5. Il risultato della query/update torna alla GUI lungo gli strati attraversati.

CAPITOLO 3

APPLICATIVO

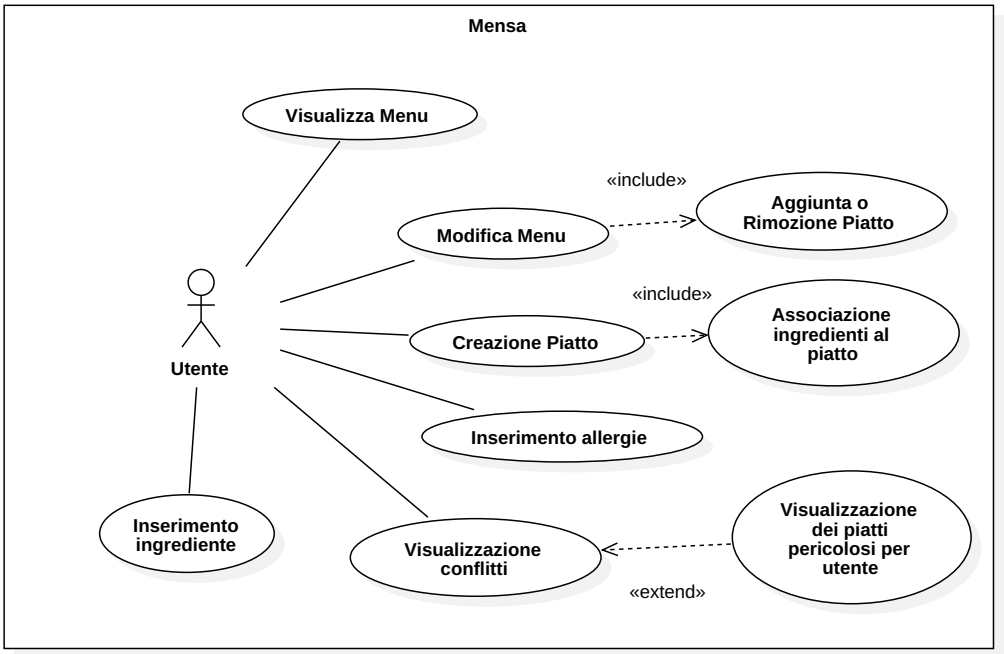
Per l'applicazione si è realizzata una suddivisione degli utenti in modo che ogni categoria di utenti possa realizzare soltanto le azioni che consentite.
Nel nostro caso, ci sono 3 categorie utenti: Amministrativo, Supervisore/Maestra e Adetto mensa. Le funzionalità che possono usufruire sono illustrate dal caso d'uso sottostante.

