Università Politecnica delle Marche – Facoltà di Ingegneria
INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

APPLICAZIONE PER LA GESTIONE DI UN MAGAZZINO DI UN SUPERMERCATO
A.A. 2016 - 2017

Identificativo: **OOP1617Gruppo06**
Iezzi Valerio (matr. 1078021)
Chelli Giacomo (matr. 1069659)
Genovesi Riccardo (matr. 1067322)
Luciani Mattia (matr. 1069310)

Progetto di un'applicazione per la gestione di un magazzino di un supermercato

SOMMARIO

1. Introduzione 3

2. Analisi dei requisiti 4
   2.1 Diagramma dei casi d'uso 4

3. Analisi orientata ai dati 6
   3.1 Diagramma delle classi 6

4. Progettazione del database 10

5. Struttura del progetto 12
   5.1 Struttura generale 12
   5.2 Contenuto della cartella "src" 13

6. IL design pattern MVC 14
   6.1 Struttura 14
   6.2 Implementazione 15
   6.3 View 16
      6.3.1 Utilizzo di Swing 17
   6.4 Model 22
   6.5 Controller 24
      6.5.1 Eventi e listener 25

7. Strumenti di programmazione utilizzati 30
   7.1 Incapsulamento 30
   7.2 Ereditarietà 31
   7.3 Interfacce 33
   7.4 Eccezioni 35
   7.5 Notazioni finali 36

1. INTRODUZIONE

Il progetto è stato sviluppato nell'ottica di fornire aiuto agli amministratori di supermercati, in particolare ad uno specifico amministratore del supermercato Coop, nella gestione del proprio magazzino e dei suoi dipendenti ed inoltre fornire al cliente del supermercato la possibilità di cercare il prodotto in vendita attraverso diversi filtri, visualizzare le offerte in corso e visionare la scaffalatura del prodotto stesso.
L'applicazione permette agli amministratori di inserire i propri prodotti nel database associato. Successivamente sarà possibile cercare i prodotti inseriti, utilizzando diversi filtri, oppure procedere alla modifica degli stessi. Inoltre sarà possibile inserire e modificare i dipendenti che forniscono servizio al supermercato associando loro il rispettivo turno lavorativo.

Il lettore viene invitato a consultare i manuali di installazione e utente, forniti a parte, al fine di:

- Il manuale di installazione permette di configurare la propria macchina sia per la consultazione e modifica del codice, che per la semplice esecuzione dell'applicazione;
- Il manuale utente mostra quelle che sono le funzionalità dell'applicazione e le possibili schermate in cui l'utilizzatore può incappare.

2. ANALISI DEI REQUISITI

Dopo aver studiato a fondo il dominio della nostra applicazione, abbiamo effettuato l'analisi dei requisiti utilizzando i diagrammi UML e in particolare il diagramma dei casi d'uso riportato in seguito.

2.1 Diagramma dei casi d'uso

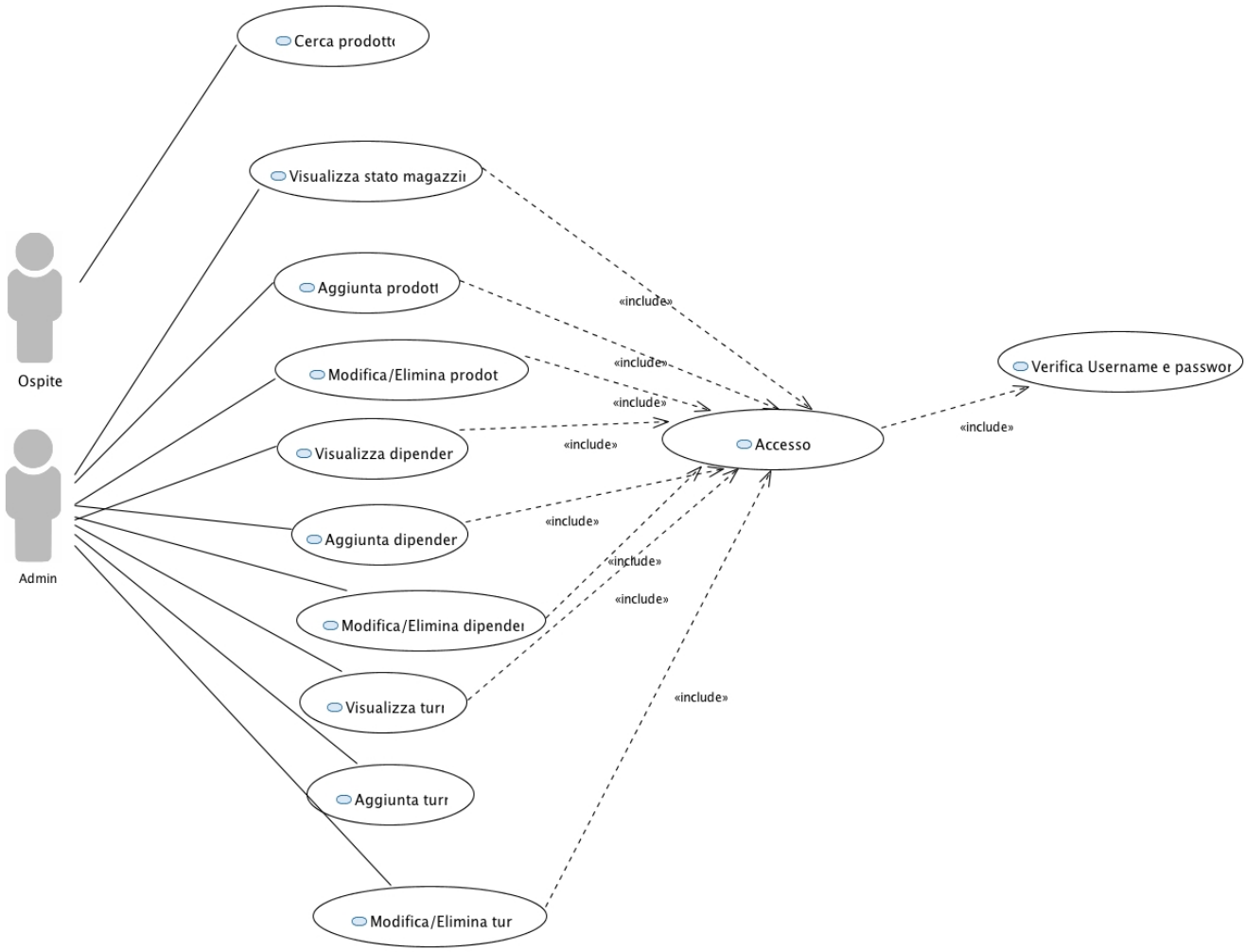

Fig. 1 – Diagramma dei casi d'uso

Come si evince dalla Fig. 1 si hanno due attori principali:

- Admin
- Ospite

L'Admin è il gestore del supermercato ed è colui che si occupa della gestione dei prodotti, dei dipendenti e dei turni.
A questo sono associati 9 casi d'uso accumunati dalla necessità di fare l'accesso all' applicazione inserendo Username e Password.
I casi d'uso in questione sono:
- Visualizza stato magazzino: si può visualizzare lo stato del magazzino tenendo sotto controllo le rimanenze di ogni singolo prodotto;
- Aggiunta prodotto: è possibile aggiungere prodotti compilando i campi obbligatori;
- Modifica/Elimina prodotto: è possibile modificare prodotti aggiunti in passato andando ad agire sui campi interessati.;
- Visualizza dipendenti: si può visualizzare la lista di tutti i dipendenti che lavorano o che hanno lavorato all'interno del supermercato;
- Aggiunta dipendenti: è possibile aggiungere dipendenti compilando i campi obbligatori;
- Modifica/Elimina dipendente: è possibile modificare dipendenti aggiunti in passato andando ad agire sui campi interessati;
- Visualizza turni: si può visualizzare la lista di tutti i turni che i dipendenti dovranno svolgere.
- Aggiunta turni: è possibile aggiungere turni andando a specificare le ore da cui è composto;
- Modifica/Elimina turni: è possibile modificare turni aggiunti in passato andando ad agire sui giorni interessati.

