



Università degli Studi di Napoli

“Parthenope”

Dipartimento di Scienze e Tecnologie

Corso di Laurea in Informatica

Corso di “Programmazione 3 e Laboratorio di Programmazione 3”



Gestione forniture ASL

Docenti:

Angelo Ciaramella
Emanuel Di Nardo

Studenti:

Riccardo Andrea Spinosa: 0124002253
Alessandro Massadoro 0124002450

INDICE

1.Requisiti Assegnati	4
2. Introduzione	4
2.1 Analisi dei requisiti	4
2.2 Progettazione	4
2.2.1 Colloquio ASL	4
2.2.2 Database	5
2.2.2.1 Diagramma EER	6
2.2.2.2 Diagramma UML	6
2.2.2.3 Diagramma Casi d'uso	7
2.2.3 Interfaccia	7
3 Design Pattern	8
3.1 Principi Solid	8
3.2 Design Pattern	10
3.2.1 Singleton	10
3.2.2 Factory Method	12
3.2.3 Data Access Object (Dao)	14
3.2.4 Model View Controller (MVC)	20
3.2.5 Strategy	22
3.2.6 Prototype	24
3.2.7 Template Method	25
4 Diagramma delle classi	27
5 Parti rilevanti del codice	27
5.1 Funzione di hashing	27
5.2 Vista	29
5.3 Change Button	30
6 Gestione degli Errori	3
	34
7 MockUp	35
8 Testing	48
9 Java Doc	49

1. Requisiti Assegnati

- usare almeno due pattern per persona tra i design pattern noti;
- attenersi ai principi della programmazione SOLID;
- usare il linguaggio Java;
- inserire sufficienti commenti (anche per Javadoc) e annotazioni ;
- gestione delle eccezioni;
- usare i file o database;

2. Introduzione

2.1 Analisi dei requisiti

Il prototipo realizzato ha lo scopo di gestire la conservazione delle informazioni di tutto il necessario per poter gestire le forniture.

Ci viene richiesto di controllare gli accessi al nostro sistema in quanto è necessario consentire l'accesso solo agli utenti abilitati. Un utente una volta all'interno del client deve poter accedere alle informazioni presenti nel database e consente all'utente di visualizzare, modificare ed inserire nuovi dati, il primo accesso al sistema verrà eseguito con una chiave Admin inserita di default.

Si deve tener conto del privilegio dell'utente in quanto l'inserimento dei nuovi utenti e la visualizzazione di quelli inseriti deve essere permessa solo all'amministratore di sistema.

L'usabilità del sistema è stata testata passo per passo dai futuri utilizzatori del sistema.

2.2 Progettazione

Per la realizzazione del prototipo si è optato per un applicazione desktop sviluppata in Java e GUI sviluppata in JavaFX mediante l'utilizzo di SceneBuilder.

La progettazione è stata suddivisa in più fasi distinte riproducendo uno scenario reale.

2.2.1 Colloquio ASL

Il primo approccio è stato carpire informazioni che potessero risultare utili alla comprensione del problema propostoci così è stato organizzato un colloquio presso l'ASL NAPOLI 1 raccogliendo le prime fondamentali informazioni sul funzionamento e sulla gestione delle forniture.

2.2.2 Database

l'ASL si suddivide in **distretti**, ogni distretto è composto da differenti **strutture**, ogni struttura stipula **contratti** di **fornitura** e riceve N **bollette**. Ogni bolletta è pagata attraverso l'emissione di un **mandato**.

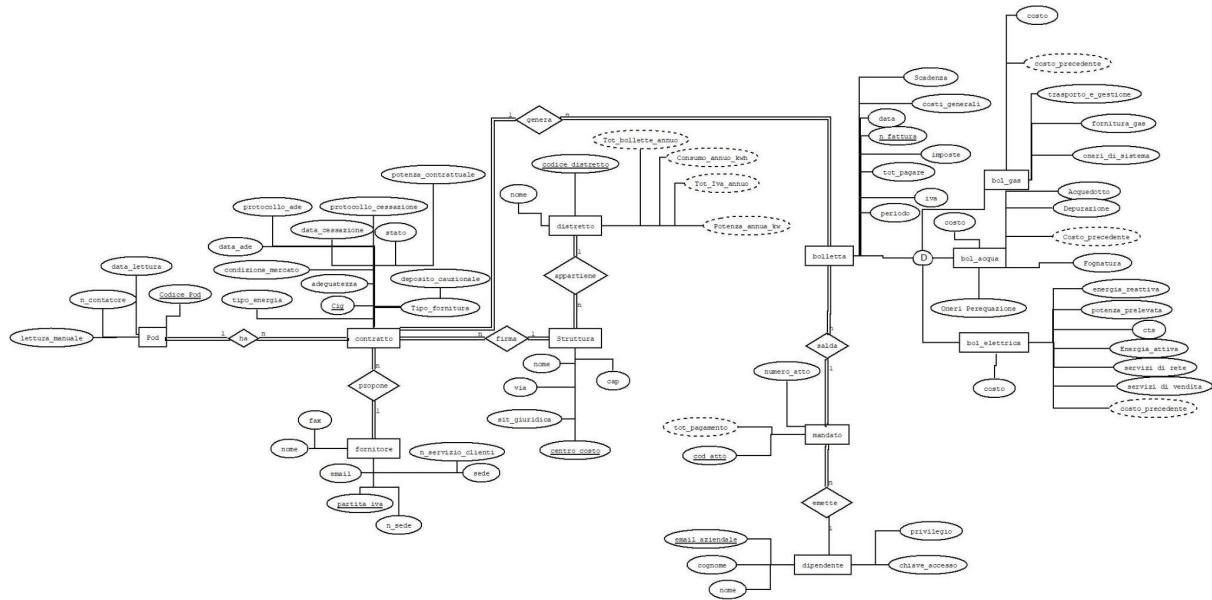
Il mandato viene poi invitato ad un ufficio che gestisce e si occupa della vera e propria parte burocratica. L'ufficio è unico e non è richiesto venga salvato all'interno del database.

È stato scelto l'utilizzo di un database relazionale in quanto i dati trattati sono dati strutturati e ciò permette una gestione ottimale dei dati, ottenendo risparmio di spazio e una maggiore efficienza data dalla potenza del SQL.

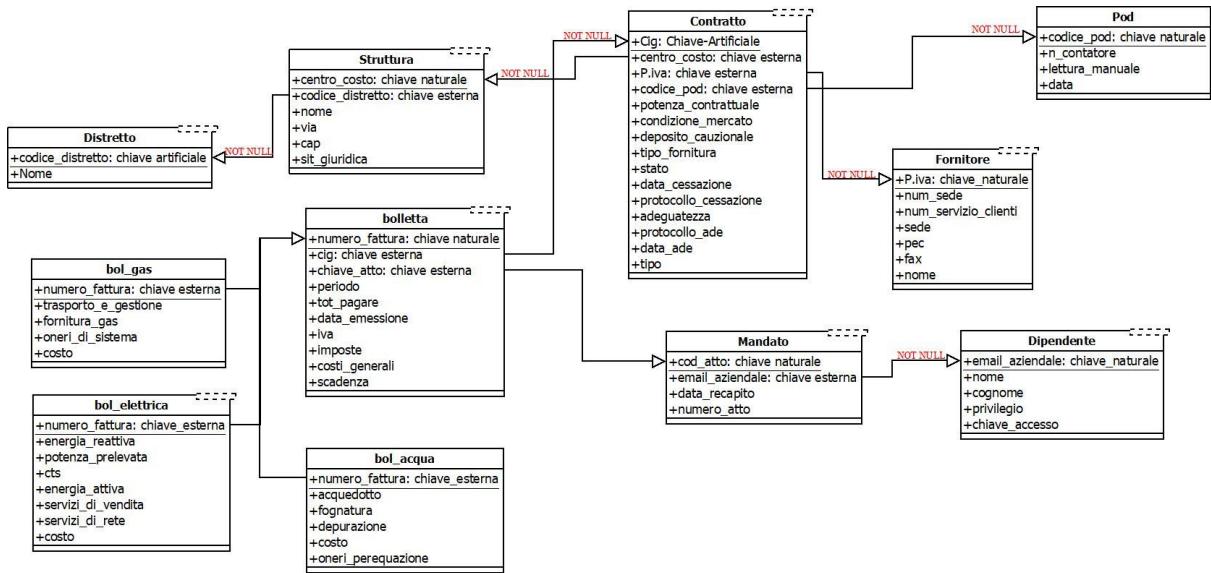
Per lo sviluppo si è quindi deciso di utilizzare Oracle PLSQL

Di seguito riportiamo gli schemi EER, UML, Casi d'uso, ricordiamo che trigger funzioni e procedure sono visibili nel documento relativo al database:

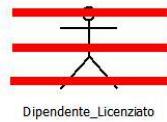
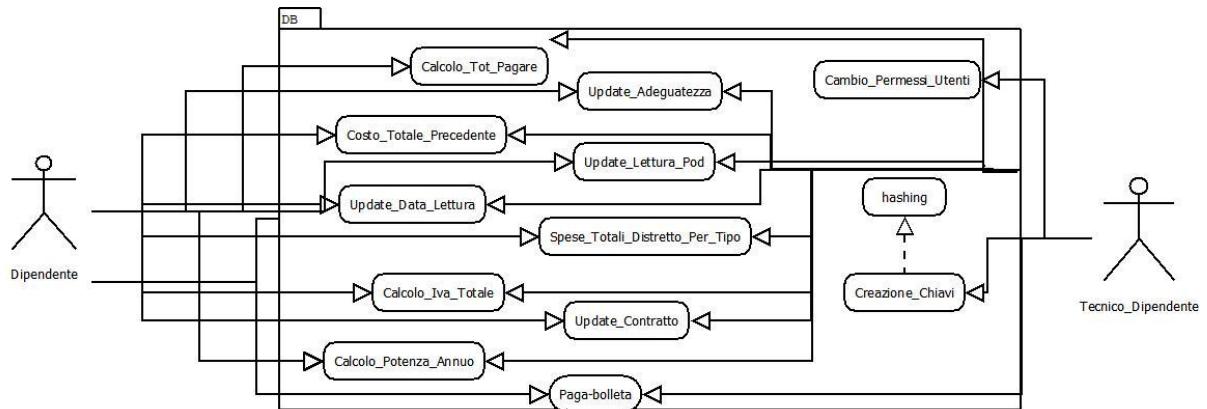
2.2.2.1 Diagramma EER



