Agent Manager

Ingegneria del software

Girolamo Macaluso, Mirko Bicchierai

UNIVERSITÀ
DEGLI STUDI
FIRENZE
Scuola di
Ingegneria

Computer Engineering
University Of Florence
Italy
19/03/2021

Agent Manager

Girolamo Macaluso, Mirko Bicchierai

March 2021

Contents

1 Abstract

Il progetto iniziale prevedeva di sviluppare un applicazione mobile in grado di gestire ordini di articoli, permettendone la lettura tramite barcode. Questi ordini sarebbero dovuti essere inseriti dagli agenti commerciali. Era anche prevista una gestione dei clienti e degli agenti operanti con il programma. Il cambiamento principale è stato quello di sviluppare il progetto come applicazione desktop; ciò ha fatto perdere senso alla lettura dei barcode che è stata quindi scartata dal progetto finale, essa però è supportata grazie ai driver di windows che permettono l'input da tastiera.

La parte degli articoli è stata sviluppata ulteriormente dando la possibilità di inserirne di composti da altri prodotti, senza limiti di livello, implementando un calcolo automatico del costo totale. E' stata anche aggiunta una gestione degli utenti più profonda, vi sono infatti due tipi di utenti, amministratori e agenti; i primi possono compiere operazioni di gestione su agenti, articoli, cataloghi e clienti; gli altri possono gestire i propri ordini, creare i clienti ed hanno associato un catalogo di vendita che contiene i prodotti che sono autorizzati a vendere.

Un nuovo ordine inserito da un agente genera due tipi di notifica, una all'interno del programma che al momento dell'accesso dell'amministratore segnalerà gli ordini inseriti; l'altra tramite delle email inviate a tutti gli amministratori e al cliente, per informarlo della presa in gestione del suo ordine.

I dati sugli utenti, clienti, ordini, cataloghi e notifiche sono immagazzinati all'interno di un database sql dal quale sono caricati all'avvio e aggiornati al momento della chiusura del programma.

L'obiettivo iniziale era di realizzare anche un interfaccia grafica, successivamente abbiamo preferito concentrarci sulle funzionalità piuttosto che sull'aspetto grafico; ma è stato comunque deciso di realizzare un interfaccia in forma testuale per rendere il programma pronto all'uso.

E' stata quindi implementata una gestione di menù che consentono agli utenti, tramite input da tastiera, di effettuare tutte le operazioni. L'applicazione prevede anche il login degli utenti, con salvataggio dell'hash della password, e il caricamento dei menù con le funzionalità di competenza del fruitore differenziando tra amministratori e agenti.

2 Diagrammi UML

In questa sezione vengono presentati tre diagrammi UML, il diagramma delle classi, il diagramma dei casi d'uso ed il diagramma E/R della struttura dati utilizzata.