L'Ospite invece è il cliente del negozio che non è registrato all'interno dell'applicazione ma che vuole visualizzare i prodotti che sono contenuti nel negozio e quindi in scaffalatura (escluso il magazzino). Entra nell'applicazione senza l'utilizzo di Username e password (non possiede account).

3. ANALISI ORIENTATA AI DATI

3.1 Diagramma delle classi

Dal diagramma dei d'uso, passando attraverso numerosi raffinamenti successivi, si è arrivati alla definizione del diagramma delle classi.
(Per una migliore visualizzazione si rimanda all'allegato).

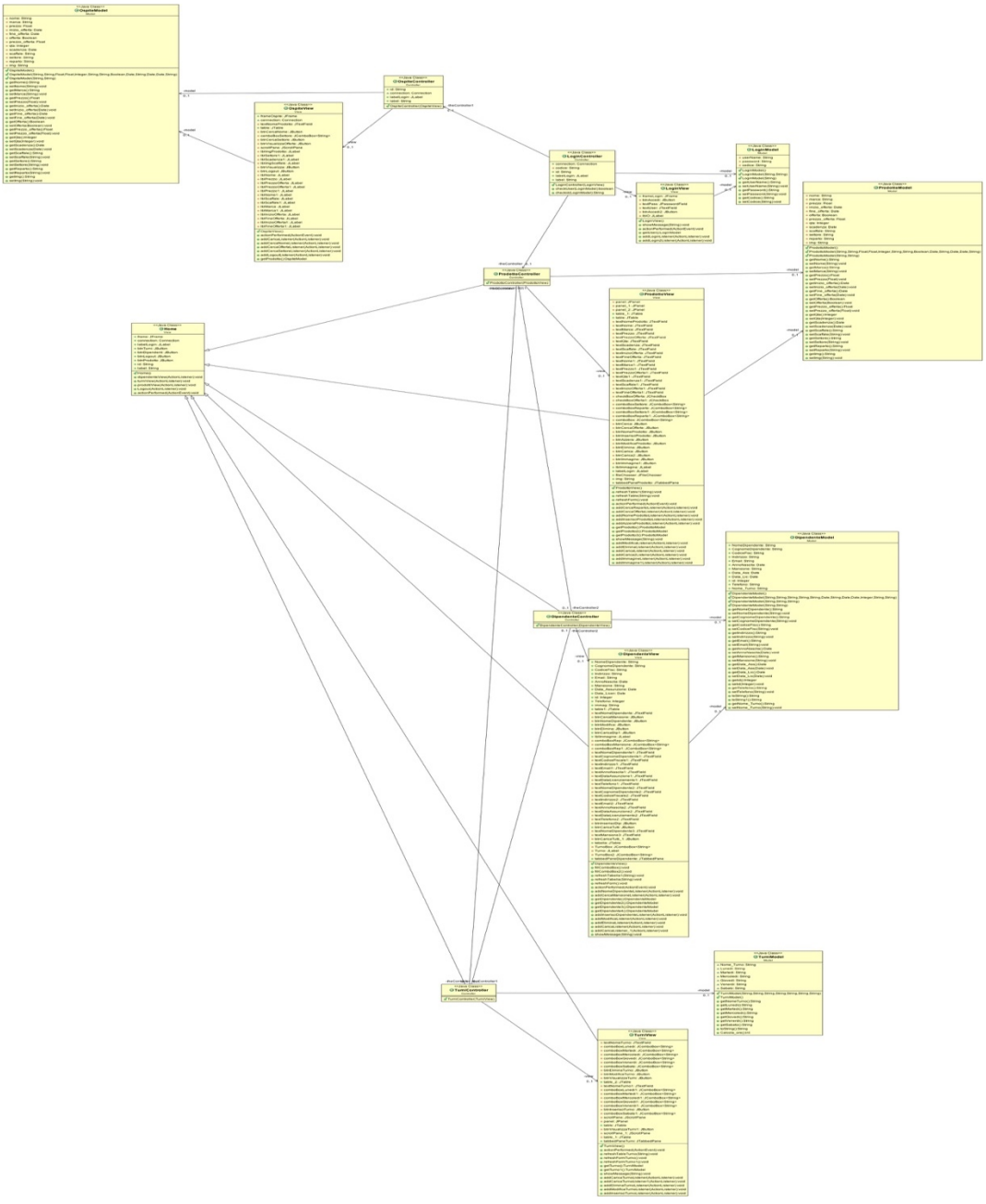

Fig. 2 – Diagramma delle Classi

Il diagramma delle classi è stato implementato secondo il pattern MVC(Model-View-Controller) di cui tratteremo nel capitolo 6.
Le classi sono raccolte all' interno di tre grandi package:
- View;
- Model;
- Controller.

Con questa struttura MVC abbiamo suddiviso 5 "macroclassi" che sono:
- Login;
- Ospite;
- Prodotto;
- Dipendente;
- Turno.

Login: è caratterizzata dalla struttura MVC (LoginView, LoginModel, LoginController). Si occupa dell'accesso nell'applicazione dell'amministratore. La LoginView è la classe che gestisce la costruzione dell'interfaccia grafica della finestra di Login. La LoginModel è la classe che permette la lettura e la memorizzazione dei dati inseriti nella finestra di Login. La LoginController è la classe che permette alla LoginModel e alla LoginView di lavorare insieme.

Ospite: è caratterizzata dalla struttura MVC(OspiteView, OspiteModel, OspiteController). Si occupa della navigazione all'interno del sito del cliente che non ha la possibilità di fare l'accesso. La OspiteView è la classe che gestisce la costruzione dell'interfaccia grafica della finestra di visualizzazione prodotti a cui si può accedere come ospite. La OspiteModel è la classe che permette la lettura e la memorizzazione dei dati inseriti nella finestra di visualizzazione prodotti. La OspiteController è la classe che permette alla OspiteModel e alla OspiteView di lavorare insieme.

Prodotto: è caratterizzata dalla struttura MVC(ProdottoView, ProdottoModel, ProdottoController). Si occupa della gestione del prodotto all'interno della applicazione, ossia permette di svolgere tutte

le azioni descritte nei casi d'uso legate al prodotto. La ProdottoView è la classe che gestisce la costruzione dell'interfaccia grafica delle finestre che riguardano la visualizzazione dello stato del magazzino, dell'inserimento del prodotto e della modifica del prodotto. La ProdottoModel è la classe che permette la lettura e la memorizzazione dei dati inseriti nelle finestre di visualizzazione, inserimento e modifica. La ProdottoController è la classe che permette alla ProdottoModel e alla ProdottoView di lavorare insieme.

Dipendente: è caratterizzata dalla struttura MVC(DipendenteView, DipendenteModel, DipendenteController). Si occupa della gestione del dipendente all'interno dell'applicazione, ossia permette di svolgere tutte le azioni descritte nei casi d'uso legate al dipendente. La DipendenteView è la classe che gestisce la costruzione dell'interfaccia grafica delle finestre che riguardano la visualizzazione della lista dei dipendenti, dell'inserimento del dipendente e della modifica del dipendente. La DipendenteModel è la classe che permette la lettura e la memorizzazione dei dati inseriti nelle finestre di visualizzazione, inserimento e modifica. La DipendenteController è la classe che permette alla DipendenteModel e alla DipendenteView di lavorare insieme.