2.2.2.2 Diagramma UML

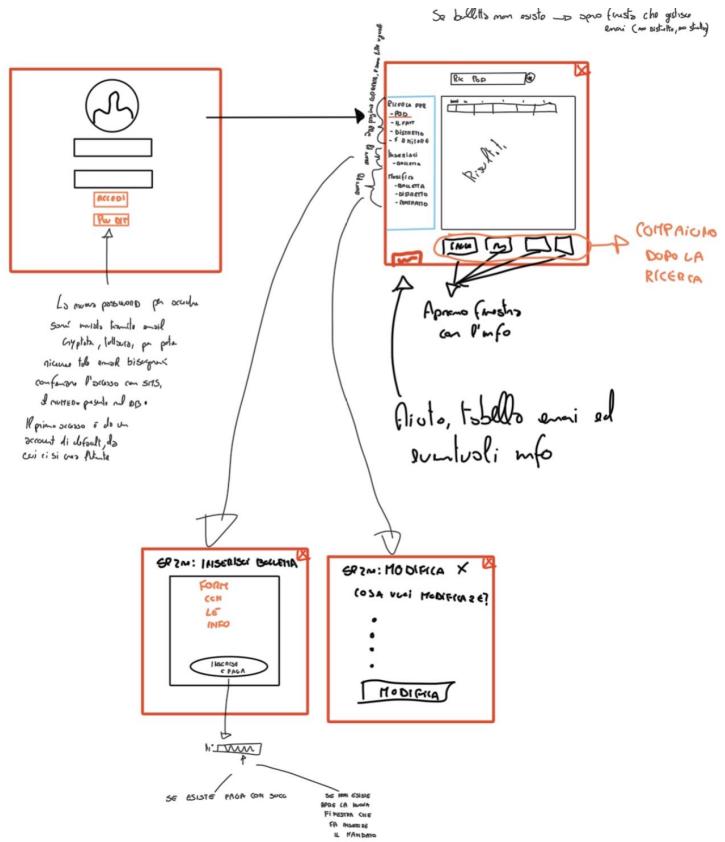


2.2.2.3 Diagramma Casi d'uso



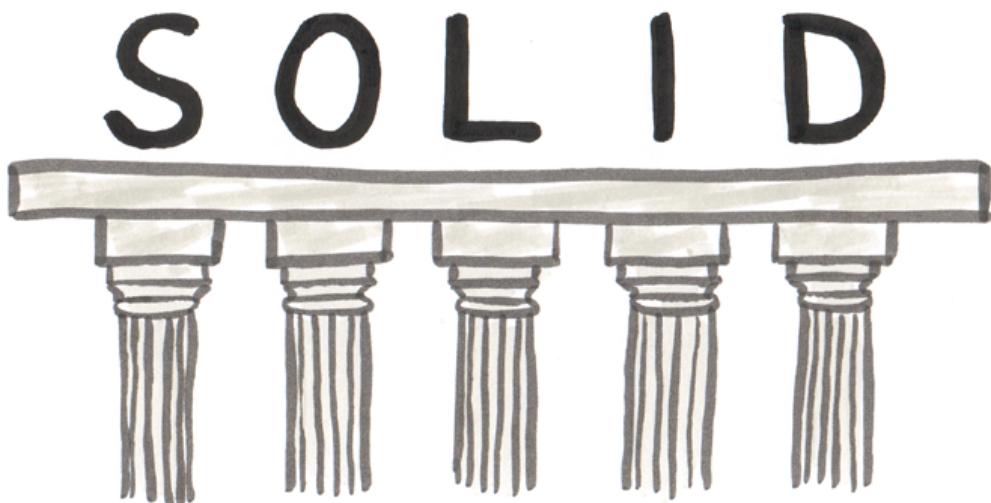
2.2.3 Interfaccia

Dopo molte discussioni tra colleghi siamo giunti ad una conclusione comune, portando ad una realizzazione di una **bozza**. L'obiettivo del prototipo deve essere la realizzazione di un'interfaccia intuitiva e di facile comprensione, nonostante sia un gestionale complesso rivolto ad un personale specializzato. La nostra interfaccia deve comunque nonostante la complessità dell'argomento essere semplice sin dal suo primo utilizzo.



3 Design Pattern

3.1 Principi Solid



I principi SOLID sono un insieme di regole di progettazione software che mirano a creare codice leggibile, manutenibile ed estendibile. I principi SOLID sono stati definiti da Robert C. Martin e sono composti dalle seguenti regole:

Single Responsibility Principle (SRP): ogni classe dovrebbe avere una sola responsabilità e un solo motivo per essere modificata, nel nostro caso non seguiamo questo principio nonostante la maggior parte delle classi ha una singola responsabilità nel caso di Dipendente violiamo il principio dato che dipendente ha una doppia funzionalità sia per loggarci che per inserire nuovi dipendenti. Per poter rispettare tale principio si dovrebbe separare tale classe in due classi distinte, una che gestisce il dipendente che effettua il login definendola superclasse e l'altra che eredita da essa per gestire l'inserimento nel database.

Open/Closed Principle (OCP): le classi dovrebbero essere aperte per estensione ma chiuse per modifica, nel nostro caso rispettiamo questo principio in quanto creiamo varie interfacce per la gestione di varie funzionalità ed inoltre usufruiamo dell'ereditarietà (es: bolletta)

Liskov Substitution Principle (LSP): le sottoclassi dovrebbero essere in grado di sostituire le classi base senza causare problemi al sistema. Nel nostro caso rispettiamo questo principio in quanto le nostre sotto classi, sostituiscono le classi basi senza causare problemi al sistema.

Interface Segregation Principle (ISP): le interfacce dovrebbero essere piccole e specifiche in modo che le classi che le implementano non siano obbligate a implementare metodi che non utilizzeranno. Nel nostro caso in fase di progettazione le interfacce sono piccole e specifiche ed implementano pochi metodi, inoltre in nessuna classe obblighiamo l'implementazione di un metodo che o nel prototipo o nel futuro non verranno utilizzate.

Dependency Inversion Principle (DIP): i moduli di basso livello sono quelli che forniscono funzionalità di base e sono meno astratti rispetto ai moduli di alto livello.

Il principio di progettazione dei software che stabilisce che i moduli di alto livello (ad esempio, le classi che rappresentano le funzionalità principali del sistema) non dovrebbero dipendere dai moduli di basso livello. Ciò consente di modificare o sostituire facilmente le classi basse senza influire sulle classi alte e migliora la flessibilità e la manutenibilità del sistema. Nel nostro caso i moduli di alto livello possono essere utilizzati senza essere vincolati ad una specifica implementazione di basso livello, possiamo effettuare operazioni di basso livello attraverso i moduli DAO senza toccare la logica di business delle classi ad alto livello e viceversa.

3.2 Design Pattern

Si è scelto di utilizzare diversi design pattern in quanto si aveva necessità di risolvere problemi di progettazione comuni in modo efficiente e riutilizzabile.

I design pattern infatti forniscono soluzioni ben testate e collaudate a problemi di progettazione specifici che possono verificarsi in diversi contesti.

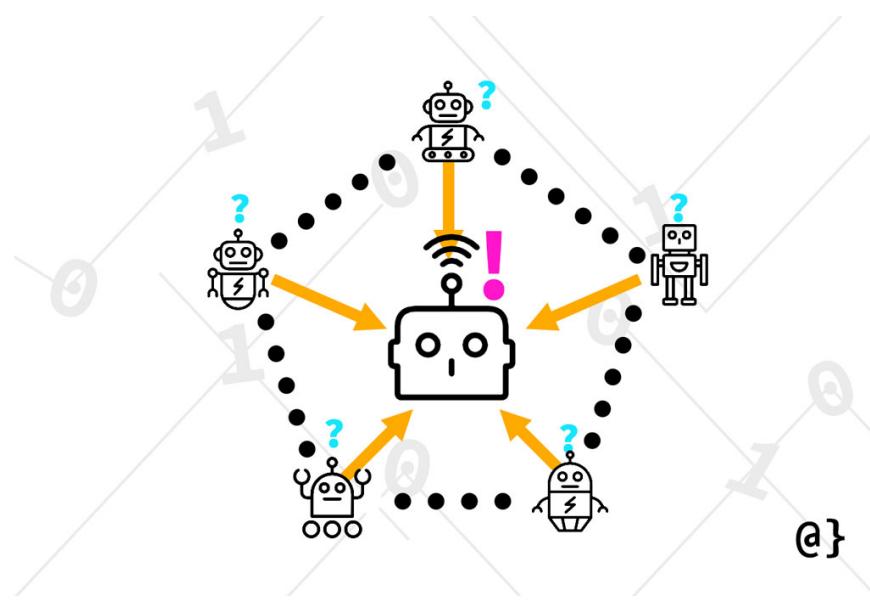
Utilizzando i design pattern, si evita di dover reinventare la ruota ogni volta che si presenta un problema simile e ci si può invece concentrare sull'implementazione della soluzione specifica per il progetto.

Inoltre, l'utilizzo dei design pattern rende il codice più leggibile e comprensibile per gli altri sviluppatori, poiché i pattern sono ben noti e documentati nella comunità dello sviluppo software.

Pattern utilizzati:

- Singleton
- Factory Method
- Data Access Object (Dao)
- Model View Controller (MVC)
- Strategy
- Prototype
- Template Method

3.2.1 Singleton



Il pattern Singleton è un design pattern che garantisce che una classe abbia una sola istanza e fornisce un punto di accesso globale a essa. In particolare, il pattern Singleton è stato utilizzato in due situazioni principali: tenere traccia dell'utente connesso per poter

tracciare le operazioni eseguite e, inoltre, combinato con il pattern Prototype, è stato possibile gestire e trasportare il clone solo quando necessario, istanziandolo una sola volta durante la fase di login

```
public class OggettoCondiviso {
    2 usages
    private static Dipendente sharedObject; //oggetto condiviso tra le classi
    2 usages
    private static Contratto oggettoContratto; //oggetto condiviso per il contratto
    no usages new*
    ● private OggettoCondiviso() {} // costruttore privato

    //metodo per impostare l'oggetto condiviso
    1 usage new*
    public static void setSharedObject(Dipendente obj)
    {
        sharedObject = obj;
    }

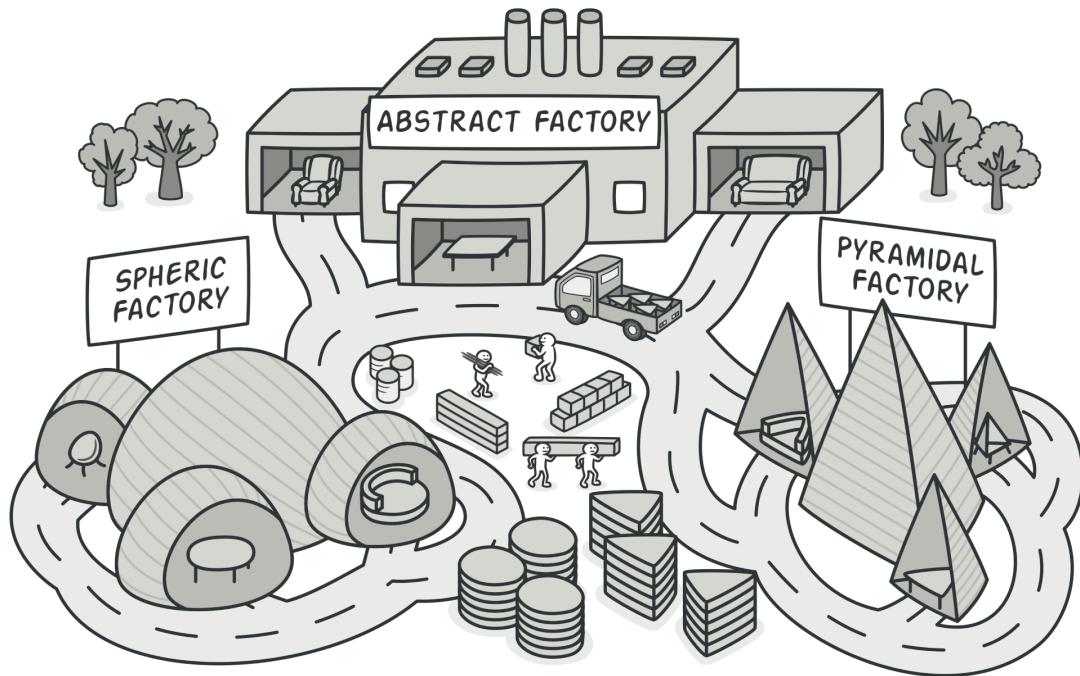
    //metodo per ottenere l'oggetto condiviso
    3 usages new*
    public static Dipendente getSharedObject()
    {
        return sharedObject;
    }

    //metodo per ottenere l'oggetto condiviso del contratto, utilizzato per instanziare il contratto
    //prototype una sola volta
    1 usage new*
    public static Contratto getOggettoContratto()
    {
        return oggettoContratto;
    }

    //metodo per settare l'oggetto condiviso di contratto, utilizzato per instanziare il contratto
    //prototype una sola volta

    1 usage new*
    public static void setOggettoContratto(Contratto obj)
    {
        oggettoContratto=obj;
    }
}
```

3.2.2 Factory Method



Il Factory Method è un pattern utilizzato per la creazione di istanze a runtime. La classe BollettaFactory utilizza il pattern di progettazione Factory Method per creare oggetti di tipo Bolletta (BollettaElettrica, BollettaAcqua, BollettaGas) senza dover specificare esattamente la classe dell'oggetto da creare. Utilizzando i metodi statici creaBolletta(), l'utente che utilizza il sistema può inserire una bolletta elettrica, acqua o gas in modo dinamico, senza dover sapere in anticipo quale tipo di bolletta creare. La logica di creazione viene spostata in questa classe, separandola dal client.