2.1 Class diagram

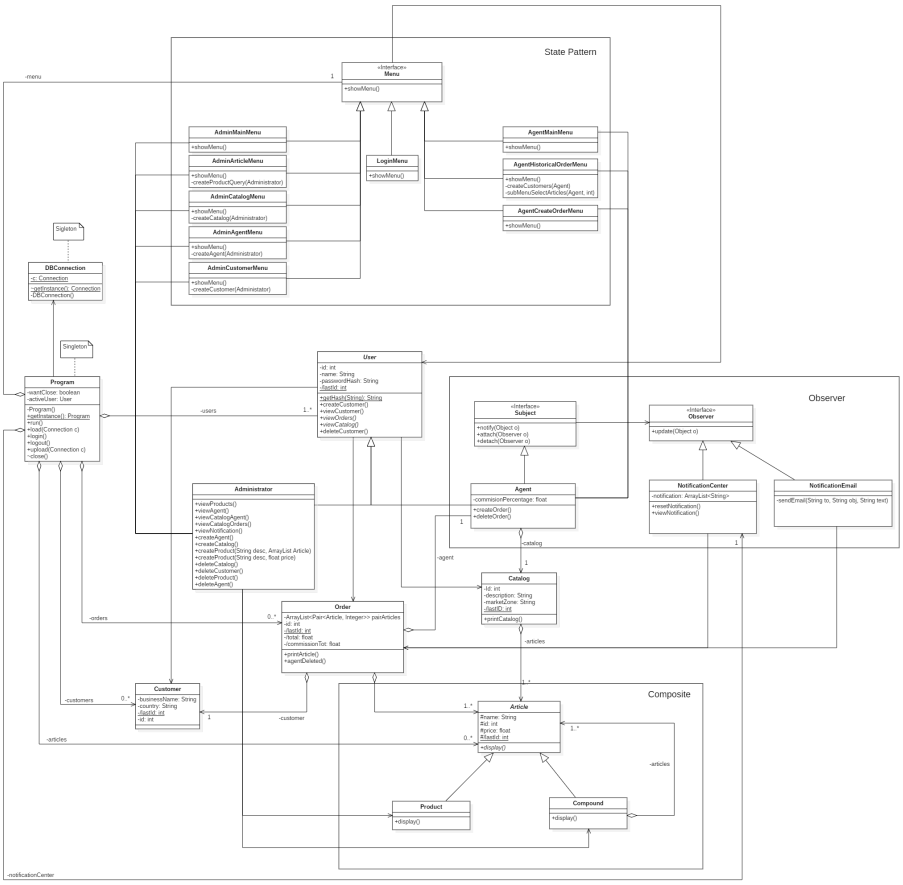

Figure 1: Class Diagram

Volevamo realizzare una piccola interfaccia grafica di supporto al programma tuttavia dopo un ridimensionamento di un progetto siamo passati da un interfaccia grafica completa ad la gestione attraverso dei menu, non escludendo però perdere la possibilità di implementare un interfaccia grafica completa senza dover compromettere la struttura dell'intero programma. L'intento era quello

di fornire un interfaccia di menu facilmente scalabile e che potesse essere trattata dal resto del programma in modo trasparente rispetto all'implementazione considerata. Per realizzare tale sistema si è deciso di utilizzare uno state pattern per la realizzazione vari menu navigabili, in modo da consentire anche una migliore fruizione da parte degli utenti. Nel caso in cui si volesse implementare una piccola interfaccia grafica basterebbe sostituire nei vari *ConcreteState* dei menu la definizione dei vari pannelli dell'interfaccia, esempio: un pannello per il login, uno per gli agenti, un altro per gli amministratori ecc.. . L'UML di tale patter è simile ad quello di uno strategy tuttavia lo state permette di cambiare il suo comportamento al cambiare del suo stato interno, nel caso considerato cambia il suo comportamento in base alle scelte effettuate nei vari menu, mentre lo strategy viene utilizzato per definire un insieme di algoritmi intercambiabili indipendentemente dai client che ne fanno uso.

Il nostro intento era quello di notificare clienti e amministratori della registrazione di un nuovo ordine da parte di un agente, il metodo di notifica però deve essere disaccoppiato dalla sua generazione poiché entrambe possono variare in momenti diversi, violando il principio di *single responsibiliy*. Si è quindi ritenuto opportuno creare una classe responsabile della notifica, questa avrebbe potuto fare *polling* per rendersi conto della registrazione di un nuovo ordine, ma abbiamo deciso di implementare un sistema *event driven*. E' stato quindi utilizzato un observer per la gestione delle notifiche all'interno del programma. E' stato deciso di non utilizzare l'*observer* standard di *java.utils*, poiché la classe Agent ha già esaurito la sua possibilità di *subclassing*, con l'estensione di *User*; sono state quindi realizzate due interfacce *Subject* e *Observer*. L'Observer è stato realizzato in modalità *push* in quanto gli agenti inviano a tutti gli observer, che si sono registrati, l'ordine che è stato emesso. All'interno del nostro programma ci sono due implementazioni di observer, uno per inviare le notifiche agli amministratori all'interno del programma ed uno per l'invio dei avvisi tramite email ad amministratori e clienti.