Turno: è caratterizzata dalla struttura MVC(TurnoView, TurnoModel,TurnoController). Si occupa della gestione dei turni all'interno dell'applicazione, ossia permette di svolgere tutte le azioni descritte nei casi d'uso legate al turno. La TurnoView è la classe che gestisce la costruzione dell'interfaccia grafica delle finestre che riguardano la visualizzazione dei turni, dell'inserimento del turno e della modifica del turno. La TurnoModel è la classe che permette la lettura e la memorizzazione dei dati inseriti nelle finestre di visualizzazione, inserimento e modifica. La TurnoController è la classe che permette alla TurnoModel e alla TurnoView di lavorare insieme.

Oltre a queste 5 “macro classi” possiamo trovare una classe che ci ha aiutato in fase di costruzione delle View, la classe Home. La classe Home è stata costruita per sfruttare l’ereditarietà (proprietà a cui si darà spazio nel capitolo 7.) La classe Home è il padre di tutte le classi View (tranne che della LoginView, OspiteView) e gestisce l’interfaccia di base di ogni singola finestra interna all’applicazione.

4. PROGGETTAZIONE DATABASE

Per l'applicazione è stato utilizzato un database di tipo relazionale, dove cioè i dati vengono raccolti in tabelle. L' DBMS (DataBase Management System) utilizzato è MySQL.
Di seguito è mostrato lo scherma E/R:

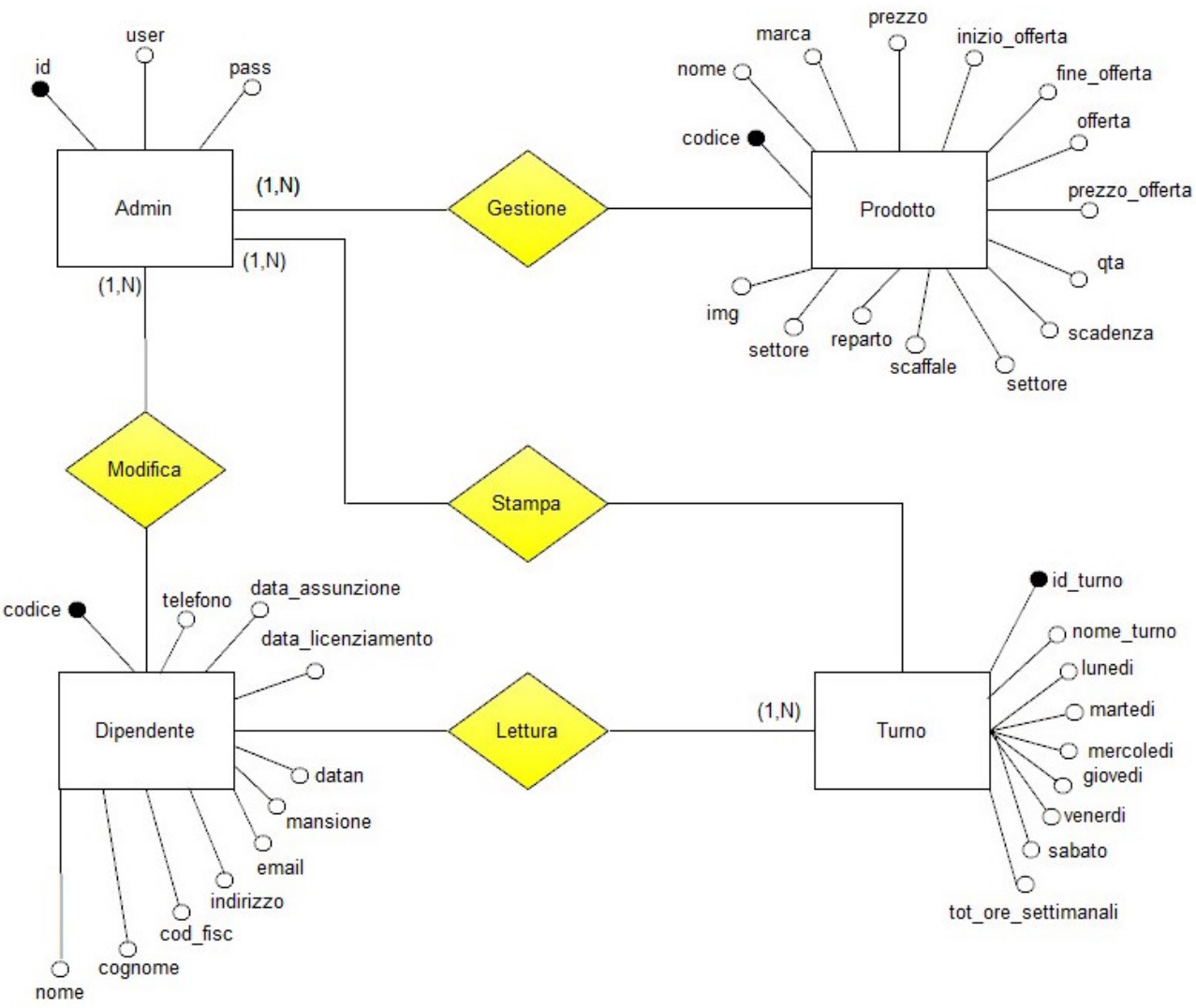

Fig. 3 – Schema E/R

Come si evince della Fig. 3, la base di dati si compone di 4 tabelle e le relative associazioni tra di loro.
La tabella *Admin* con i suoi attributi ci permette di memorizzare gli amministratori.

NB: Nella nostra applicazione l'amministratore è uno solo ed i suoi attributi sono già memorizzati e non sarà quindi necessario inserirlo/registrarsi. Attraverso l'username e la password forniti si potrà accedere alla schermata principale dell'applicazione.
Per conoscere le modalità di accesso, vi riportiamo alla consultazione del manuale utente.

La tabella *Prodotto* consente all'amministratore, una volta effettuato l'accesso, di andare a inserire e quindi memorizzare i propri prodotti con tutti gli attributi sopra descritti. In particolare si offre la possibilità di memorizzare il prodotto in offerta, con il prezzo offerta e la data di inizio e fine offerta. Degno di nota è anche la possibilità di poter memorizzare la locazione del prodotto intesa come scaffalatura a settore.
Inoltre sarà possibile modificare o eliminare i prodotti da questa tabella.

La tabella *Dipendete* consente di andare a memorizzare tutti i dati anagrafici dei dipendenti e la relativa data di assunzione e licenziamento e la rispettiva mansione che esso ricompre all'interno del supermercato.

Nella tabella *Turno* sarà possibile memorizzare i turni scelti dall'amministratore, ovvero i turni possono essere personalizzati a seconda delle diverse esigenze. Sarà possibile scegliere un nome del turno e successivamente, per ogni giorno della settimana, sarà possibile selezione mattina, pomeriggio o riposo. Per mattina è inteso il turno dalle 08:00 alle 13:00, per pomeriggio dalle 15:00 alle 20:00. Andando ad inserire il turno personalizzato verranno quindi calcolate le ore settimanali relative a quel turno, che poi sarà possibile associare ai dipendenti.