Inoltre, l'interfaccia InterfacciaFactory definisce i metodi comuni che dovranno essere implementati dalle classi BollettaElettrica, BollettaAcqua e BollettaGas. Ciò garantisce che tutte le classi che implementano questa interfaccia offriranno i medesimi metodi, in modo da poter essere utilizzate in modo intercambiabile.

```

package progetto.progetto_asl;
// QUESTA INTERFACCIA RICHIAMA I METODI COMUNI ALL'INTERNO DELLE BOLLETTE
// e la utilizziamo in quanto utilizziamo il Pattern factory per gestire la creazione
// delle bollette
9 usages 3 implementations new *
public interface InterfacciaFactory
{

    //metodo per inserire i dati all'interno della bolletta
    no usages 3 implementations new *
    public void inserisciDati(); // INSERIMENTO DENTRO DATABASE

    //metodo che mostra tutti i dati presenti nella bolletta
    no usages 3 implementations new *
    public void stampaDettagli();

}

```

```

package progetto.progetto_asl;

3 usages new *
public class BollettaFactory {
    /*La classe BollettaFactory utilizza il pattern di progettazione Factory Method per creare oggetti di tipo Bolletta
    (BollettaElettrica, BollettaAcqua, BollettaGas) senza dover specificare esattamente la classe dell'oggetto da creare.
    Utilizzando i metodi statici creaBolletta(), l'utente che utilizza il sistema può inserire una bolletta elettrica,
    acqua o gas in modo dinamico, senza dover sapere in anticipo quale tipo di bolletta creare.
    La logica di creazione viene spostata in questa classe, separandola dal client.*/
    /*
    Il metodo creaBolletta() è un metodo statico che permette di creare un'istanza di un oggetto Bolletta.
    In base ai parametri passati, verrà creata una Bolletta di tipo specifico (Elettrica, Acqua o Gas)
    */
    1 usage new *
    public static Bolletta creaBolletta(String numeroFattura, String periodo, float totPagare, String dataEmissione, float iva, float imposte, String scadenza, float costiGenerali,
                                         String codiceMandato, String cig, String pod, float primo, float secondo, float terzo, float quarto, float quinto, float sesto)
    {
        return new BollettaElettrica(numeroFattura, periodo, totPagare, dataEmissione, iva, imposte, scadenza, costiGenerali, codiceMandato, cig, pod, primo, secondo, terzo, quarto, quinto, sesto);
    }

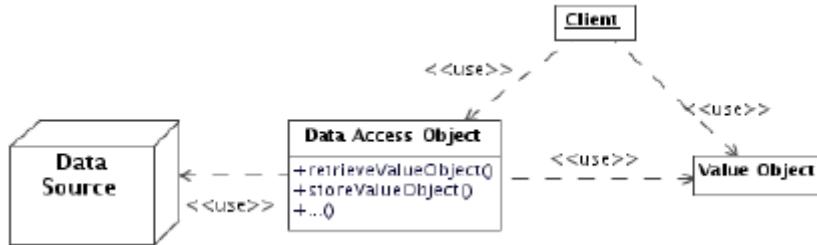
    1 usage new *
    public static Bolletta creaBolletta(String numeroFattura, String periodo, float totPagare, String dataEmissione, float iva, float imposte, String scadenza, float costiGenerali,
                                         String codiceMandato, String cig, String pod, float primo, float secondo, float terzo, float quarto, float quinto)
    {
        return new BollettaAcqua(numeroFattura, periodo, totPagare, dataEmissione, iva, imposte, scadenza, costiGenerali, codiceMandato, cig, pod, primo, secondo, terzo, quarto, quinto);
    }

    1 usage new *
    public static Bolletta creaBolletta(String numeroFattura, String periodo, float totPagare, String dataEmissione, float iva, float imposte, String scadenza, float costiGenerali,
                                         String codiceMandato, String cig, String pod, float primo, float secondo, float terzo, float quarto)
    {
        return new BollettaGas(numeroFattura, periodo, totPagare, dataEmissione, iva, imposte, scadenza, costiGenerali, codiceMandato, cig, pod, primo, secondo, terzo, quarto);
    }

}

```

3.2.3 Data Access Object (Dao)



Il Data Access Object (DAO) è un pattern di progettazione che consente di separare il codice di accesso ai dati dalle altre parti dell'applicazione. Il DAO fornisce un'interfaccia per l'accesso ai dati, indipendentemente dalla fonte dei dati stessa

Nel nostro codice utilizziamo il pattern DAO (Data Access Object) per gestire l'accesso ai dati delle bollette elettriche. Il pattern DAO consente di isolare la logica di accesso ai dati dal resto dell'applicazione, rendendo più facile modificare o sostituire la fonte dei dati senza modificare il resto dell'applicazione.

In questo caso, è stato creato un'interfaccia chiamata InterfacciaDao, che definisce i metodi per ottenere tutte le istanze di una determinata classe, filtrare un'istanza per chiave, salvare un'istanza in un database e cancellare un'istanza dal database.

Successivamente, è stata creata una classe astratta ConnessioneDao che si occupa di stabilire e gestire la connessione al database. La classe BollettaElettricaDao estende ConnessioneDao e implementa l'interfaccia InterfacciaDao per la classe VistaLuce.

La classe BollettaElettricaDao contiene il metodo getAllObj() che recupera tutti i dati dalla tabella TABELLONE_STRUTTURA_LUCE e li inserisce in una lista osservabile di oggetti VistaLuce.

L'interfaccia InterfacciaDao fornisce i metodi per filtrare una determinata istanza della classe per chiave. In questo caso, il metodo getTByKey(T x) consente di filtrare un oggetto di tipo T in base ad una chiave specifica, che può essere un valore univoco come un ID o una combinazione di valori.

Per quanto riguarda l'implementazione di questo metodo, dipende dalla classe specifica che lo utilizza. Ad esempio, nella classe BollettaElettricaDao, potrebbe essere utilizzato per filtrare una bolletta elettrica in base al suo numero di fattura. In questo caso, il metodo

potrebbe richiedere un parametro String numeroFattura e utilizzare questo valore per creare una query SQL per recuperare la bolletta elettrica con quel numero di fattura dal database.

Inoltre, la classe ConnessioneDao è una classe astratta che estende java.sql.Connection e fornisce una connessione al database. Il costruttore della classe crea una connessione al database utilizzando le informazioni di connessione specificate e fornisce un metodo getConn() per recuperare l'oggetto Connection creato.

Per quanto riguarda DipendenteDao

La classe eredita i metodi per la gestione delle connessioni al database dalla classe ConnessioneDao e implementa l'interfaccia InterfacciaDao per poter utilizzare i metodi CRUD (create, read, update, delete) per la classe Dipendente. Il metodo getAllObj() dovrebbe restituire una lista osservabile di tutti i dipendenti presenti nel database, tuttavia nel codice fornito il metodo non è stato implementato e restituisce null. Il metodo getTByKey(Dipendente x) dovrebbe essere utilizzato per filtrare un dipendente in base alla sua email aziendale e alla sua password, utilizzando una query preparata per prevenire attacchi di SQL injection. Il metodo esegue la query, recupera i dati del dipendente e li stampa a console. Il metodo save(Dipendente x) dovrebbe essere utilizzato per salvare un nuovo dipendente nel database. Nel codice fornito, il metodo esegue una chiamata ad una funzione di hashing per criptare la password del dipendente e poi esegue una query per inserire i dati del dipendente nel database. I metodi delete(Dipendente x) non sono stati implementati nel codice fornito.

```

package progetto.progetto_asl;

import ...
// interfaccia da implementare per tutte le classi che richiedono il DAO

2 usages 2 implementations new*
public interface InterfacciaDao<T> {

    // operazione per ottenere tutte le instance di una determinata classe
    2 usages 2 implementations new*
    ObservableList <T> getAllObj();
    // operazione per filtrare l'stanza per chiave
    1 usage 2 implementations new*
    T getTByKey(T x);
    // operazione che ci permette di salvare l'oggetto all'interno del database
    1 usage 2 implementations new*
    void save(T x);

    // operazione che consente la cancellazione del determinato oggetto, se presente nel database
    no usages 2 implementations new*
    void delete(T x);

}