Gli Amministratori hanno il compito di creare agenti, articoli e cataloghi; il nostro obiettivo era quello di supportare anche articoli composti, cioè assemblati a partire da altri prodotti. Ad esempio l'articolo X può essere creato a partire dai prodotti X e Y i quali possono essere comunque venduti singolarmente. Per modellare tale realtà e trattare tutti gli articoli allo stesso identico modo, si è deciso per l'utilizzo di un composite pattern. Questo porta ad avere per i prodotti una struttura relativamente complessa, a prima vista potrebbe sembrare un albero tuttavia risulta essere un grafo in quanto è possibile creare articoli composti da altri prodotti, che a loro volta sono componenti in altri composti. Tuttavia in tale grafo non possono crearsi cicli in quanto gli articoli non possono essere modificati e alla creazione di un composto è possibile scegliere come composizione di esso solamente i prodotti già esistenti.

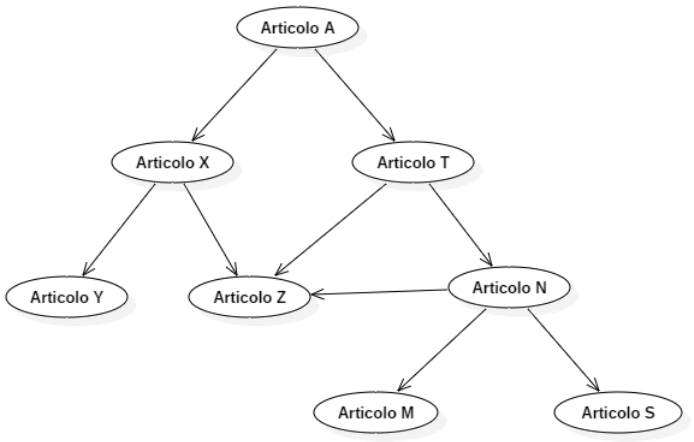

Figure 2: Esempio di una composizione di un articolo

All'interno del diagramma delle classi possiamo inoltre identificare una generalizzazione degli utenti in quanto le classi *Administrator* e *Agent* estendono la classe padre *User*. Entrambe le categorie di fruitori possono accedere al sistema ed hanno dati e funzionalità comuni tra di loro, tuttavia all'interno del programma non verrà mai istanziato un oggetto *user* e quindi risulta essere una classe astratta.

E' stata inoltre utilizzata la classe *Program* come contenitore, rappresenta una facciata per l'utilizzo dell'intero sistema, essa potrebbe somigliare ad un *Facade* tuttavia non lo è in quanto le classi del sottosistema interagiscono con essa. La classe *Program* è inoltre un singleton in quanto si necessita di avere una singola istanza di essa ovunque reperibile. *Program* offre metodi come *load(Connection c)* e *upload(Connection c)* per potersi interfacciare con il DB e *run()* per poter gestire il *loop* di sistema.

2.2 Use case diagram

I possibili attori che interagiscono con la nostra applicazione sono due: Agenti ed amministratori. Ci sono inoltre alcuni casi d'uso comuni per i due attori dell'applicazione che identificano una generalizzazione tra agenti ed amministratori. Di seguito è mostrato il diagramma dei casi d'uso completo successivamente verranno approfonditi i vari casi d'uso dei singoli attori.