5. STRUTTURA DEL PROGETTO

5.1 Struttura generale

All'interno della cartella del progetto OOP1617 vi sono 4 sottocartelle (Fig. 4 di seguito).
La cartella *img* contiene tutte le immagini utilizzate e richiamate dall'applicazione, come ad esempio la mappa del supermercato che indica la scaffalatura o le immagini che vogliamo attribuire ai prodotti.
La cartella *Resources* contiene le librerie che andremo in fase di configurazione ad importare. In particolare esse sono: rs2xml.jar e mysql-connector-java-5.1.40-bin.jar.
La prima è una libreria che può essere usata per fare il set di risultati di una query e fornirlo in ingresso per il modello di tabella.
La seconda è una libreria con la quale MySQL fornisce la connettività per le applicazioni client sviluppate nel linguaggio di programmazione Java con MySQL Connector / J ,un driver che implementa l'API Java Database Connectivity (JDBC).
La cartella *doc* contiene il JavaDoc esportato alla fine del progetto. Ed infine la cartella *src* è il cuore del progetto.

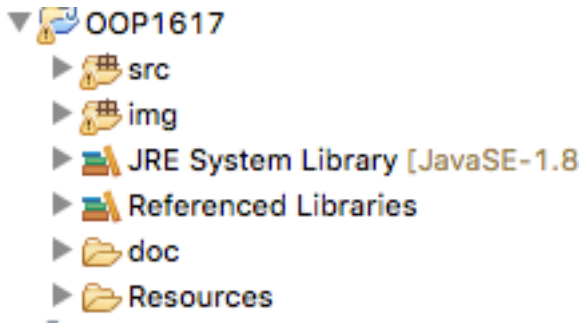

Fig. 4 – Struttura del progetto

5.2 Contenuto della cartella “src”

Veniamo ora al cuore del programma: tutto il codice sorgente si trova all’interno di src. Il progetto viene strutturato sul pattern MVC di cui andremo ad approfondire nel capitolo successivo. Il questo paragrafo andremo ad esaminare più attentamente gli altri package di cui è comporta la cartella src.

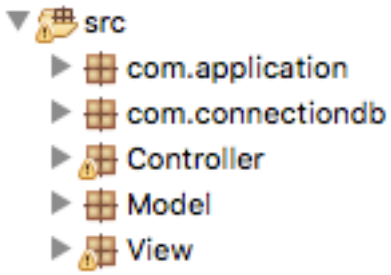

Fig. 5 – Contenuto della cartella “src”

Come mostrato in Fig. 5 la cartella src è comporta da 5 package:

- com.application: contiene la classe necessaria per l’avvio dell’applicazione.
- com.connectiondb: contiene la classe necessaria per stabilire la connessione al database (nel manuale di installazione, verrà spiegato come configurarlo).
- Controller: contiene le classi controller dell’applicazione.
- Model: contiene le classi model dell’applicazione.
- View: contiene le classi view dell’applicazione.

6. IL DESIGN PATTERN MVC

Il Model-View-Controller è un design pattern, cioè un'architettura progettuale che descrive un problema, una soluzione e le relative conseguenze. L'obbiettivo principale di un pattern è rendere la soluzione indipendente rispetto alla sua implementazione in modo tale da rendere il codice riutilizzabile e facilmente modificabile.
Sostanzialmente la "filosofia" dell'MVC è quella di progettare un software in modo da dividere l'interfaccia utente (GUI) dalla parte che rappresenta i dati e le regole che governano le modalità con cui questi vengono acceduti e dalla parte che gestisce le richieste dell'utente e le azioni che devono essere intraprese per modificare i dati stessi.
In generale l'applicazione deve separare i componenti software che implementano il modello delle funzionalità di business (Model), dai componenti che implementano la logica di presentazione (View) e di controllo che utilizzano tali funzionalità (Controller).

6.1 Struttura

Analizziamo nello specifico gli elementi fondamentali che compongono la struttura progettuale MVC.

- Model o Modello: Rappresenta l'elemento responsabile della gestione dei dati. Legge e memorizza i dati usati dall'applicazione. Per lo sviluppo del Model si utilizzano delle tipiche tecniche di progettazione le quali consentono di ottenere componenti software che rappresentano al meglio i concetti importanti del mondo reale.

- View o Vista: Responsabile della presentazione dei dati gestiti dal model in un formato specifico. La logica di visualizzazione dei dati viene gestita solo e solamente dalla View, ciò implica che questa deve fondamentalmente gestire la costruzione dell'interfaccia grafica (GUI) che rappresenta il mezzo mediante il quale gli utenti si interfacciano con il sistema. L'esecuzione dei processi richiesti

dall'utente viene infine delegata al controller dopo la cattura degli input e la scelta delle eventuali schermate da presentare.

- Controller: Costituisce l'organo che permette al Model e alla View di lavorare insieme. Ha la responsabilità di trasformare le interazioni dell'utente con la View in azioni eseguite dal Model. Il controller non rappresenta un semplice "ponte" tra View e Model, ma gestendo gli input dell'utente, selezionando le schermate della View da presentare e richiamando i processi da eseguire sul Model implementa interamente la logica di controllo del software.

6.2 Implementazione

La nostra applicazione è stata costruita sulla base dell'architettura progettuale MVC.
Diversi sono i package realizzati in cui sono raggruppate classi che hanno determinate caratteristiche e precisi compiti. Andremo in seguito ad analizzare la directory contenenti il package MVC.
Abbiamo già parlato dei package specifici nel capitolo precedente; andiamo ad analizzare la cartella src contenente il codice sorgente dell'applicazione. Notiamo che per semplicità i package gestiti con il pattern MVC sono stati inclusi direttamente nella directory src in modo tale da facilitare l'importazione delle classi dove questa è richiesta.
Analizziamo ora i singoli elementi fondamentali del MVC.

6.3 View

Abbiamo definito il componente View come l'organo responsabile della gestione della visualizzazione dei dati. La Vista consiste nella creazione di una interfaccia grafica con cui l'utente può interagire. Per questo motivo è costituita da una serie di schermate che verranno visualizzate all'occorrenza. La View è costituita da 6 classi che permettono all'utente di interagire e visualizzare le varie informazioni.

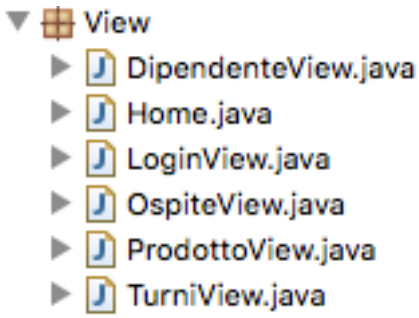

Fig. 6 – Package View

La classe LoginView prevede la creazione dell'interfaccia per permette all'utente di fare l'accesso all'applicazione come amministratore o di accedere come utente ospite.

La classe OspiteView genera l'interfaccia che permette all'utente ospite di cercare i prodotti presenti nel supermercato e di visionare le caratteristiche e in particolare la scaffalatura e i prodotti in offerta.

La classe ProdottoView mi genera l'interfaccia iniziale una volta effettuato l'accesso come amministratore. Questa interfaccia ci permetterà di cercare i prodotti in magazzino, inserirli, modificarli o eliminarli attraverso i rispettivi pannelli.

La classe TurniView genera l'interfaccia con la quale l'amministratore del supermercato potrà inserire il turno lavorativo personalizzandolo per le proprie esigenze; modificare i turni già creati ed eliminarli. Inoltre sarà possibile visualizzare per ogni turno, quali dipendenti lo svolgono.

La classe DipendeteView possiede l'interfaccia dove sarà possibile oltre che a visualizzare i dipendenti per nome o per mansione, ci permetterà anche di inserire i dipendenti associandogli un turno, modificare i dati del dipendente o il turno lavorativo ed eliminare il dipendente.