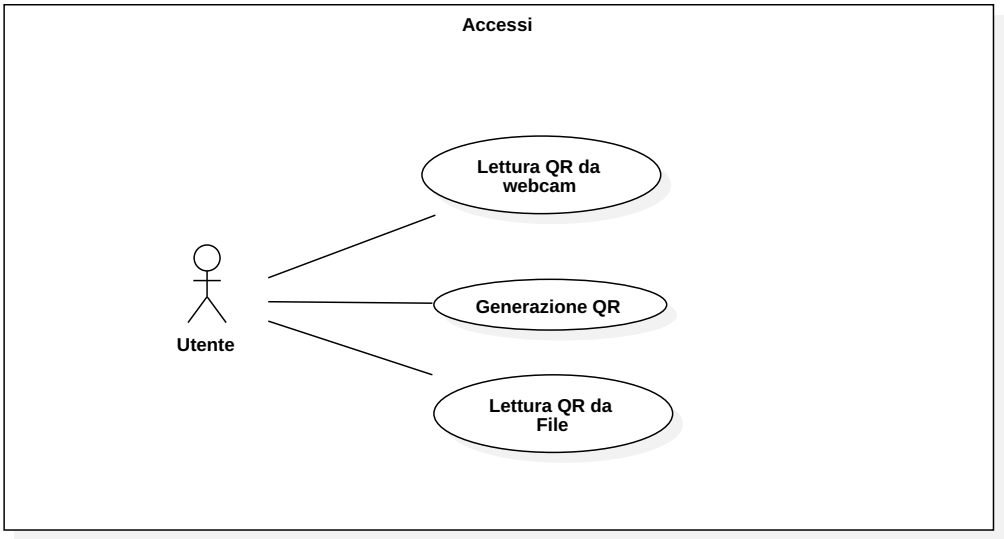


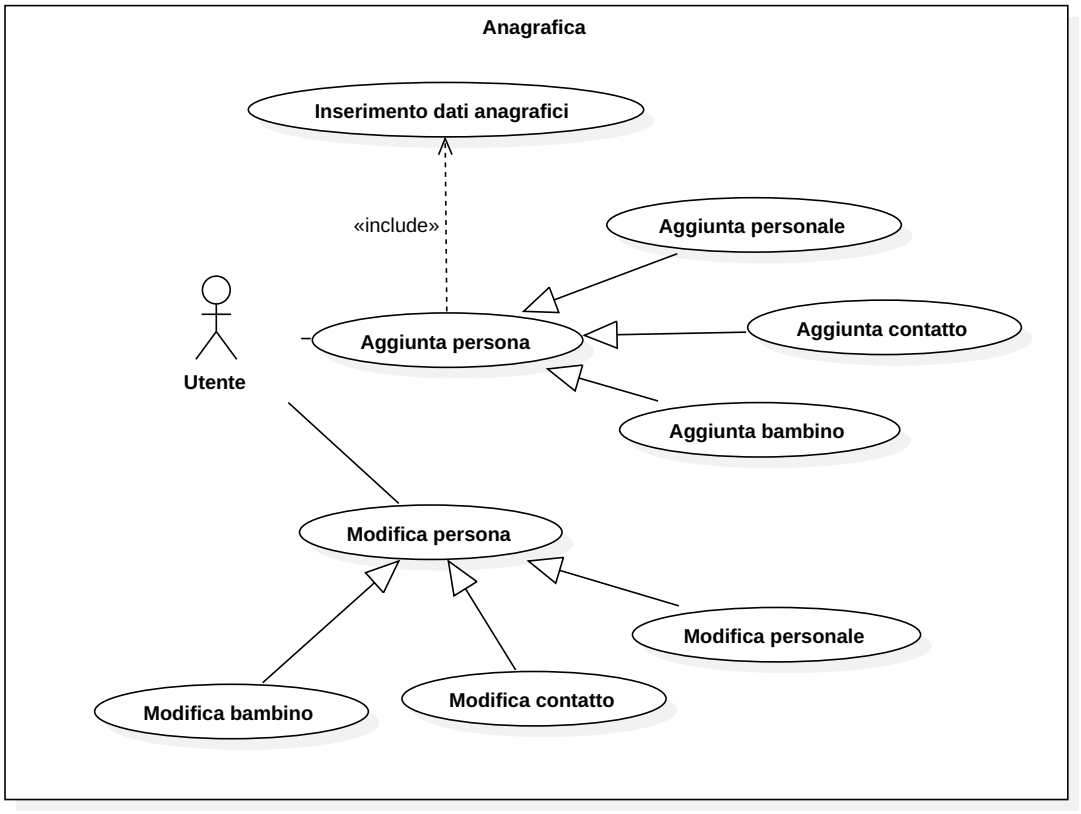
La struttura ideata permette l'aggiunta di una nuova tipologia di utente con poche righe di codice.