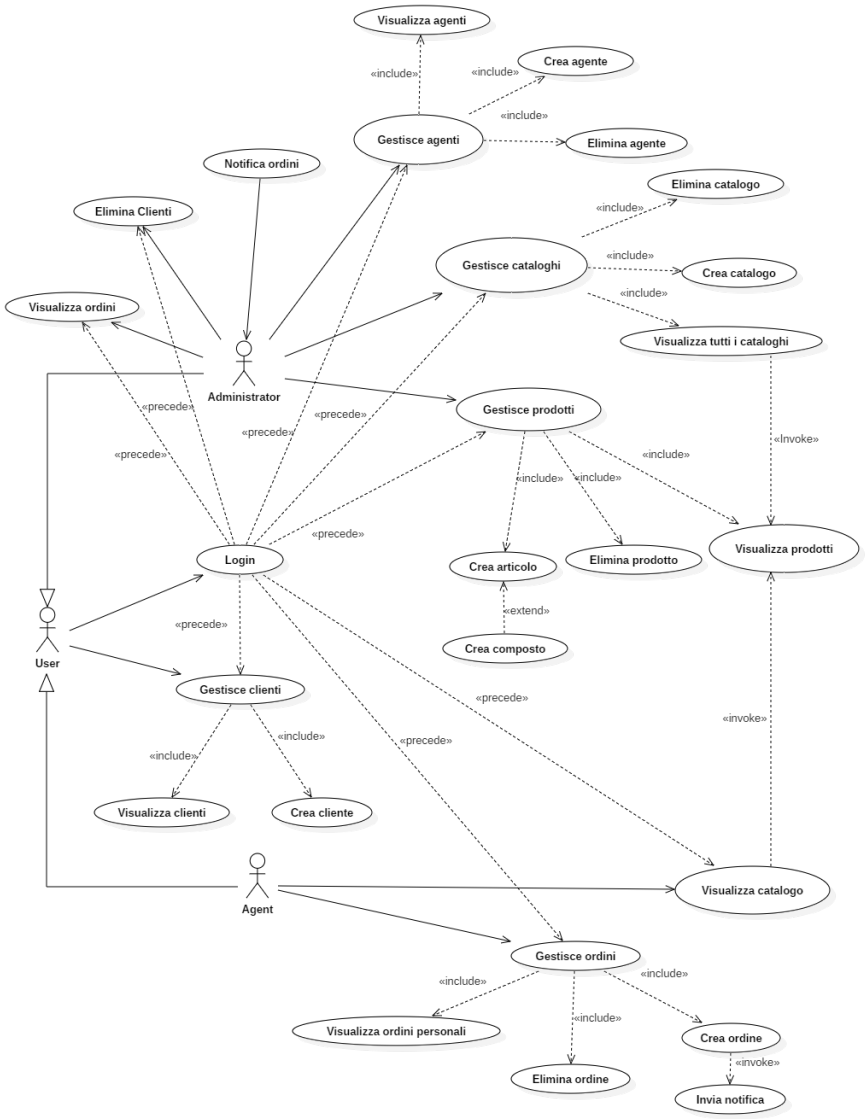

Figure 3: Use Case diagram Completo

2.2.1 User's use case

In questa sezione verranno discussi i casi d'uso comuni sia agli amministratori che agli agenti.

Per poter accedere alla gestione dei clienti si deve aver prima obbligatoriamente effettuato il login e per questo motivo il caso d'uso *login* precede il caso d'uso *gestione clienti*. In generale *login* precede sempre l'accesso a tutti i casi d'uso diretti del sistema.

Il caso d'uso *gestione clienti* include i casi d'uso *visualizza clienti* ed il caso *crea clienti* come sotto casi specifici. Questi casi d'uso sono comuni ad entrambe le categorie di utenti in quanto gli agenti hanno bisogno di poter vedere i clienti per poter erogare un ordine e nell'evenienza devono poter aggiungerne uno nuovo se non ancora registrato; mentre per gli amministratori deve essere possibile avere un completo controllo sui clienti che include sia la creazione che la visualizzazione di essi.

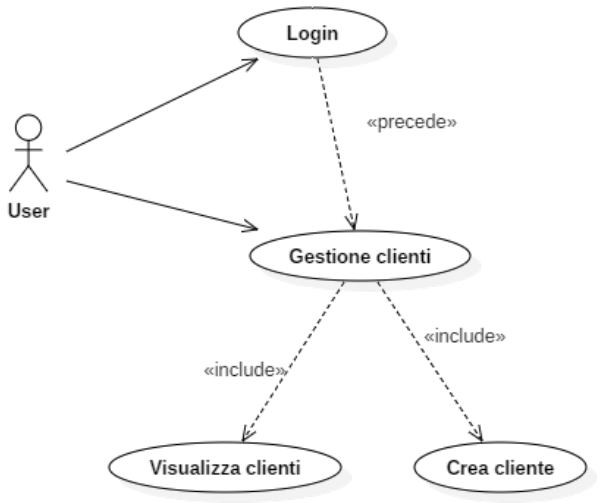

Figure 4: Use Case diagram User

2.2.2 Administrator's Use case

In questa sezione verranno discussi i casi d'uso relativi agli amministratori.