La classe Home è una classe base, dove sono definiti attributi e metodi e in particolare in essa è implementata l'interfaccia base con componenti come: la label di "Benvenuto", i pulsanti Prodotto, Dipendenti, Turni e il pulsante Logout. Le rispettive View (ProdottoView, DipendentiView e TurniView) estendono questa classe base, ereditano oltre agli attributi e metodi anche l'interfaccia grafica.

6.3.1 Utilizzo di Swing

Componenti

Le classi usate per la gestione della GUI sono Swing e Event. Della seconda diremo soltanto che in ogni classe della View vi è un metodo creato per aggiungere ai componenti grafici degli ascoltatori.
La classe Swing contiene tutti i componenti necessari per la costruzione di una finestra (o un'interfaccia più in generale). Andiamo ad esaminarne alcuni usando come esempio la classe ProdottoView.(Fig. 7)

```
public class ProdottoView extends Home{

    public JTabbedPane tabbedPaneProdotto;

    private JPanel panel;
    private JPanel panel_1;
    private JPanel panel_2;

    public JTable table_1;
    public JTable table;

    private JTextField textNomeProdotto;
    private JTextField textNome;
    private JTextField textMarca;
    private JTextField textPrezzo;
    private JTextField textPrezzoOfferta;
    private JTextField textQta;
    private JTextField textScadenza;
    private JTextField textScaffale;
    private JTextField textInizioOfferta;
    private JTextField textFineOfferta;
    private JTextField textNome1;
```

Fig. 7 – ProdottoView

La classe JPanel del package Swing permette di creare per appunto un pannello e viene poi impostata come "contenuta" nel tabbedPaneProdotto e rende possibile l'aggiunta di altri componenti al suo interno.
Infatti il nostro tabbedPaneProdotto è costituito da 3 JPanel: panel, panel_1, panel_2 rispettivamente Cerca Prodotto, Inserisci Prodotto e Modifica Prodotto. Ognuno di questi panel contiene al suo interno i componenti necessari. In Fig. 7 possiamo notare oggetti che permettono l'interazione con l'utente come le tabelle o campi di testo. La classe JTable ci permette di aggiungere al panel una tabella. La classe JTextField ci permette invece di inserire per esempio nel panel_1, specifico per l'inserimento di un prodotto, i campi di testo necessari.

Passiamo ora ai pulsanti. La classe JButton ci permette di aggiungere al panel dei "bottoni" ai quali verrà associata un'azione da compiere nel caso in cui l'utente li prema. JCheckBox permettere di aggiungere una casella di spunta, mentre JComboBox è la classe relativa al menù a tendina. Infine JLabel corrisponde alla classe che permette di aggiungere una semplice scritta. (Fig. 8).

```
private JCheckBox checkBoxOfferta;
private JCheckBox checkBoxOfferta1;

private JComboBox<String> comboBoxSettore;
private JComboBox<String> comboBoxReparto;
private JComboBox<String> comboBoxSettore1;
private JComboBox<String> comboBoxReparto1;
private JComboBox<String> comboBox;

private JButton btnCerca;
private JButton btnCercaOfferte;
private JButton btnNomeProdotto;
private JButton btnInserisci;
private JButton btnAzzera;

private JButton btnModifica;
private JButton btnElimina;
private JButton btnLogout;
private JButton btnCarica;
private JButton btnCarica2;
private JButton btnImmagine;
private JButton btnImmagine1;
private JButton btnDipendenti;
private JButton btnTurni;

public JLabel lblImmagine;
public JLabel labelLogin;
```

Fig. 8 – ProdottoView

Inizializzazione dei componenti

Per quanto riguarda l'inizializzazione dei componenti dell'interfaccia grafica andremo a visualizzare i metodi di alcuni componenti di esempio della classe ProdottoView.

```
textNomeProdotto = new JTextField();
textNomeProdotto.setBounds(170, 33, 147, 26);
textNomeProdotto.setColumns(10);
panel.add(textNomeProdotto);


comboBox = new JComboBox<String>();
comboBox.setBounds(635, 32, 141, 27);
comboBox.addItem("");
comboBox.addItem("Magazzino");
comboBox.addItem("Utility");
comboBox.addItem("Alimentare");
panel.add(comboBox);


JLabel lblReparto = new JLabel("Scegli reparto");
lblReparto.setBounds(541, 36, 92, 16);
panel.add(lblReparto);

btnCerca = new JButton("Cerca per reparto");
btnCerca.setBounds(771, 30, 161, 29);
panel.add(btnCerca);
```

Fig. 9 – Inizializzazione dei componenti

Come mostrato in Fig. 9 il metodo setBounds (int x, int y, int width, int heigth) permette di settare la posizione e la grandezza dell'oggetto. Il metodo addItem per la comboBox permette di inserire un elemento nel menù a tendina. Il metodo add attribuito al panel permette di aggiungere l'oggetto passato come parametro al panel corrispondente.

Metodi presenti

All'interno delle classi appartenenti al package View sono presenti dei metodi che vengono principalmente richiamati dalle classi corrispondenti nei rispettivi controller.

```
public void refreshTable(String codice){

        try {
            String sql ="select * from prodotto where reparto='Magazzino' "
                    + "and qta < 5 and id='"+codice+"'ORDER BY qta ASC";
            PreparedStatement pst=connection.prepareStatement(sql);
            ResultSet rs=pst.executeQuery(sql);
            table.setModel(DbUtils.resultSetToTableModel(rs));

            rs.close();
        } catch (SQLException e) {

            e.printStackTrace();
        }

    }
```

Fig. 10 – Metodo “mostraprodotti”

Il metodo refreshTable, presente nella classe ProdottoView, permette tramite connessione al db di eseguire una query che seleziona tutti i prodotti del reparto Magazzino e con quantità minore di cinque, ordinandoli per quantità ascendente. Il risultato della query viene poi visualizzato nella tabella tabel.

```
    public void fillComboBox(){
        try {
             String sql="select * from turno";
             PreparedStatement pst=connection.prepareStatement(sql);
             ResultSet rs=pst.executeQuery();
             while(rs.next())
             {
                TurnoBox.addItem(rs.getString("nome_turno"));
             };

        } catch (SQLException e) {

            e.printStackTrace();
        }

    }
```

Fig. 11 – Metodo “fillComboBox”

Il metodo fillComboBox, presente sulla classe DipendeteView, ci permette tramite preventiva connessione al db, e tramite una query di selezione tutti i campi dalla tabella turno; andando adesso a scorrere il risultato della query con il metodo next, eseguiamo il metodo addItem all'oggetto TurnoBox, aggiungendo cosi elementi al menù a tendina. Gli elementi, di volta in volta, saranno costituiti dalla stringa contenuta nel db nel campo nome_turno.

Vi sono poi i metodi per l'aggiunta degli ascoltatori ai componenti di swing; questi ultimi prevedono come parametro un oggetto della classe ActionListener e lo aggiungono al pulsante in questione.

```
public void addModificaListener(ActionListener log) {
      btnModifica.addActionListener(log);
     }
public void addEliminaListener(ActionListener log) {
      btnElimina.addActionListener(log);
     }
public void addLogoutListener(ActionListener log) {
      btnLogout.addActionListener(log);
     }
public void addCaricaListener(ActionListener log) {
      btnCarica.addActionListener(log);
     }
```

Fig. 12 – Listener

6.4 Model

Il Modello è la componente che si preoccupa della gestione e dell'organizzazione dei dati utili. Le classi definite nel package Model sono rispettivamente 5 come mostrato nella Fig. 13 sottostante.