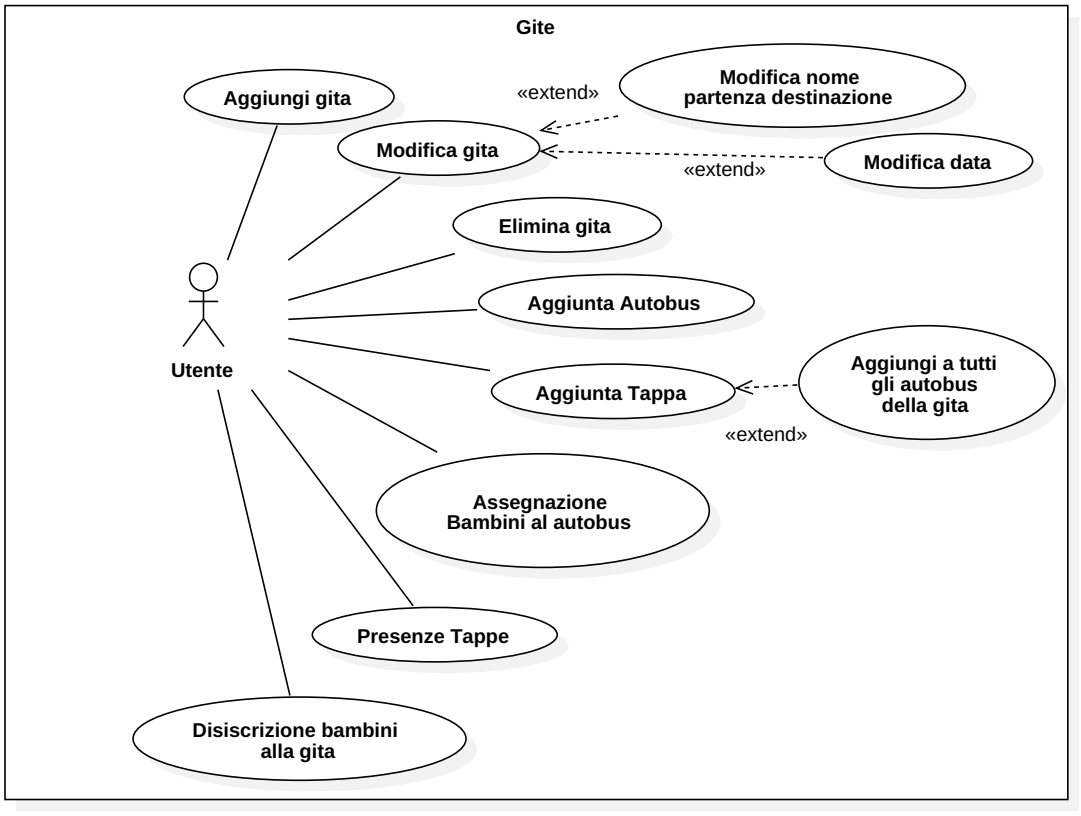
La schermata di login è personalizzata per ogni tipologia di utente è vengono disabilitate le funzionalità non permesse per quella tipologia di utenti.

Le funzionalità dell'applicativo sono evidenziati dai seguenti use case:






Anagrafica
Inserimento dati anagrafici
«include»
Aggiunta personale
Aggiunta persona
Aggiunta contatto
Utente
Aggiunta bambino
Modifica persona
Modifica personale
Modifica bambino
Modifica contatto


Gite
Aggiungi gita
«extend»
Modifica nome
partenza destinazione
Modifica gita
Modifica data
«extend»
Elimina gita
Aggiunta Autobus
Utente
Aggiunta Tappa
Aggiungi a tutti
gli autobus
della gita
«extend»
Assegnazione
Bambini al autobus
Presenze Tappe
Disiscrizione bambini
alla gita

Nell'applicativo si sono usati diversi Design Pattern e si sono applicati delle tecniche di codice viste a lezione. Ad esempio, nei controller della GUI, grazie alla conoscenza dei *Generic*, relazioni di ereditarietà e polimorfismo e una adeguata progettazione delle query al DB si sono potuti creare dei metodi che permettessero l'aggiornamento di una tabella con poche righe di codice facilitando a una persona estranea al progetto l'immediata comprensione del codice.

```
private void personTableRefresh(TableView table,
    ObservableList<StringPropertyPerson> observableList, String classN){

        observableList.clear();
        List<? extends Person> personArrayList = CLIENT.ExtractFromDb(classN);
        if(personArrayList!=null){
            for(Person p : personArrayList){
                observableList.add((StringPropertyPerson) p.toStringProperty());
            }
        }
        table.setItems(observableList);
    }
```

Si ricorda che è la GUI richiede delle ObservableList che contengano oggetti con attributi di tipologia StringProperty per questo motivo si sono create delle classi "normali" e delle classi "string property", le classi "normali" implementano una interfaccia che obbliga l'implementazione del metodo toStringProperty() mentre quelle "string property" implementano un'altra interfaccia che obbliga l'implementazione del metodo duale toNormalClass().