```

```

2 usages 2 inheritors new*
public abstract class ConnessioneDao {

    3 usages
    private Connection conn;

    {
        try {
            // Carica il driver JDBC per Oracle
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Crea la connessione al database
            conn = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe", "c##tecnico_dipendente", "tecnico");

        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    // metodo getter per ottenere l'oggetto Connection
    8 usages new*
    public Connection getConn() { return conn; }

    // metodo per chiudere la connessione
    3 usages new*
    public void chiudiConnessione() throws SQLException {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```
public class BollettaElettricaDao extends ConnessioneDao implements InterfacciaDao<VistaLuce> {
    2 usages new*
    @Override
    public ObservableList<VistaLuce> getAllObj() {
        try{
            // Creo la query per recuperare tutti i dati dalla tabella TABELLONE_STRUTTURA_LUCE
            String query = "SELECT * FROM TABELLONE_STRUTTURA_LUCE";
            PreparedStatement pstmt = getConn().prepareStatement(query);
            // Creo una lista osservabile per i dati delle bollette elettriche
            ObservableList<VistaLuce> data = FXCollections.observableArrayList();
            ResultSet rs = pstmt.executeQuery();
            // Ciclo per recuperare tutti i dati dalla tabella
            while(rs.next()){
                // Recupero i dati dalla tabella
                System.out.println("sono entrato nel ciclo while");
                String centroCosto = rs.getString( columnLabel: "CENTRO_COSTO");
                String dataEmissione = rs.getString( columnLabel: "DATA_EMISSIONE");
                String numeroFattura = rs.getString( columnLabel: "NUMERO_FATTURA");
                String costo = rs.getString( columnLabel: "COSTO");
                String energiaAttiva= rs.getString( columnLabel: "ENERGIA_ATTIVA");
                String energiaReattiva = rs.getString( columnLabel: "ENERGIA_REATTIVA");
                String potenzaPrelevata = rs.getString( columnLabel: "POTENZA_PRELEVATA");
                String serviziDiVendita = rs.getString( columnLabel: "SERVIZI_DI_VENDITA");
                String serviziDiRete= rs.getString( columnLabel: "SERVIZI_RETE");
                String imposte = rs.getString( columnLabel: "IMPOSTE");
                String iva = rs.getString( columnLabel: "IVA");
                String codicePod= rs.getString( columnLabel: "CODICE_POD");
                VistaLuce vistaluce = new VistaLuce(centroCosto,dataEmissione,numeroFattura,costo,energiaAttiva,energiaReattiva,
                                                potenzaPrelevata,serviziDiVendita,serviziDiRete,imposte,iva,codicePod);
                data.add(vistaluce);
            }
            System.out.println("ho terminato");
            return data;
        }catch(SQLException e){
            System.out.println("ECCEZIONE SQL");
            return null;
        }
    }
}
```

```
public class DipendenteDao extends ConnessioneDao implements InterfacciaDao<Dipendente>{

    //il metodo ci permette di ritornare a schermo tutti i dipendenti presenti nel database
    @Override
    public ObservableList getAllObj() {
        return null;
    }

    @Override
    public Dipendente getTByKey(Dipendente x) {
        try{
            System.out.println("SONO DENTRO DIPENDENTE DAO");
            // crea un oggetto per poter eseguire la query
            //PER MOTIVI DI SICUREZZA CONVIENE UTILIZZARE STATEMENT PER PREVENIRE ATTACCHI SQL INJECTION
            PreparedStatement stmt = getConn().prepareStatement( sql: "SELECT * FROM DIPENDENTE WHERE EMAILAZIENDALE = ? AND CHIAVEACCESSO = ? ");
            System.out.println(getConn().isClosed());
            stmt.setString( parameterIndex: 1, x.getEmailAziendale());
            stmt.setString( parameterIndex: 2, x.getPassword());
            // Esegue la query
            ResultSet rs = stmt.executeQuery();
            System.out.println("HO ESEGUITO LA QUERY");

            // Lavora con i dati restituiti dalla query
            while (rs.next()) {
                System.out.println("sono dentro il while");
                String Email = rs.getString( columnLabel: "EMAILAZIENDALE");
                String nome = rs.getString( columnLabel: "NOME");
                String cognome = rs.getString( columnLabel: "COGNOME");
                x.setPrivilegio(rs.getInt( columnLabel: "PRIVILEGIO"));
                int privilegio=x.getPrivilegio();
                System.out.println("Email: " + Email + " Nome: " + nome + " Cognome: " + cognome + " Privilegio: " + privilegio );
            }
            System.out.println("HO FINITO");
            chiudiConnessione();
        } catch (SQLException e) {
```

```

        } catch (SQLException e) {
            throw new RuntimeException(e);
        }

        return null;
    }

    //il metodo esegue la ricerca di un dipendente tramite key e ritorna l'eventuale dipendente trovato
    //metodo di salvataggio dei dati all'interno del database, il dipendente inserito
    //viene salvato all'interno del database
    usage new*
    @Override
    public void save(Dipendente x) {
        try{
            //prima di inserire un dipendente creiamo la password, questo avviene richiamando la funzione di hashing
            //di fatto la password quando viene creato un nuovo dipendente per essere sicura
            //non verrà inserita ma creata da una funzione di hashing dal database
            System.out.println("SONO DENTRO DIPENDENTE DAO");
            String call= "{call ?:=FUNZIONE_DI_HASHING(?, ?, ?)}";
            CallableStatement pstmt = getConn().prepareCall(call);
            pstmt.registerOutParameter( parameterIndex: 1, Types.VARCHAR);
            pstmt.setInt( parameterIndex: 2, x: 3);
            pstmt.setInt( parameterIndex: 3, x: 3);
            pstmt.setInt( parameterIndex: 4, x: 4);
            pstmt.setInt( parameterIndex: 5, x: 4);
            pstmt.executeUpdate();
            System.out.println("HO SUPERATO LA CHIAMATA DI FUNZIONE");
            System.out.println("SONO ANCORA IN GARDA");
            String pw = pstmt.getString( parameterIndex: 1);
            System.out.println(pw);
            //query
            String query = "INSERT INTO DIPENDENTE (EMAIL_AZIENDALE,NOME,COGNOME,PRIVILEGIO,CHIAVE_ACCESSO) VALUES (?,?,?,?,?)";
            //apriamo la connessione e crea un oggetto per poter eseguire la query
            PreparedStatement stmt = getConn().prepareStatement(query);
            //chiudiamo la funzione dopo aver richiamato la procedura contenente la funzione che genera una password forte di 16 cifre sicura
            System.out.println(getConn().isClosed());
            System.out.println("HO ESEGUITO LA QUERY");
            stmt.setString( parameterIndex: 1,x.getEmailAziendale());
        }
    }
}

```

```

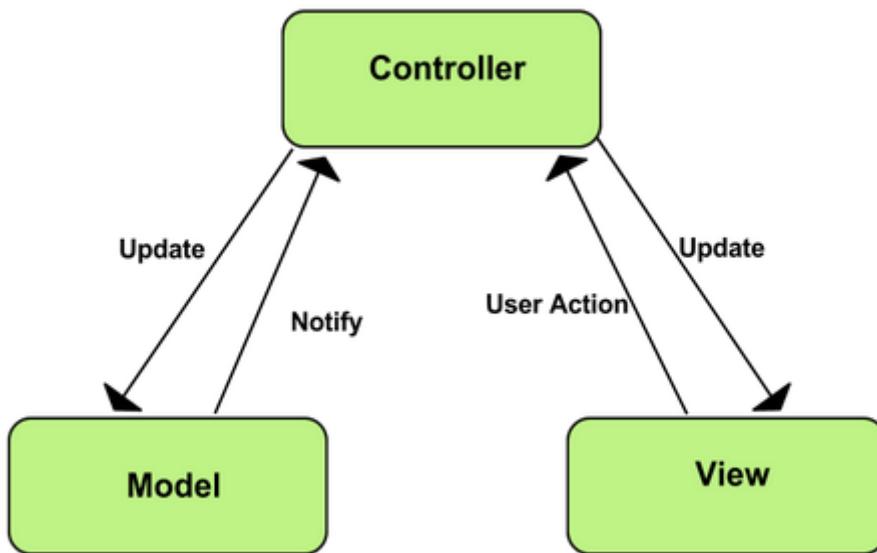
        stmt.setString( parameterIndex: 2,x.getNome());
        stmt.setString( parameterIndex: 3,x.getCognome());
        stmt.setString( parameterIndex: 4, Integer.toString(x.getPrivilegio()));
        stmt.setString( parameterIndex: 5,pw);
        //Esegue la query
        int rowsAffected = stmt.executeUpdate();
        System.out.println("HO FINITO");
        chiudiConnessione();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }

}

no usages new*
@Override
// L'ELIMINAZIONE è SOLO LOGICA SETTIAMO IL PRIVILEGIO A 0 IN MODO CHE NON POSSA NUOVA ENTRARE
//questo è il metodo di cancellazione di un dipendente
//dopo che il privilegio viene settato a 0 non si hanno più i permessi per entrare
public void delete(Dipendente x) {
    try{
        String query = "UPDATE DIPENDENTE SET PRIVILEGIO = 0 WHERE EMAIL = ?";
        System.out.println("SONO DENTRO DIPENDENTE DAO");
        // crea un oggetto per poter eseguire la query
        PreparedStatement stmt = getConn().prepareStatement(query);
        System.out.println(getConn().isClosed());
        System.out.println("HO ESEGUITO LA QUERY");
        stmt.setString( parameterIndex: 1,x.getEmailAziendale());
        //Esegue la query
        int rowsAffected = stmt.executeUpdate();
        System.out.println("HO FINITO");
        chiudiConnessione();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

3.2.4 Model View Controller (MVC)



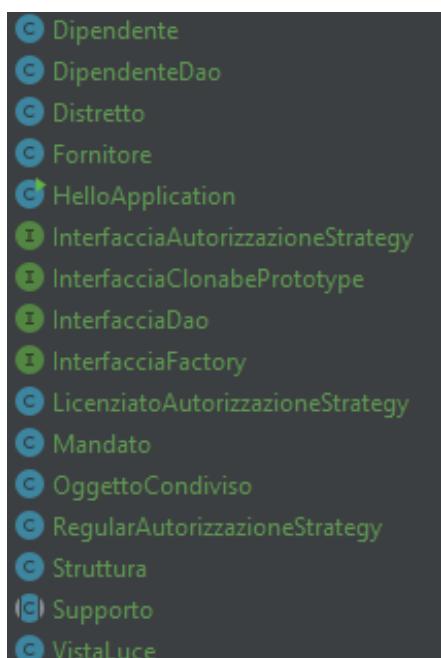
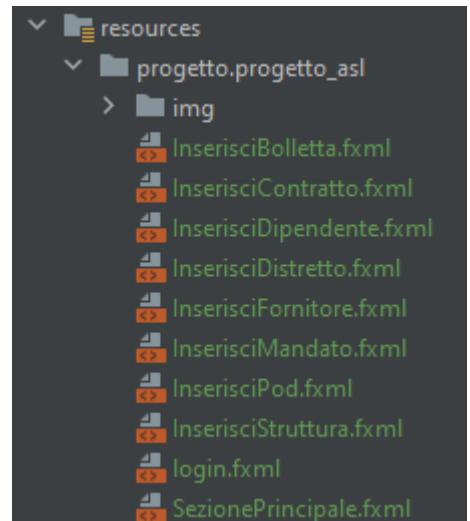
MVC (Model-View-Controller) è un pattern di progettazione software che consente di separare la logica dell'applicazione in tre componenti distinti: il Modello, la Vista e il Controller.

È stato utilizzato questo pattern per due motivi principali: mantenere il codice pulito e separare nettamente le diverse sezioni logiche del lavoro. Il pattern richiede di separare la logica dei moduli in un'applicazione Java e noi abbiamo suddiviso i documenti e le classi in tre sezioni principali, come indicato dal pattern.

La sezione Controller contiene l'interazione e l'elaborazione della sezione View e di quella presente nel Modello. Ciò permette ai controller di essere isolati dietro la logica di funzionamento dei metodi delle classi Modello che creano il nostro ambiente e permette loro di restare in ascolto e reagire in maniera dinamica agli input ricevuti dalla sezione View. Questa sezione è composta dalle classi "Controller".

La sezione View contiene ciò che l'utente vede a schermo, quindi la GUI con cui esso si rapporta. Nel nostro caso, è definita dai file FXML presenti nella sezione Resurse. Questi file sono privi di qualsiasi tipo di logica di implementazione e contengono solo ed esclusivamente la GUI mostrata all'utente.

La sezione Modello contiene tutte le classi utili per il nostro specifico modello. Troviamo quindi le classi che utilizziamo per le funzionalità con il database e quelle utili al nostro sistema "Gestione Forniture ASL".



3.2.5 Strategy

Il pattern Strategy è un pattern di progettazione comportamentale che consente di definire una famiglia di algoritmi, incapsularli e fare in modo che questi algoritmi possano essere intercambiati tra loro. In questo modo, si può cambiare l'algoritmo utilizzato senza dover modificare il codice delle classi che lo utilizzano.

Nel nostro caso la classe AutorizzazioneContext è una classe che serve ad utilizzare la strategia di autorizzazione. La sua funzione principale è quella di incapsulare la logica di autorizzazione e di fornire un punto unico per controllare i privilegi degli utenti. In generale, la classe mantiene un riferimento alla strategia di autorizzazione corrente che può essere impostata in base alle esigenze del sistema. Le tre classi LicenziatoAutorizzazioneStrategy, RegularAutorizzazioneStrategy e AdminAutorizzazioneStrategy implementano l'interfaccia InterfacciaAutorizzazioneStrategy e forniscono l'implementazione del metodo "checkPermission" per verificare se un utente ha i privilegi per eseguire un'azione specifica in base al suo stato (licenziato, utente normale o amministratore).

L'interfaccia InterfacciaAutorizzazioneStrategy fornisce un metodo checkPermission che consente di effettuare il login di un utente regular o admin e di far visualizzare determinate funzionalità a seconda dei loro privilegi.