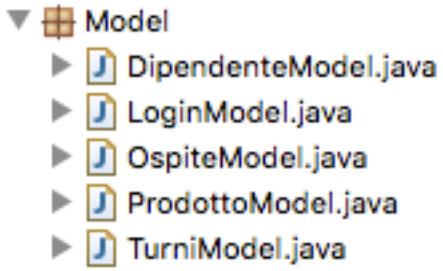

Fig. 13 – Model

Attributi e costruttori

Si analizzerà nel seguito la classe TurniModel.java a titolo esemplificativo: le altre implementazioni risultano analoghe.
Passiamo alla descrizione e analisi degli attributi e costruttori. (Fig. 14)

```
public class TurniModel {

    private String Nome_Turno;
    private String Lunedi;
    private String Martedi;
    private String Mercoledi;
    private String Giovedi;
    private String Venerdi;
    private String Sabato;

    public TurniModel (String Nome_Turno, String Lunedi, String Martedi,
            String Mercoledi, String Giovedi, String Venerdi, String Sabato) {
        this.Nome_Turno=Nome_Turno;
        this.Lunedi=Lunedi;
        this.Martedi=Martedi;
        this.Mercoledi=Mercoledi;
        this.Giovedi=Giovedi;
        this.Venerdi=Venerdi;
        this.Sabato=Sabato;
    }
```

Fig. 14 – TurniModel

Gli attributi mostrati sono dichiarati private String. Per quanto riguarda la costruzione di nuovi oggetti TurniModel, sarà necessario passare al costruttore 7 stringhe i quali valori diventeranno gli attributi del nuovo oggetto creato. Inoltre è presente il costruttore vuoto.

Metodi presenti

I metodi get hanno il compito di restituire i valori degli attributi privati presenti all'interno della classe; per ogni attributo è definito un relativo metodo get. I metodi set ci permettono di andare a modificare gli attributi della classe; per ogni attributo è definito un relativo metodo

set. In seguito verranno elencati i metodi get della classe TurniModel.java.

```
public String getNomeTurno () {
    return this.Nome_Turno;
}
public String getLunedi () {
    return this.Lunedi;
}
public String getMartedi () {
    return this.Martedi;
}
public String getMercoledi () {
    return this.Mercoledi;
}
public String getGiovedi () {
    return this.Giovedi;
}
public String getVenerdi () {
    return this.Venerdi;
}
public String getSabato () {
    return this.Sabato;
}
```

Fig. 15 - Metodi get

Il metodo Calcola_ore() è un metodo pubblico che restituisce un intero, ovvero il numero totale di ore settimanali da cui è composto quel rispettivo turno.

```
public int Calcola_ore () {
    int x = 0;
    if (this.Lunedi == "Mattina" || this.Lunedi == "Pomeriggio")
        x = x + 5;
    if (this.Martedi == "Mattina" || this.Martedi == "Pomeriggio")
        x = x + 5;
    if (this.Mercoledi == "Mattina" || this.Mercoledi == "Pomeriggio")
        x = x + 5;
    if (this.Giovedi == "Mattina" || this.Giovedi == "Pomeriggio")
        x = x + 5;
    if (this.Venerdi == "Mattina" || this.Venerdi == "Pomeriggio")
        x = x + 5;
    if (this.Sabato == "Mattina" || this.Sabato == "Pomeriggio")
        x = x + 5;
    return x;
}
```

Fig. 16 – Calcola_ore

Ogni turno che sia mattina o pomeriggio è composto da 5 ore. Si procede con una serie di if che vanno a verificare per ogni giorno della settimana quale turno è stato scelto. Se corrisponde a Mattina o Pomeriggio, il

numero di ore viene incrementato di 5, altrimenti si procede al successivo giorno della settimana. Infine sarà possibile quindi sapere il numero ore settimanali di cui sarà composto quel turno.

6.5 Controller

Il Controller è l'elemento che permette di unire View e Model trasformando gli input e l'interazione dell'utente con la GUI in operazioni da eseguire in quest'ultimo.
Le classi definite nel package Controller permettono di gestire i vari Eventi che si verificano nelle schermate della View, associando ad ogni azione un'operazione.
Un clic su un bottone o in una casella di testo viene gestito interamente dal controller; questo, in base all'opzione selezionata, si preoccupa di creare determinati oggetti e richiamare opportuni metodi che permettono di aggiornare il Model o visualizzare altre tipologie di schermate.

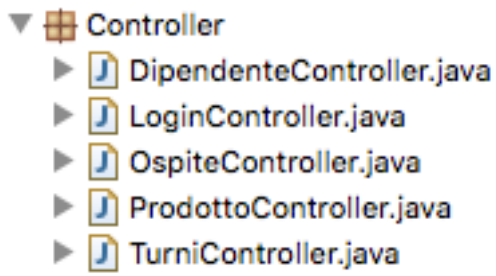

Fig. 17 – Controller

6.5.1 Eventi e Listener

Il controller è uno degli elementi essenziali dell'architettura progettuale MVC e viene definito come il componente che fa da tramite tra View e Model. La responsabilità fondamentale di questo organo è la gestione degli input e delle varie azioni che l'utente compie con l'interfaccia grafica, producendo un'adeguata risposta.

Il controller si preoccupa della gestione delle operazioni da eseguire in base al tipo di interazione da parte dell'utente con la GUI. Nel momento in cui l'utente conferma un determinato form, o aggiunge una particolare Attività è il controller che fa in modo di visualizzare delle "finestre aggiuntive" o di modificare i dati utili interfacciandosi con il Model. Una delle caratteristiche fondamentali e comuni di tutte le classi appartenenti alla sezione Controller è la gestione degli Eventi tramite particolari elementi Java, i Listener.
Con il termine Evento si indica un oggetto indicante un'azione, come la pressione di un bottone, la chiusura di una finestra, o qualunque altra cosa che ci si aspetta produca risposta. Un evento generato da un oggetto viene detto "scatenato" o "emesso". Un oggetto che è in grado di generare un evento deve avere uno o più oggetti ascoltatori detti Listeners. Nel momento in cui un oggetto genera un evento, quest'ultimo viene immediatamente inviato all'oggetto ascoltatore che, attraverso opportuni metodi detti "gestori di eventi", produce una risposta adeguata. Questo tipo di programmazione viene definita "ad eventi" e nella maggior parte dei casi l'ordine con cui saranno eseguite le varie azioni non è mai noto a priori ma dipende dall'utente.

Attributi e costruttori: ProdottoController

In seguito sarà analizzata una determinata classe del package controller (ProdottoController), questo poiché tutte le classi inerenti alla sezione Controller sono simili, ciò che cambia solo le operazioni che vengono eseguite nel momento in cui si verifica un evento. Per esempio nel momento in cui viene premuto il bottone "btnInserisciProdotto" viene memorizzato nel database un nuovo prodotto, mentre se viene premuto il bottone "btnModificaProdotto" va ad aggiornare il prodotto nel database con i campi modificati nella form apposita. L'evento gestito è lo stesso (bottone premuto) ma le operazioni che vengono eseguite sono diverse.
Tutte le classi del package Controller importano determinate librerie di Java che permettono l'uso di particolari strumenti come eventi, ascoltatori, action, componenti di Swing… inoltre è presente

l'importazione di package inerenti al progetto che consentono di utilizzare oggetti del Model e della View (Fig. 18).

```
package Controller;

import java.awt.Font;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.sql.Connection;
import java.sql.Date;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import javax.swing.JFileChooser;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import com.connectiondb.*;
import Model.DipendenteModel;
import Model.LoginModel;
import View.DipendenteView;
import View.LoginView;
import Model.ProdottoModel;
import Model.TurniModel;
import View.ProdottoView;
import View.TurniView;
import net.proteanit.sql.DbUtils;
import Controller.LoginController;
```