Per l'aggiornamento della tabella del personale basta eseguire la riga di codice.

```
personTableRefresh(staffTable,staffObservableList, Staff.class.getSimpleName());
```

Si noti inoltre che l'esecuzione della seguente riga avrebbe lo stesso effetto.

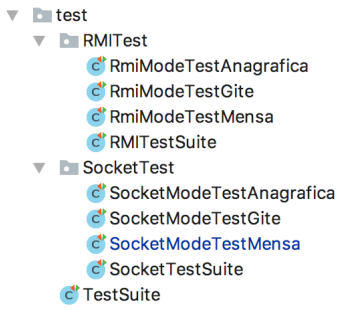
```
personTableRefresh(staffTable,staffObservableList, "Staff");
```

Si è però preferito il primo modo perché in caso di una modifica al nome della classe, il tool di refactor nel IDE può gestire tutti i cambiamenti necessari nel codice.

CAPITOLO 4

TEST

I test sono suddivisi in 2 test suite: test RMI e test Socket. Ogni suite è composta da 3 unit test, una per ogni sezione dell'applicativo.



Ogni test case è stato realizzato in modo che la sua esecuzione non dipenda dai dati già esistenti nel database, quindi l'esito del test non è influenzato in alcun modo dai contenuti o dal momento di esecuzione degli stessi.
Ogni test case ha come intenzione il test di un solo metodo, ma non vale il viceversa, ovvero, possono esserci molteplici metodi di test che verificano il funzionamento di un solo metodo dell'applicativo, e che simulano i possibili casi che possono presentarsi. Ad esempio per il test dell'inserimento di una intolleranza alimentare sono stati progettati i test:

```
insertInvalidChildIntoleranceDue2Ingredient(){...}

insertInvalidChildIntoleranceDue2ChildCode(){...}
```

La realizzazione dell'applicativo ha seguito un ordine ben preciso tenendo sempre in mente il testing, in effetti si è iniziato a implementare l'applicazione a partire dall'anagrafica, essa infatti contiene metodi e funzionalità indispensabili al fine di poter poi procedere con la realizzazione delle funzioni legate a mensa e gite. Una volta che un metodo è stato realizzato e testato, quindi "convalidato" rispetto alla sua specifica, esso è stato utilizzato come metodo di supporto per la costruzione dei successivi test case. Per esempio, in riferimento ai test legati all'iscrizione di un bambino alla gita, il bambino viene prima inserito nel database (metodo facente parte delle funzionalità associate alla gestione dell'anagrafica), viene poi testato il metodo vero e proprio di iscrizione alla gita e infine si svolgono operazioni di "house keeping" per riportare il database allo stato in cui si trovava prima dell'esecuzione del metodo di test.

ESEMPIO

```
@Test
    public void insertDishIntoMenuIntoDb() throws Exception {
        Dish dish = new Dish("PiattoTest", Dish.DishTypeFlag.PRIMO);
        rmiMode.insertDishIntoDb(dish);
        assertTrue(rmiMode.insertDishIntoMenu(getRandMenu(), dish.getNomeP()));
        rmiMode.deleteDishFromMenuFromDb(getExistingMenu(), dish.getNomeP());
        rmiMode.deleteDishFromDb(dish);
    }
```

Si è deciso di seguire particolarmente uno dei principi chiave dell'ingegneria del software, ovvero quello secondo il quale i metodi di test devono essere fra loro indipendenti (non richiamarsi a vicenda e non vincolarsi nell'ordine) e avere un solo metodo assert in ogni test case: ogni test case si concentra infatti sulla verifica di una sola funzionalità.

Trattandosi inoltre di un'applicazione dotata di interfaccia grafica, seguendo lo use case diagram facente parte della documentazione di progetto, si sono eseguiti dei test "manuali" provando in prima persona l'applicativo verificandone così il corretto funzionamento e soprattutto la sua usabilità.