```
package progetto.progetto_asl;

//interfaccia dovuta al pattern Strategy
6 pages 3 implementations new*
public interface InterfacciaAutorizzazioneStrategy {
    //IL METODO CI CONSENTE DI EFFETTUARE IL LOGIN DI UN UTENTE REGULAR O
    // ADMIN E DI FAR VISUALIZZARE DETERMINATE FUNZIONALITA' A SECONDA DEI
    // LORO PRIVILEGI
    1 usage 3 implementations new*
    boolean checkPermission(Dipendente user, String permission);
}
```

```

package progetto.progetto_asl;

// Classe che implementa l'interfaccia InterfacciaAutorizzazioneStrategy
// per gestire le autorizzazioni degli amministratori
4 usages new*
class AdminAutorizzazioneStrategy implements InterfacciaAutorizzazioneStrategy {

    // Implementazione del metodo checkPermission dell'interfaccia
    // che verifica se l'utente passato come parametro è un amministratore e ha i privilegi per eseguire l'azione specifica
    1 usage new*
    @Override
    public boolean checkPermission(Dipendente user, String permission) {
        // controlla se l'input "permission" è uguale a "2"
        if(permission.equals("2")){
            System.out.println("ho stampato admin");
            return true;
        }
        // Restituisce false se non soddisfa la condizione
        return false;
    }
}

```

```

package progetto.progetto_asl;

// questa classe non implementa nient'altro che la strategy di un utente che non potrà più accedere al database
// di fatti SE L'UTENTE E' STATO LICENZIATO OPPURE E' ANDATO IN PENSIONE NON AVRA' PIU' GLI ACCESSI, ma teniamo
// comunque traccia della sua esistenza ed operazioni passate nel database

1 usage new*
class LicenziatoAutorizzazioneStrategy implements InterfacciaAutorizzazioneStrategy{
    1 usage new*
    @Override
    //riceve il dipendente e controlla la stringa di operazione, in questo caso permesso
    //se il permesso è 0 allora l'utente è licenziato e non può accedere
    public boolean checkPermission(Dipendente user, String permission) {
        if(permission.equals("0"))
            return true;
        return false;
    }
}

```

```

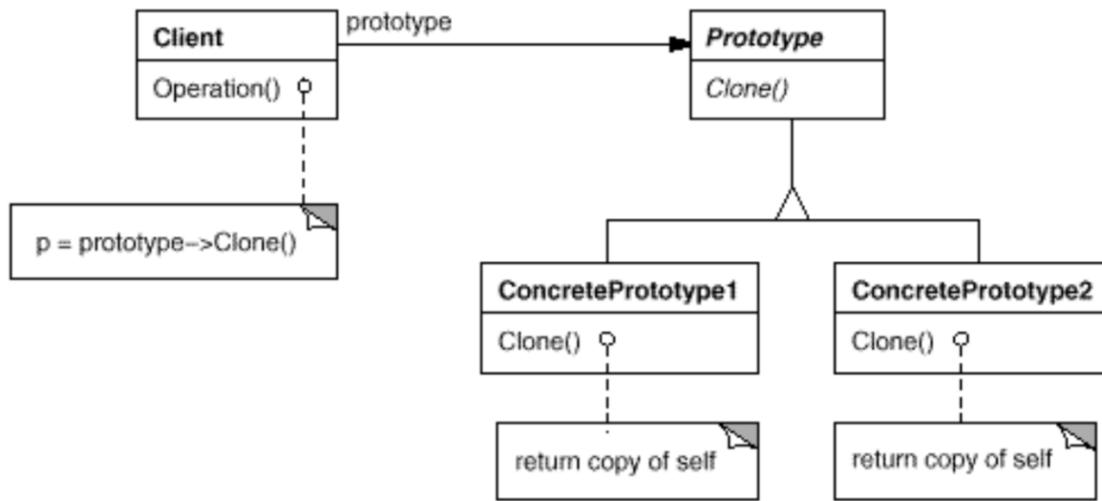
package progetto.progetto_asl;

//la classe implementa la strategy di autorizzazione per un dipendente con accesso "regolari"
//cioè può connettersi al database ma non ha accesso alle funzioni admin

n usages new*
class RegularAutorizzazioneStrategy implements InterfacciaAutorizzazioneStrategy{
    //CONTROLLO PRIVILEGIO
    1 usage new*
    @Override
    public boolean checkPermission(Dipendente user, String permission) {
        if(permission.equals("1"))
            return true;
        return false;
    }
}

```

3.2.6 Prototype



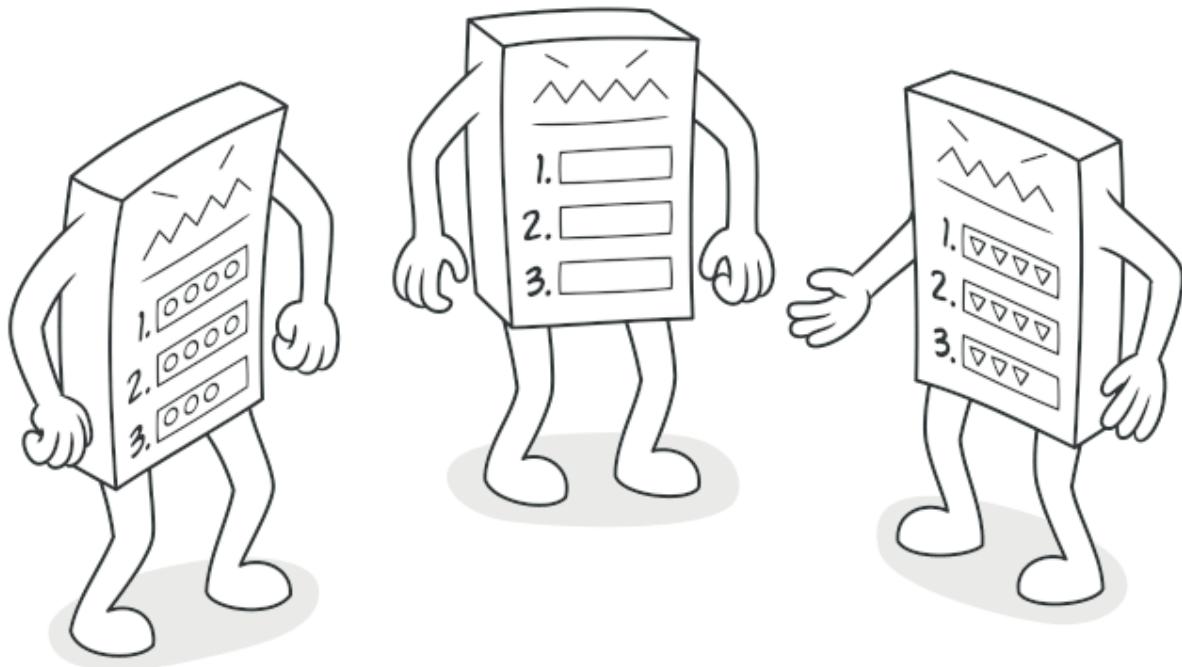
Il pattern Prototype è un design pattern creazionale che consente di creare nuove istanze di un oggetto a partire da una copia di un oggetto esistente, invece di creare un nuovo oggetto da zero.

In generale, l'operazione di scrittura è più lenta rispetto all'operazione di confronto (ad esempio, confronto tra due valori in memoria), a seconda delle dimensioni dei dati e del tipo di storage utilizzato. Scrivere dati su un disco nel nostro caso scrivere dati all'interno della classe contratto può essere più lento rispetto a confrontare i dati in memoria poiché richiede tempi di accesso e di creazione dell'oggetto e d'invocazione del costruttore aumentando così i tempi in quanto i parametri passati sono tanti.

Se invece di creare un nuovo oggetto Contratto utilizziamo un oggetto clonato e sovrascriviamo solo i parametri che cambiano, risparmiamo tempo e miglioriamo l'efficienza del sistema. Il pattern Prototype ci permette di creare facilmente copie di un oggetto esistente, in modo da non dover utilizzare nuovamente il costruttore ogni volta. In questo modo, se una parte dei parametri restano gli stessi tra i vari oggetti Contratto, utilizzare il pattern Prototype potrebbe risultare in una scelta più efficiente.

Abbiamo quindi deciso di implementare il pattern prototype per migliorare l'efficienza del sistema.

3.2.7 Template Method



Il pattern Template Method è un pattern di progettazione che consente di definire in una classe base un algoritmo o un comportamento generale, lasciando alle sottoclassi nel nostro caso i controller possibilità di definire alcuni passi specifici di quell'algoritmo o comportamento, per il cambio di scena. In questo modo, le sottoclassi possono modificare il comportamento del metodo senza doverlo riscrivere completamente. Il risultato è una gerarchia di classi in cui la classe base fornisce un'implementazione generale che può essere personalizzata da classi specifiche.

In questo caso, si è scelto di utilizzare il pattern "Template Method", dove una classe di supporto fornisce una struttura generale per l'esecuzione di una serie di operazioni, lasciando alle sottoclassi la possibilità di personalizzare alcuni passaggi specifici. La classe Supporto fornisce una serie di metodi e proprietà che possono essere utilizzate dalle sottoclassi (i controller) per gestire le funzionalità comuni, come il cambio di scena, senza dover ripetere lo stesso codice in ogni classe.

```

//questo metodo controlla tutti i campi obbligatori per ogni inserimento, nel caso
//uno dei campi obbligatori non venga inserito la funzione ritorna true
//altrimenti ritorna false
//sono presenti diversi overload della funzione in quanto sono necessari
//per i diversi controller
1 usage new*
public boolean obbligatorio(String cig, String centroCosto, String potenzaContrattuale, String stato, String adeguatezza, String partitaIva,
String codicePod, String condizioneMercato, String tipoFornitura, String tipo)
{
    //mettiamo tutti i campi passati come attributi all'interno dell'array così da poterli scorrere
    System.out.println("siamo dentro la funzione obbligatorio");
    String []array={cig,
                    centroCosto,
                    potenzaContrattuale,
                    stato,
                    adeguatezza,
                    partitaIva,
                    codicePod,
                    condizioneMercato,
                    tipoFornitura,
                    tipo,
                    };

    for(String singolo:array){
        //i campi sono ciò che l'utente ha inserito in un TextField, quindi se non è stato inserito
        //risulterà essere vuoto
        if(singolo.isEmpty()){

            return true;
        }
    }
    System.out.println("Obbligatorio è terminato");
    return false;
}