Fig. 18 – Importazione librerie

Passiamo alla descrizione degli attributi e del metodo costruttore.

```
public class ProdottoController extends Home{

    private ProdottoModel model;
    private ProdottoView view;

    public ProdottoController(ProdottoView view){

        this.view = view;

        view.addNomeProdottoListener(new CercaProdottoNome());
        view.addCercaRepartoListener(new CercaProdottoReparto());
        view.addCercaOffertaListener(new CercaOfferte());
        view.addInserisciProdottoListener(new InserisciProdotto());
        view.addAzzeraProdottoListener(new AzzeraProdotto());
        view.addModificaListener(new ModificaProdotto());
        view.addEliminaListener(new EliminaProdotto());
        view.addCaricaListener(new CaricaProdotti());
        view.addCarica2Listener(new CaricaProdotti2());
        view.addImmagineListener(new ScegliImmagine());
        view.addImmagine1Listener(new ScegliImmagine());
        view.Logout(new Logout());
        view.dipendenteView(new dipendenteView());
        view.turniView(new turniView());
        view.prodottiView(new prodottiView());


    }
```

Fig. 19 – Attributi e metodi costruttore

Un aspetto comune a tutte le classi del package controller sono alcuni attributi e metodi della classe Home e quindi come già accennato prima anche i controller estendono la classe Home.
Poiché l'oggetto di tipo ProdottoController gestisce gli eventi del corrispettivo oggetto del package View, viene definito un attributo di tipo ProdottoView settato nel costruttore passato come argomento.

Action-ProdottoController

Il metodo addNomeProdottoListener(ActionListener) viene richiamato tramite l'oggetto view di tipo ProdottoView passandogli come paramento un'istanza della classe CercaProdottoNome. Questa classe, permette di associare un ascoltatore ad un particolare componente dell'oggetto view di tipo ProdottoView. In generale abbiamo un componente (un bottone in questo caso) del form di tipo ProdottoView a cui viene assegnato un ascoltatore, per questo motivo andiamo ad analizzare la classe interna CercaProdottoNome, la quale gestisce gli eventi che possono verificarsi (Fig. 20).

Tutte le classi del package Controller permetto di gestire ulteriori eventi riferiti a particolari finestre dell'interfaccia grafica. Abbiamo deciso di descrivere la classe ProdottoController perché si tratta di una delle classi più complete e rappresentative riguardo il concetto di Controllo del paradigma MVC. Verranno esaminate solo alcune, data la forte somiglianza delle stesse.
La classe CercaProdottoNome implementa l'interfaccia ActionListener. Ciò significa che la classe CercaProdottoNome deve definire il metodo actionPerformed(ActionEvent). Questo è il gestore dell'evento "pressione del bottone" e definisce le azioni da eseguire nel momento in cui l'evento si verifica. In questo caso possiamo notare dall'implementazione del metodo che l'azione da eseguire al momento in cui si preme il rispettivo bottone è quella di recuperare il nome del prodotto dal model (get), successivamente eseguire la query che seleziona tutti i prodotti aventi quel nome (appartenenti all'utente

loggato, andando a fare un controllo sull'id) ordinando il risultato per prezzo.
Il risultato della query viene visualizzato sull'oggetto table della view corrispondente.

```
class CercaProdottoNome implements ActionListener {
        public void actionPerformed(ActionEvent e) {

            String nome;

                model=view.getProdotto();
                nome=model.getNome();
                String sql ="select * from prodotto where nome='"+nome+"'and id='"+id+"'order by prezzo";
                try {
                    Statement st=connection.createStatement();
                    rs=st.executeQuery(sql);
                } catch (SQLException e2) {

                    e2.printStackTrace();
                }

                view.table.setModel(DbUtils.resultSetToTableModel(rs));

    }}
```

Fig. 20 – CercaProdottoNome

Passiamo adesso alla descrizione di una classe che consente di gestire l'evento "pressione di un bottone". Il metodo implementato permette di eliminare il prodotto selezionato sulla table, dal database.

```
class EliminaProdotto implements ActionListener {
    public void actionPerformed(ActionEvent e){
        int action=JOptionPane.showConfirmDialog
                (null, "Sei sicuro di voler eliminare il prodotto selezionato?","Elimina", JOptionPane.YES_NO_OPTION);
        if (action==0){
        try{

            int row=view.table_1.getSelectedRow();
            String codice_=(view.table_1.getModel().getValueAt(row, 0)).toString();

            String query="Delete from prodotto where codice='"+codice_+"'";

            PreparedStatement pst=connection.prepareStatement(query);
            pst.execute();

            JOptionPane.showMessageDialog(null, "Prodotto eliminato correttamente!");
            pst.close();
            view.refreshTable1(id);
            view.refreshForm();
                }catch(Exception e1){

                    e1.printStackTrace();
                }
    }
}}
```

Fig. 21 – EliminaProdotto

La classe EliminaProdotto, come nella descrizione della classe precedente, anche questa implementa l'ActionListener. Questa volta però il metodo "gestore dell'evento" compie un'azione differente. Inizialmente verifico se la risposta alla domanda "Sei sicuro di di voler eliminare il prodotto selezionato?" sia positiva. Se positiva il valore action sarà uguale a 0. Verifico con un if che action sia zero e procedo. Nella variabile int row viene salvato il numero della riga selezionata sull'oggetto table_1. Poi nella variabile String codice_ viene salvata la stringa contenuta nella riga selezionata di colonna zero, ovvero contenente il codice identificativo del prodotto. A questo punto viene eseguita una query per l'eliminazione del prodotto, dal database nella tabella prodotto, avente quel codice identificativo sopra ricavato.
Se l'operazione è andata a buon fine verrà visualizzato il messaggio Prodotto eliminato correttamente, e si procederà a riaggiornare la tabella table_1 con il metodo visto nel capitolo 6.3.1.

7. STRUMENTI DI PROGRAMMAZIONE UTILIZZATI

Verranno analizzati nel seguito alcuni costrutti di programmazione utilizzati nel nostro software.

7.1 Incapsulamento

L'incapsulamento è proprio legato al concetto di "impacchettare" in un oggetto i dati e le azioni che sono riconducibili ad un singolo componente. Un altro modo di guardare all'incapsulamento, che abbiamo già accennato, è quello di pensare a suddividere un'applicazione in piccole parti (gli oggetti, appunto) che raggruppano al loro interno alcune funzionalità legate tra loro. Nella pratica come prima cosa bisogna rendere gli attributi della classe in maniera tale che non siano direttamente accessibili dall'esterno. Ciò si ottiene dichiarando gli attributi della classe usando il modificatore di visibilità private. Usando private si rendono gli attributi delle classi visibili solo ed esclusivamente all'interno della classe, dunque potranno manipolarli solo i metodi facente parte della stessa classe.
Nel nostro software, l'incapsulamento è utilizzato in qualsiasi classe del package Model, ad esempio nella classe "DipendenteModel" è stato applicato l'incapsulamento infatti non è possibile accedere direttamente agli attributi definiti al suo interno ma bisogna adoperare i relativi metodi Get e Set. (Fig. 22).

```
public class DipendenteModel {
    private  String NomeDipendente;
    private  String CognomeDipendente;
    private  String CodiceFisc;
    private  String Indirizzo;
    private  String Email;
    private  Date AnnoNascita;
    private  String Mansione;
    private  Date Data_Ass;
    private  Date Data_Lic;
    private  Integer id;
    private  String Telefono;
    private  String Nome_Turno;

    public String getNomeDipendente() {
        return NomeDipendente;
    }

    public void setNomeDipendente(String NomeDipendente) {
        this.NomeDipendente = NomeDipendente;
    }

    public String getCognomeDipendente() {
        return CognomeDipendente;
    }

    public void setCognomeDipendente(String CognomeDipendente) {
        this.CognomeDipendente = CognomeDipendente;
    }
```

Fig. 22 – Dipendente Model

7.2 Ereditarietà

L'ereditarietà costituisce il secondo principio fondamentale della programmazione ad oggetti. In generale, essa rappresenta un meccanismo che consente di creare nuovi oggetti che siano basati su altri già definiti. Si definisce oggetto figlio (child object) quello che eredita tutte o parte delle proprietà e dei metodi definiti nell'oggetto padre (parent object).