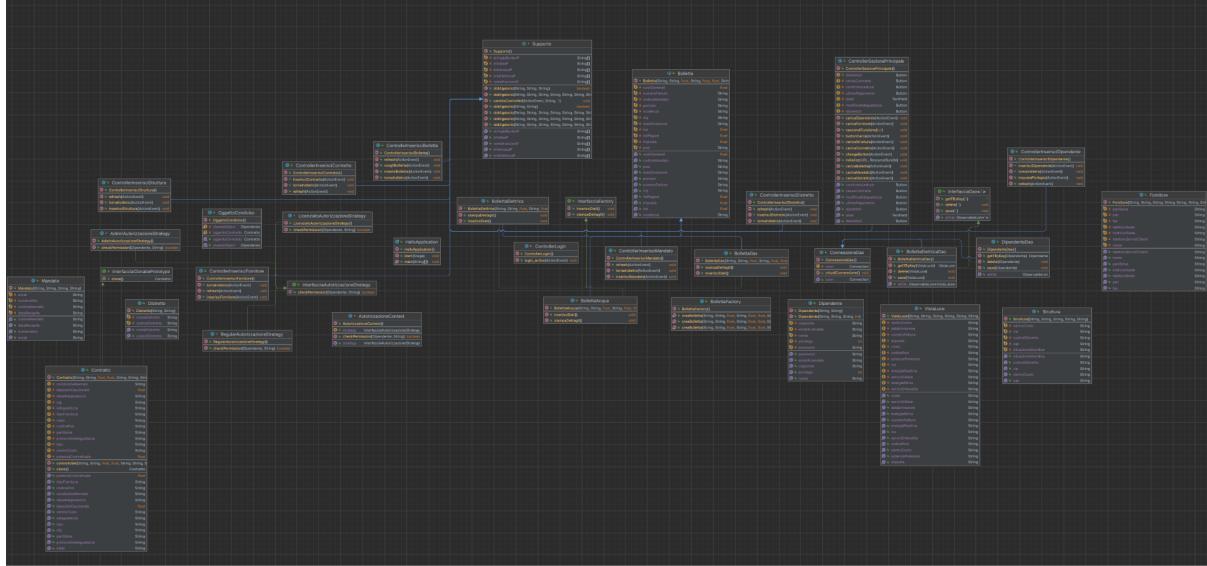
```

```

public <T> void cambioController(ActionEvent actionEvent, String pagina, T x) throws IOException {
    FXMLLoader loader = new FXMLLoader(getClass().getResource(pagina));
    Object root = loader.load();
    x = loader.getController();
    /*Questi attributi gestiscono il cambio delle scene*/
    Stage stage = (Stage) ((Node) actionEvent.getSource()).getScene().getWindow();
    Scene scene = new Scene((Parent) root);
    stage.setScene(scene);
    stage.show();
}

```

4 Diagramma delle classi



per una maggiore comprensione abbiamo allegato il jpg dello schema.

5 Parti rilevanti del codice

Oltre i pattern precedentemente mostrati abbiamo ritenuto opportuno mostrare il funzionamento nello specifico di altri elementi sviluppati.

5.1 Funzione di hashing

Questa funzione viene richiamata per motivi di sicurezza di fatti quando viene inserito un nuovo dipendente la password sarà generata automaticamente dal software in modo che nessuno possa conoscerla se non l'utente stesso, tale password verrà spedita via email(futura implementazione), questa funzione può essere richiamata solo da un utente Admin, il quale avrà il permesso di inserire e modificare dipendenti .

Attenzione a non confondersi per il nome, in quanto si è scelto di utilizzare un nome sbagliato sarebbe opportuno chiamarla “Genera Password”

Questo codice crea una funzione denominata "FUNZIONE DI HASHING" in SQL che prende come parametri di input il numero di numeri, il numero di caratteri speciali, il numero di caratteri minuscoli e il numero di caratteri maiuscoli. La funzione restituisce una stringa di lunghezza totale pari alla somma dei quattro parametri di input.

La funzione utilizza un ciclo "loop" per generare una stringa casuale utilizzando la funzione "dbms_random.string" e quindi utilizza la funzione "regexp_count" per contare il numero di caratteri minuscoli, maiuscoli e numeri nella stringa generata. Se il numero di caratteri minuscoli, maiuscoli e numeri nella stringa generata corrisponde ai parametri di input forniti, il ciclo si interrompe e la stringa viene restituita come output. Se il ciclo raggiunge il numero massimo di iterazioni (impostato a 500) senza che la stringa generata soddisfi i criteri di input, la funzione restituirà una stringa vuota.

```

CREATE OR REPLACE FUNCTION FUNZIONE_DI_HASHING(
    nums IN NUMBER,
    ch_sp IN NUMBER,
    ch_min IN NUMBER,
    ch_maiusc IN NUMBER) RETURN VARCHAR2
IS
    lun number := nums + ch_sp + ch_min + ch_maiusc;
    pw varchar2(200);
    v_iterations number := 0;
    max_iter number := 500;
BEGIN
    loop
        pw := dbms_random.string('P',lun);
        v_iterations := v_iterations + 1;
        exit when (regexp_count(pw,'[a-z]') = ch_min
                    and regexp_count(pw,'[A-Z]') = ch_maiusc
                    and regexp_count(pw,'[0-9]') = nums)
                or v_iterations=max_iter;
    end loop;
    if v_iterations = max_iter THEN
        pw := '';
    end if;
    return(pw);
END;

```

```

public void save(Dipendente x) {
    try{
        //prima di inserire un dipendente creiamo la password, questo avviene richiamando la funzione di hashing
        //di fatto la password quando viene creata un nuovo dipendente per essere sicura
        // non verrà inserita ma creata da una funzione di hashing dal database
        System.out.println("SONO DENTRO DIPENDENTE DAO");
        String call= "{call ?:=FUNZIONE_DI_HASHING(?, ?, ?, ?)}";
        CallableStatement pstmt = getConn().prepareCall(call);
        pstmt.registerOutParameter( parameterIndex: 1, Types.VARCHAR);
        pstmt.setInt( parameterIndex: 2, x: 3);
        pstmt.setInt( parameterIndex: 3, x: 3);
        pstmt.setInt( parameterIndex: 4, x: 4);
        pstmt.setInt( parameterIndex: 5, x: 4);
        pstmt.executeUpdate();
        System.out.println("HO SUPERATO LA CHIAMATA DI FUNZIONE");
        System.out.println("SONO ANCORA IN GARA");
        String pw = pstmt.getString( parameterIndex: 1);
        System.out.println(pw);
        //query
        String query = "INSERT INTO DIPENDENTE (EMAILAZIENALE,NOME,COGNOME,PRIVILEGIO,CHIAVE_ACCESSO) VALUES (?,?,?,?,?)";
        //apriamo la connessione e crea un oggetto per poter eseguire la query
        PreparedStatement stmt = getConn().prepareStatement(query);
        //chiudiamo la funzione dopo aver richiamato la procedura contenente la funzione che genera una password forte di 16 cifre sicure
        System.out.println(getConn().isClosed());
        System.out.println("HO ESEGUITO LA QUERY");
        stmt.setString( parameterIndex: 1,x.getEmailAziendale());
        stmt.setString( parameterIndex: 2,x.getName());
        stmt.setString( parameterIndex: 3,x.getCognome());
        stmt.setString( parameterIndex: 4, Integer.toString(x.getPrivilegio()));
        stmt.setString( parameterIndex: 5,pw);
        //Esegue la query
        int rowsAffected = stmt.executeUpdate();
        System.out.println("HO FINITO");
        chiudiConnessione();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    }
}

```

5.2 Vista

Questo codice SQL crea o sostituisce una vista chiamata TABELLONE STRUTTURA LUCE utilizzando una query di selezione. La vista seleziona i dati dalle tabelle STRUTTURA, CONTRATTO, BOLLETTA e BOLLETTA ELETTRICA e li unisce in base alle relazioni tra le colonne CENTRO COSTO, CIG e NUMERO FATTURA. La vista seleziona tutte le colonne specificate nella query, tra cui CENTRO COSTO, DATA EMISSIONE, NUMERO FATTURA, COSTO, ENERGIA ATTIVA, ENERGIA REATTIVA, POTENZA PRELEVATA, SERVIZI DI VENDITA, SERVIZI RETE, IMPOSTE e IVA.

```
CREATE OR REPLACE VIEW TABELLONE_STRUTTURA_LUCE AS SELECT
STRUTTURA.CENTRO_COSTO, BOLLETTA.DATA_EMISSIONE,BOLLETTA.NUMERO_FATTURA,
BOLLETTA_ELETTRICA.COSTO, BOLLETTA_ELETTRICA.ENERGIA_ATTIVA,
BOLLETTA_ELETTRICA.ENERGIA_REATTIVA, BOLLETTA_ELETTRICA.POTENZA_PRELEVATA,
SERVIZI_DI_VENDITA, SERVIZI RETE, BOLLETTA.IMPOSTE, BOLLETTA.IVA FROM STRUTTURA
JOIN CONTRATTO ON CONTRATTO.CENTRO_COSTO = STRUTTURA.CENTRO_COSTO
JOIN BOLLETTA ON BOLLETTA.CIG = CONTRATTO.CIG
JOIN BOLLETTA_ELETTRICA ON BOLLETTA_ELETTRICA.NUMERO_FATTURA = BOLLETTA.NUMERO_FATTURA;
```

In java

```
public ObservableList<VistaLuce> getAllObj() {
    try{
        // Creo la query per recuperare tutti i dati dalla tabella TABELLONE_STRUTTURA_LUCE
        String query = "SELECT * FROM TABELLONE_STRUTTURA_LUCE";
        PreparedStatement pstmt = getConn().prepareStatement(query);
        // Creo una lista osservabile per i dati delle bollette elettriche
        ObservableList<VistaLuce> data = FXCollections.observableArrayList();
        ResultSet rs = pstmt.executeQuery();
        // Ciclo per recuperare tutti i dati dalla tabella
        while(rs.next()){
            // Recupero i dati dalla tabella
            System.out.println("sono entrato nel ciclo while");
            String centroCosto = rs.getString( columnLabel: "CENTRO_COSTO");
            String dataEmissione = rs.getString( columnLabel: "DATA_EMISSIONE");
            String numeroFattura = rs.getString( columnLabel: "NUMERO_FATTURA");
            String costo = rs.getString( columnLabel: "COSTO");
            String energiaAttiva= rs.getString( columnLabel: "ENERGIA_ATTIVA");
            String energiaReattiva = rs.getString( columnLabel: "ENERGIA_REATTIVA");
            String potenzaPrelevata = rs.getString( columnLabel: "POTENZA_PRELEVATA");
            String serviziDiVendita = rs.getString( columnLabel: "SERVIZI_DI_VENDITA");
            String serviziDiRete= rs.getString( columnLabel: "SERVIZI RETE");
            String imposte = rs.getString( columnLabel: "IMPOSTE");
            String iva = rs.getString( columnLabel: "IVA");
            String codicePod= rs.getString( columnLabel: "CODICE_POD");
            VistaLuce vistaluce = new VistaLuce(centroCosto,dataEmissione,numeroFattura,costo,energiaAttiva,energiaReattiva,
                                              potenzaPrelevata,serviziDiVendita,serviziDiRete,imposte,iva,codicePod);
            data.add(vistaluce);
        }
        System.out.println("ho terminato");
        return data;
    }catch(SQLException e){
        System.out.println("ECCEZIONE SQL");
        return null;
    }
}
```

Le funzione precedenti in SQL sono riportate solo per rendere più chiaro il loro utilizzo e come sono implementate

5.3 Change Button

Questo codice rappresenta un metodo JavaFX per gestire gli eventi su una serie di pulsanti. Il metodo prende in input un evento di tipo ActionEvent, che viene generato quando un utente clicca su uno dei pulsanti. Il metodo crea un nuovo oggetto ControllerSezionePrincipale e utilizza questo oggetto per ottenere un array di stringhe chiamato "stringIdButton". Successivamente, crea un array di oggetti di tipo Button chiamato "idButton" che contiene i riferimenti ai pulsanti dell'interfaccia utente. Il metodo utilizza un ciclo "for" per esaminare ogni bottone nell'array "idButton". Se il bottone corrente corrisponde alla fonte dell'evento, cioè se è stato premuto, il metodo modifica lo stile del bottone, imposta l'ID del label e il suo testo, e chiama un metodo "nascondiFunzione" passando come parametro l'indice del bottone premuto. Se l'indice del bottone premuto è 6, il metodo esegue un controllo di autorizzazione per verificare se l'utente ha i privilegi necessari per accedere alla funzionalità associata al bottone. Se l'utente non ha i privilegi necessari, viene visualizzata una finestra di errore. Se l'indice del bottone premuto è 0 o 1, la tabella viene mostrata, altrimenti viene nascosta. Per ogni bottone che non è stato premuto, il metodo modifica lo stile del bottone per renderlo bianco con testo nero.