Uno dei maggiori vantaggi derivanti dall'uso dell'ereditarietà è la maggiore facilità nella manutenzione del software. Il termine ereditarietà indica il meccanismo consistente nella definizione di una classe più generale che sarà il modello di classi più specializzate a cui vengono aggiunti nuovi dettagli rispetto a quella di partenza. La classe definita mediante un'estensione è chiamata classe derivata, estesa o sottoclasse, mentre la classe originaria, da cui le altre discendono, è detta superclasse o classe base. Nella definizione della sottoclasse sono presenti implicitamente tutti i metodi e gli attributi della classe base ed è possibile definire ed implementare attributi e metodi aggiuntivi. Tutti i metodi della classe base con l'indicatore di visibilità protected o public sono acquisiti implicitamente dalla classe derivata e possono essere ridefiniti (override dei metodi).

Nel nostro software l'ereditarietà è stata applicata definendo una classe "Home" come classe base, e per esempio la classe ProddottoView estende la classe "Home" ereditando da essa appunto gli attributi e i metodi, come riportato in figura 23.

```
public class Home implements ActionListener{

    public JFrame frame;
    public Connection connection=sqlConnection.dbConnector();
    public JLabel labelLogin;
    public JButton btnTurni;
    public JButton btnDipendenti;
    public JButton btnLogout;
    public JButton btnProdotto;
    public String id;
    public String label;

    public Home() {
        frame = new JFrame();
        frame.setBounds(100, 100, 1280, 800);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().setLayout(null);

        labelLogin = new JLabel();
        labelLogin.setBounds(143, 7, 61, 16);
        labelLogin.setFont(new Font("Lucida Grande", Font.BOLD, 16));
```

Fig.23A - Classe Home

```
public class ProdottoView extends Home{

    private JPanel panel;
    private JPanel panel_1;
    private JPanel panel_2;

    public JTable table_1;
    public JTable table;

    private JTextField textNomeProdotto;
    private JTextField textNome;
    private JTextField textMarca;
    private JTextField textPrezzo;
    private JTextField textPrezzoOfferta;
    private JTextField textQta;
```

Fig.23B - Classe ProdottoView

7.3 Interfaccia

In generale, un'interfaccia rappresenta una sorta di "promessa" che una classe si impegna a mantenere. La promessa è quella di implementare determinati metodi di cui viene resa nota soltanto la definizione.
Si può dichiarare che una classe implementa (implements) una data interfaccia: in questo caso deve realizzare tutti i suoi metodi astratti, fornendo dei metodi con la stessa intestazione e corpo. La realizzazione di un metodo deve rispettare la specifica del corrispondente metodo astratto.

Ciò che è importante non è tanto come verranno implementati tali metodi all'interno della classe ma, piuttosto, che la denominazione ed i parametri richiesti siano assolutamente rispettati. Infine, una classe può implementare più di una interfaccia. Ovvero, è possibile obbligare una classe ad implementare tutti i metodi definiti nelle interfacce con le quali essa è legata. Questa ultima caratteristica fornisce, indiscutibilmente, la massima flessibilità nella definizione del comportamento che si desidera attribuire ad una classe.
Nel programma sono state utilizzate per la gestione degli eventi come è stato spiegato nel paragrafo 6.5.1.
Tutte le classi del package controller che rappresentano "gestori di eventi" implementano l'interfaccia ActionListener, quest'ultima definisce il metodo: "public void actionPerformed(ActionEvent e)".
Tutte le classi che implementano l'interfaccia in esame devono dare corpo a questo metodo.
Vediamo un esempio riportato nella classe ProdottoController nella figura 24.

```
class CercaProdottoReparto implements ActionListener {
    public void actionPerformed(ActionEvent e) {

            String reparto;
            String sql;
            model=view.getProdotto();
            reparto=model.getReparto();

            try {
                if(reparto.equals("")==true){
                    sql=null;
                }else{
                sql ="select * from prodotto where reparto='"+reparto+"'and id='"+id+"'ORDER BY qta ASC";
                }
                Statement st=connection.createStatement();
                ResultSet rs=st.executeQuery(sql);
                view.table.setModel(DbUtils.resultSetToTableModel(rs));
                st.close();
                rs.close();


            } catch (SQLException e2) {
                JOptionPane.showMessageDialog(null, "Impossibile cercare il prodotto: inserire il reparto!");

            }


}}
```

Fig.24 - CercaProdottoReparto

Questa classe implementa appunto l'interfaccia, la quale definisce il metodo actionPerformed e al suo interno c'è il corpo del metodo, attraverso il quale tramite una query sql vengono recuperati tutti i prodotti con reparto uguale a quello indicato e il risultato viene visualizzato nella table.

7.4 Eccezioni

Una eccezione, in un programma, è una sorta di evento che non era stato previsto che genera un errore e anche la terminazione prematura dell'applicazione stessa. Per gestire l'eccezione bisogna inserire il codice che potrebbe generarla nel blocco **try** e nel blocco **catch** il codice di gestione dell'eccezione (come per esempio una stampa a video che notifichi all'utente che si è verificato un errore).

Nel nostro programma sono presenti diversi blocchi try/catch, in figura 25 viene riportato un esempio presente nella classe ProdottoController.

```
class CercaProdottoNome implements ActionListener {
    public void actionPerformed(ActionEvent e) {

            String nome;
            String sql;
            model=view.getProdotto();
            nome=model.getNome();

            try {
                if (nome.equals("")==true){
                    sql=null;
                }else{
                    sql ="select * from prodotto where nome='"+nome+"'and id='"+id+"'order by prezzo";
                }

                PreparedStatement pst=connection.prepareStatement(sql);
                ResultSet rs=pst.executeQuery(sql);
                view.table.setModel(DbUtils.resultSetToTableModel(rs));
                pst.close();
                rs.close();
            } catch (SQLException e2) {
                JOptionPane.showMessageDialog(null, "Impossibile cercare il prodotto: inserire il nome!");

            }


}}
```

Fig. 25 – CercaProdottoNome (Eccezione)

Durante l'esecuzione del codice nel blocco try (sempre eseguito), nel caso in cui la variabile String sql è uguale a null, l'istruzione PreparedStatement pst=connection.prepareStatement(sql) mi genera il seguente errore: `java.sql.SQLException: SQL String can not be NULL`
Nel blocco catch, viene gestita questa eccezione visualizzando uno showMessageDialog con la scritta "Impossibile cercare il prodotto: inserire il nome!", evitando l'interruzione del programma.

7.5 Notazioni finali

Il programma è stato sviluppato con l'IDE Eclipse Neon, invece per la realizzazione del database MySql è stato utilizzato phpMyAdmin, tramite il software MAMP.

In allegato sarà fornita l'immagine chiara del diagramma delle classi.