```

public void changeButton(ActionEvent actionEvent) {
    ControllerSezionePrincipale controller = new ControllerSezionePrincipale();
    String [] stringIdButton=controller.getStringIdButtonP();
    //array di buttoni di supporto
    Button [] idButton={

        pod,
        n_fattura,
        distretto,
        fornitore,
        struttura,
        contratto,
        dipendente,
        mandato
    };

    int i =0;
    //capiamo quale bottone è stato premuto e a seconda di ciò adattiamo la pagina
    for(Button singleButton:idButton)
    {

        if (singleButton==actionEvent.getSource()){
            if(i==6){
                Dipendente myObject = OggettoCondiviso.getSharedObject();
                AutorizzazioneContext context = new AutorizzazioneContext();
                AdminAutorizzazioneStrategy strategy = new AdminAutorizzazioneStrategy();
                context.setStrategy(strategy);
                //nel cas si provi ad inserire un utente controlliamo se l'utente ha accesso
                if(context.checkPermission(myObject,Integer.toString(myObject.getPrivilegio()))){
                    singleButton.setStyle("-fx-background-color: gray; -fx-border-color: black; -fx-text-fill: white");
                    label.setId("label_"+stringIdButton[i]);
                    System.out.println(label.getId());
                    label.setPromptText("Ricerca per "+stringIdButton[i]);
                    nascondiFunzione(i);

                }
                else{
                    //accesso negato
                    Alert alert =new Alert(Alert.AlertType.ERROR, "PERMESSO NEGATO NON HAI L'ACCESO");
                    alert.showAndWait();
                }
            }else{
                //caso in cui mostriamo la tabella di vista
                if(i==0 || i==1){
                    tabella.setVisible(true);
                }
                else
                    tabella.setVisible(false);
                //nascondiamo eventuali bottoni fuzioni non necessari
                singleButton.setStyle("-fx-background-color: gray; -fx-border-color: black; -fx-text-fill: white");
                label.setId("label_"+stringIdButton[i]);
                System.out.println(label.getId());
                //setiamo la laber di ricerca in modo che mostri il corretto Prompttext
                label.setPromptText("Ricerca per "+stringIdButton[i]);
                nascondiFunzione(i);
            }
            else{
                singleButton.setStyle("-fx-background-color: white; -fx-border-color: black;-fx-text-fill: black");
            }
        }
        i++;
    }
}

```

6 Gestione degli Errori

All'interno del nostro software sono stati gestiti gli errori di situazioni critiche.

Nel nostro caso eventuali errori di input genereranno un errore di SQLException dato che all'interno del database sono già presenti delle regole di business (trigger) che definiscono le regole del sistema. Inoltre il sistema è stato progettato per richiamare alcune funzioni scritte in SQL, se si passano ad esempio date errate, dove richiesto sarà il database a generare errori, noi attraverso i try catch, catturiamo gli eventuali errori generati da SQL. Inoltre ci sono altri tipi di errori che abbiamo gestito da software in quanto da database non potevano essere gestiti, come la connessione a quest'ultimo o l'inserimento di un dipendente. Sono stati gestite anche da software le operazioni a cui i diversi utenti hanno accesso. \

```
public abstract class ConnessioneDao {  
    3 usages  
    private Connection conn;  
  
    {  
        try {  
            // Carica il driver JDBC per Oracle  
            Class.forName(className: "oracle.jdbc.driver.OracleDriver");  
  
            // Crea la connessione al database  
            conn = DriverManager.getConnection(url: "jdbc:oracle:thin:@localhost:1521:xe", user: "c##tecnico_dipendente", password: "tec  
        } catch (ClassNotFoundException e) {  
            e.printStackTrace();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
  
    // metodo getter per ottenere l'oggetto Connection  
    8 usages new *  
    public Connection getConn() { return conn; }  
  
    // metodo per chiudere la connessione  
    3 usages new *  
    public void chiudiConnessione() throws SQLException {  
        try {  
            conn.close();  
        } catch (SQLException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Caso in cui controlliamo da software il privilegio dell'utente ad eseguire una determinata azione.

```
if(context.checkSelfPermission(myObject, Integer.toString(myObject.getPrivilegio()))){
    singleButton.setStyle("-fx-background-color: gray; -fx-border-color: black; -fx-text-fill: white");
    label.setId("label_"+stringIdButton[i]);
    System.out.println(label.getId());
    label.setPromptText("Ricerca per "+stringIdButton[i]);
    nascondiFunzione(i);
}

else{
    //accesso negato
    Alert alert =new Alert(Alert.AlertType.ERROR, s: "PERMESSO NEGATO NON HAI L'ACCESO");
    alert.showAndWait();
}
```

In questo caso nella classe DipendenteDao abbiamo messo in evidenza soltanto i try catch che gestiscono gli errori precedentemente menzionati, in quanto l'intera classe è già stata riportata nella sezione [Design Pattern Dao](#)

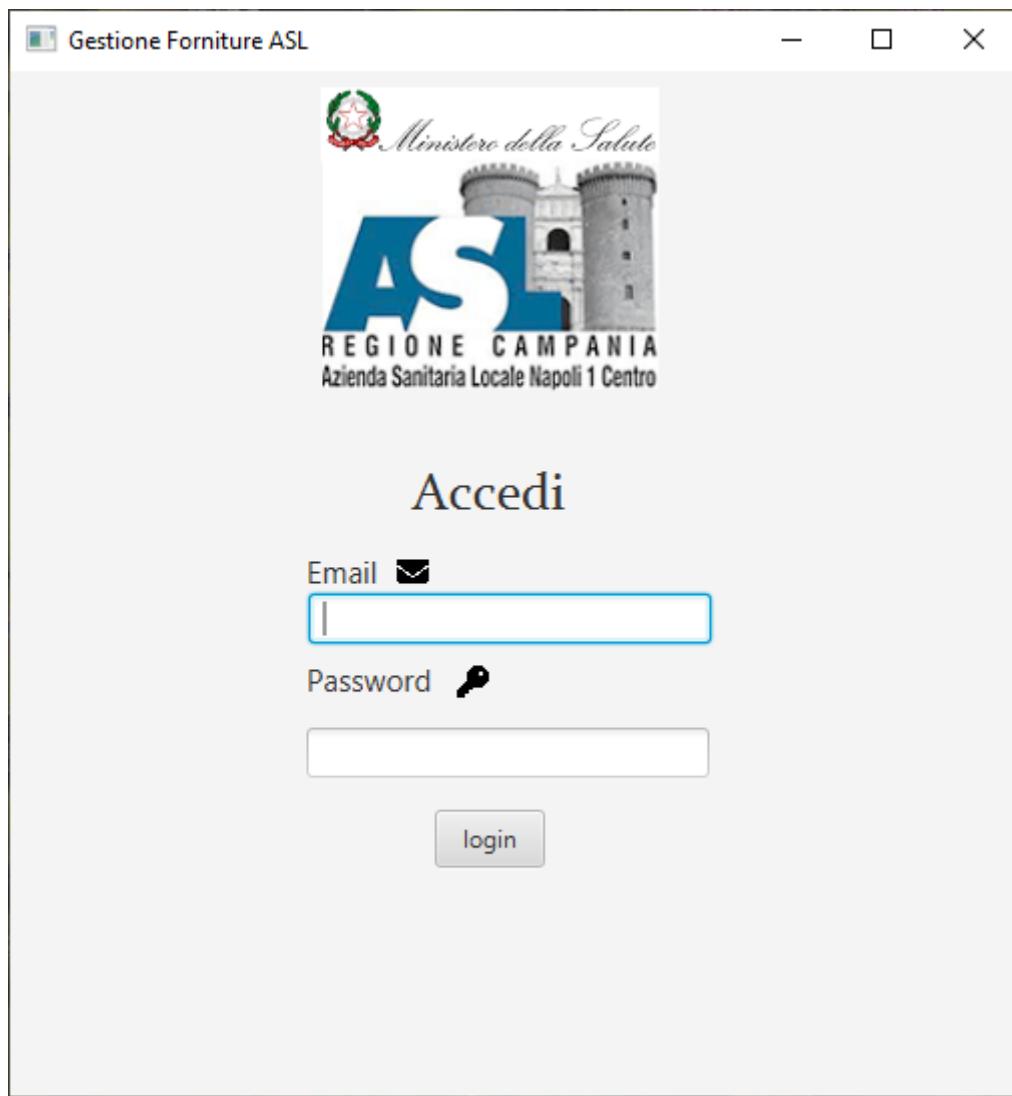
```
public Dipendente getTByKey(Dipendente x) {
    try{...} catch (SQLException e) {
        throw new RuntimeException(e);
    }
    return null;
}

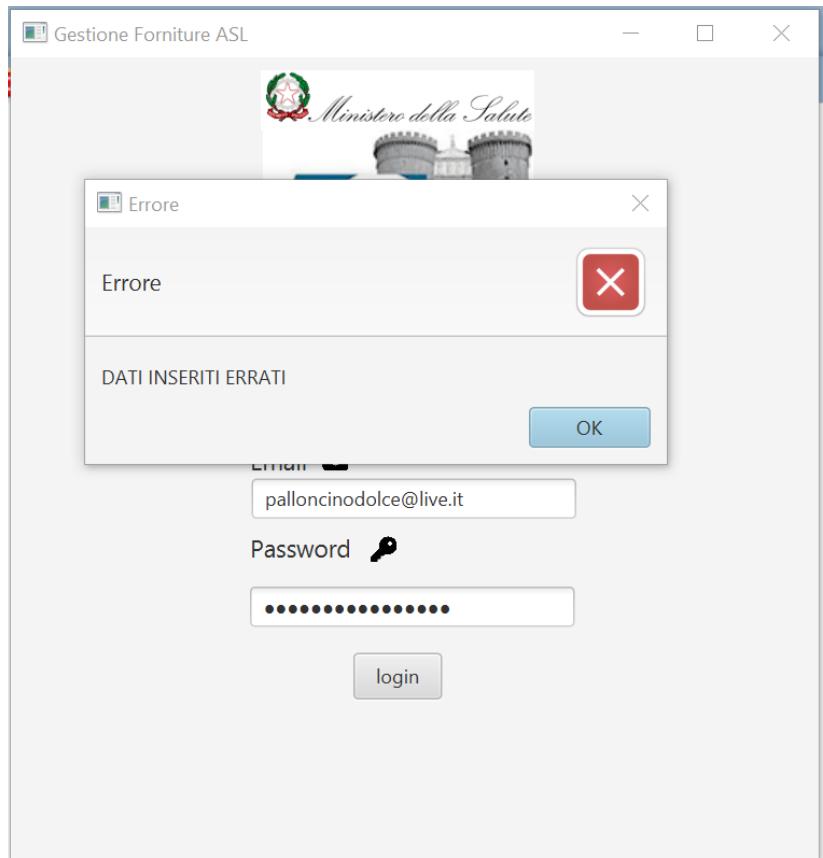
//il metodo esegue la ricerca di un dipendente tramite key e ritorna l'eventuale dipendente trovato
//metodo di salvataggio dei dati all'interno del database, il dipendente inserito
//viene salvato all'interno del database
1 usage new *
@Override
public void save(Dipendente x) {
    try{...} catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
no usages new *
@Override
// L'ELIMINAZIONE è SOLO LOGICA SETTIAMO IL PRIVILECIO A 0 IN MODO CHE NON POSSA PIÙ ENTRARE
//questo è il metodo di cancellazione di un dipendente
//dopo che il privileggio viene settato a 0 non si hanno più i permessi per entrare
public void delete(Dipendente x) {
    try{...} catch (SQLException e) {
        throw new RuntimeException(e);
    }
}
```

In questo caso nella classe BollettaElettricaDao abbiamo messo in evidenza soltanto i try catch che gestiscono gli errori precedentemente menzionati, in quanto l'intera classe è già stata riportata nella sezione [BollettaElettricaDao](#)

```
public class BollettaElettricaDao extends ConnessioneDao implements InterfacciaDao<VistaLuce> {
    2 usages new *
    @Override
    public ObservableList<VistaLuce> getAllObj() {
        try{...} catch(SQLException e){
            System.out.println("ECCEZIONE SQL");
            return null;
        }
    }
}
```

7 MockUp





Gestione Forniture ASL

Ricerca	Ricerca per distretto	
<input type="button" value="Pod"/>	<input type="text"/>	
<input type="button" value="N° fattura"/>	<input type="button" value=""/>	
<input type="button" value="Distretto"/>	<input type="button" value=""/>	
<input type="button" value="Fornitore"/>	<input type="button" value=""/>	
<input type="button" value="Struttura"/>	<input type="button" value=""/>	
<input type="button" value="Contratto"/>	<input type="button" value=""/>	
<input type="button" value="Dipendente"/>	<input type="button" value=""/>	
<input type="button" value="Mandato"/>	<input type="button" value=""/>	
Inserisci		
<input type="button" value="Bolletta"/>	<input type="button" value=""/>	
<input type="button" value="Mandato"/>	<input type="button" value=""/>	
<input type="button" value="Contratto"/>	<input type="button" value=""/>	
<input type="button" value="Fornitore"/>	<input type="button" value=""/>	
<input type="button" value="Struttura"/>	<input type="button" value=""/>	
<input type="button" value="Distretto"/>	<input type="button" value=""/>	
<input type="button" value="Dipendente"/>	<input type="button" value=""/>	
	<input type="button" value="Iva Totale"/>	<input type="button" value="Potenza Media Annua"/>
	<input type="button" value="Potenza Annua"/>	<input type="button" value="Totale Pagato"/>
	<input type="button" value="Consumo Annuo"/>	<input type="button" value="Spese Totali"/>

Gestione Forniture ASL

Ricerca	Ricerca per numero pod	
<input type="button" value="Pod"/>	<input type="text"/>	
<input type="button" value="N° fattura"/>	<input type="button" value=""/>	
<input type="button" value="Distretto"/>	<input type="button" value=""/>	
<input type="button" value="Fornitore"/>	<input type="button" value=""/>	
<input type="button" value="Struttura"/>	<input type="button" value=""/>	
<input type="button" value="Contratto"/>	<input type="button" value=""/>	
<input type="button" value="Dipendente"/>	<input type="button" value=""/>	
<input type="button" value="Mandato"/>	<input type="button" value=""/>	
Inserisci		
<input type="button" value="Bolletta"/>	<input type="button" value=""/>	
<input type="button" value="Mandato"/>	<input type="button" value=""/>	
<input type="button" value="Contratto"/>	<input type="button" value=""/>	
<input type="button" value="Fornitore"/>	<input type="button" value=""/>	
<input type="button" value="Struttura"/>	<input type="button" value=""/>	
<input type="button" value="Distretto"/>	<input type="button" value=""/>	
<input type="button" value="Dipendente"/>	<input type="button" value=""/>	
	<input type="button" value="Ultimo Pagamento"/>	<input type="button" value="Adeguetza"/>
	<input type="button" value="Cessa Contratto"/>	<input type="button" value="Confronto Lettura"/>

Gestione Forniture ASL

Ricerca

- Pod
- N° fattura
- Distretto
- Fornitore
- Struttura
- Contratto
- Dipendente
- Mandato

Inserisci

- Bolletta
- Mandato
- Contratto
- Fornitore
- Struttura
- Distretto
- Dipendente

Ricerca per dipendente

Cambio Permessi Crea Nuova Chiave

Caso in cui il dipendente non ha accesso alla sezione Ricerca Dipendenti

Gestione Forniture ASL

Ricerca

- Pod
- N° fattura
- Distretto
- Fornitore
- Struttura
- Contratto
- Dipendente
- Mandato

Inserisci

- Bolletta
- Mandato
- Contratto
- Fornitore
- Struttura
- Distretto
- Dipendente

Ricerca per numero pod

Numero Pod	Codice	...
132384	14146.2	1
132384	14146.2	2
132384	14146.2	2
132384	14146.2	2
132384	14146.2	2
132384	14146.2	2

Errore

PERMESSO NEGATO NON HAI L'ACCESO

OK

Ultimo Pagamento Adeguatezza Cessa Contratto Confronta Letture

Gestione Forniture ASL

← Selezione il tipo Elettrica ▾

NUMERO FATTURA*	CIG*
PERIODO*	POD*
TOTALE DA PAGARE*	POTENZA PRELEVATA*
DATA EMISSIONE*	SERVIZI DI VENDITA
IVA*	CTS
IMPOSTE*	SERVIZI DI RETE
SCADENZA*	ENERGIA ATTIVA*
COSTI GENERALI*	ENERGIA REATTIVA*
CODICE MANDATO*	

Inserisci

Gestione Forniture ASL

Selezione il tipo

NUMERO FATTURA*	PERIODO*	POD*
TOTALE DA PAGARE*	POTENZA PRELEVATA*	SERVIZI DI VENDITA
DATA EMISSIONE*	CTS	SERVIZI DI RETE
IVA*	ENERGIA ATTIVA*	ENERGIA REATTIVA*
IMPOSTE*	SCADENZA*	COSTI GENERALI*
CODICE MANDATO*	Inserisci	

Gestione Forniture ASL

← Selezione il tipo Gas ▾

NUMERO FATTURA*	CIG*
PERIODO*	POD*
TOTALE DA PAGARE*	TRASPORTO E GESTIONE*
DATA EMISSIONE*	FORNITURA GAS*
IVA*	CONSUMO PDR*
IMPOSTE*	ONERI DI SISTEMA*
SCADENZA*	
COSTI GENERALI*	
CODICE MANDATO*	

Inserisci

Gestione Forniture ASL

← Mandato

|

NUMERO ATTO*
DATA RECAPITO*

TOTALE MANDATO:

INSERISCI



Contratto

LSD24

PARTITA IVA*

CENTRO COSTO*

CODICE POD*

POTENZA CONTRATTUALE*

CONDIZIONE MERCATO*

DEPOSITO CAUZIONALE

TIPO FORNITURA*

STATO*

ACQUA LUCE O GAS*

ADEGUATEZZA*

PROTOCOLLO ADEGUATEZZ

DATA ADEGUATEZZA

INSERISCI



Contratto

LSD24

PARTITA IVA*

CENTRO COSTO*

CODICE POD*

POTENZA CONTRATTUALE*

CONDIZIONE MERCATO*

DEPOSITO CAUZIONALE

TIPO FORNITURA*

STATO*

ACQUA LUCE O GAS*

ADEGUATEZZA*

PROTOCOLLO ADEGUATEZZ

DATA ADEGUATEZZA

Inserisci i campi obbligatori

INSERISCI



Gestione Forniture ASL



Fornitore

PARTITA IVA*

3331787999

TELEFONO SERVIZIO CLI

INDIRIZZO SEDE*

PEC

FAX

NOME*

INSERISCI



Gestione Forniture ASL

- □ ×



Struttura

CAP

VIA*

SITUAZIONE GIURIDICA

CODICE DISTRETTO*

INSERISCI



Gestione Forniture ASL



Distretto

NOME DISTRRETTO*

INSERISCI

Gestione Forniture ASL

Dipendente

riccardo@live.it

Riccardo

Spinsa

Imposta privilegio

ADMIN

INSERISCI

Caso in cui il dipendente non ha accesso alla sezione Inserimento Dipendenti

Gestione Forniture ASL

Ricerca

Pod

N° fattura

Distretto

Fornitore

Struttura

Contratto

Dipendente

Mandato

Inserisci

Bolletta

Mandato

Contratto

Fornitore

Struttura

Distretto

Dipendente

Ricerca per numero pod

Número fat...	Codice	Azi...	Codice Prof.	Centro Custo...	Data Inserzione
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00
102944	14162	100	400	LDS2910206	2023-02-28 00:00:00
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00
102944	14162	100	400	LDS2910206	2022-02-28 00:00:00

OK

PERMESSO NEGATO NON HAI L'ACCESO

8 Testing

La fase di testing è una parte essenziale del processo di sviluppo software che mira a garantire che il prodotto finito soddisfi i requisiti e funzioni correttamente. Proprio per questo durante lo sviluppo del nostro software abbiamo interagito il più possibile con il cliente che utilizzerà in prima persona con il sistema.

Lo scopo del testing è la valutazione dell'usabilità del sistema ed inoltre si eccepisce che il software è ancora in fase di sviluppo e per ciò lo anche la fase di testing. È stato richiesto al cliente di esprimere le proprie opinioni sull'usabilità del sistema e di eccepire eventuali implementazioni future.

Di seguito riportiamo:

Il campo input del contratto può assumere solo i 3 valori riportati(acqua, luce, gas).

Il campo input adeguatezza del contratto, può assumere solo il valore Si o No.

ASL eccepisce: tali campi non devono essere campi d'inserimento libero, ma menù a tendina dove dev'essere possibile scegliere la parola chiave da inserire, come sviluppato per la scelta del tipo della bolletta.

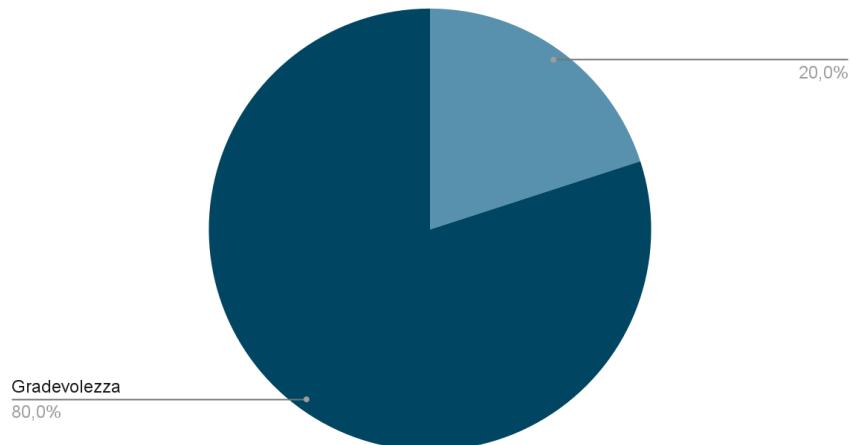
Inoltre in base al campo Adeguatezza il campo Data Adeguatezza dovrà comparire o sparire in base al suo valore.

Asl eccepisce: quando si inseriscono determinati valori di determinati campi correlati tra loro(chiavi esterne) dev'essere possibile selezionare/filtrare tra la lista di quelli già inseriti.

esempio:

partita iva nell'inserimento del contratto è una chiave esterna del fornitore, l'utente vorrebbe poter scegliere tra quelli già presenti nel database.

Points scored



9 Java Doc

Il Javadoc è uno strumento per generare documentazione per il codice Java. Permette agli sviluppatori di inserire commenti nel codice sorgente che verranno poi utilizzati per generare documentazione in formato HTML. Questi commenti devono essere scritti utilizzando un formato specifico, iniziando con `/**` e terminando con `*/`. Il Javadoc può essere utilizzato per generare documentazione per classi, metodi, variabili e altri elementi del codice.

Abbiamo inserito il javadoc come file allegato.

10 Future implementazioni

- Gestione e ricerca di tutte le entità presenti nel database.
- Utilizzo di tutte le funzioni necessarie implementate già nel database, ad esempio il calcolo dell'IVA.
- Implementazione della funzionalità di suggerimento campi come richiesto dal cliente. Il pattern Observer può essere utilizzato per implementare un meccanismo di notifica in tempo reale per le modifiche ai dati del database. In questo caso, il database funge da "oggetto osservato" e gli script o i software che effettuano le ricerche sono gli "osservatori". Quando ci sono modifiche ai dati del database, gli osservatori vengono notificati e possono aggiornare la loro visualizzazione dei dati in modo tempestivo.
- Invio della password in modo sicuro tramite email, generata dall'utente Admin, al corrispettivo nuovo dipendente inserito nel sistema
- Implementazione di un sistema di hashing per proteggere le password salvate nel database, questo ci protegge da eventuali attacchi al database e da sql injection, di conseguenza l'utente inserirà la password in chiaro e dovremo utilizzare lo stesso algoritmo per confrontare se la chiave generata è la stessa di quella presente nel database criptata.
- Aggiungere la richiesta di recupero password in caso un dipendente dovesse smarirsi, per gli utenti con privilegio Regular la password potrà essere resettata solo da un utente Admin. Nel caso di password dimenticata da un Admin la password potrà essere richiesta via email con autenticazione a due fattori obbligatoria, l'invio di tale email dovrà essere eseguita in maniera sicura.

Abbiamo quindi raggiunto i risultati posti in fase di progettazione.