Si identificano tre macro casi d'uso *gestione agenti, gestione cataloghi, gestione prodotti* che a suo volta includono sotto casi d'uso. Il caso d'uso *Visualizza tutti i cataloghi*, incluso nella *gestione cataloghi*, invoca (*invoke*) il caso d'uso visualizza i prodotti, che permette la visualizzazione di tutti i prodotti contenuti nel catalogo richiesto, per ogni catalogo.Il caso d'uso *crea articolo* viene esteso (*extend*) dal caso d'uso *crea composto* in quanto quanto un amministratore può scegliere se creare un articolo semplice o un articolo composto da più prodotti.

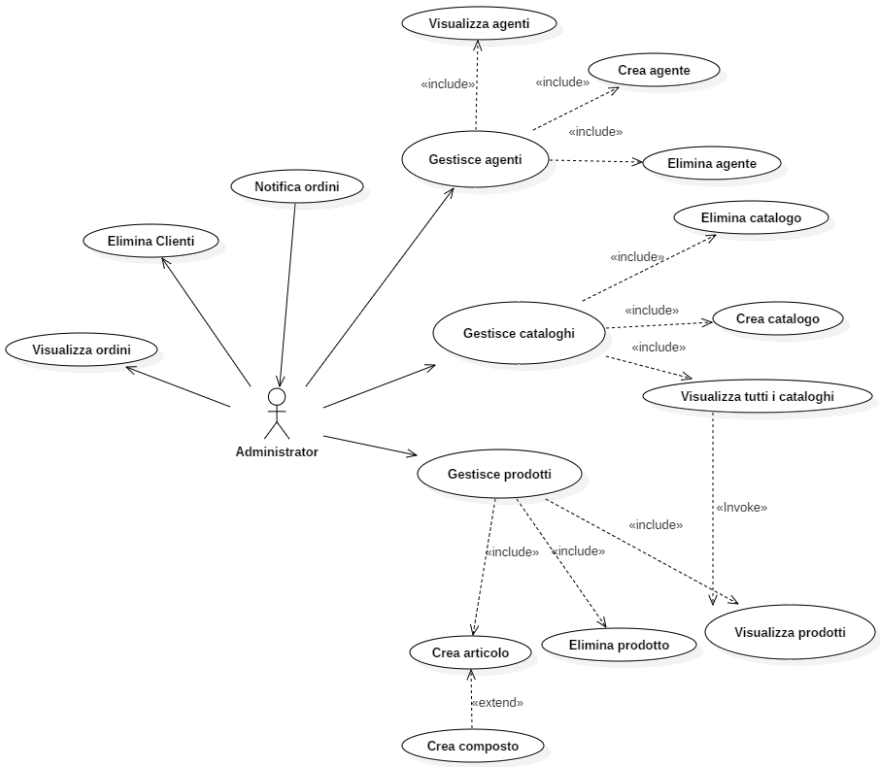

Figure 5: Use Case diagram Amministratore

2.2.3 Agent's use case

In questa sezione verranno discussi i casi d'uso relativi agli agenti.

Si identificano due macro casi d'uso *gestione ordini, visualizza catalogo*.

Il caso d'uso *crea ordine* invoca (*invoke*) il caso d'uso *invia notifica* in quanto al termine dell'esecuzione del caso d'uso *crea ordine* deve esser notificato agli amministratori che è stato erogato un nuovo ordine ed i clienti devono ricevere un email di riepilogo dell'ordine.

Il caso d'uso *Visualizza catalogo*, invoca (*invoke*) il caso d'uso *visualizza i prodotti*, che permette di visualizzare tutti gli articoli presenti all'interno del proprio catalogo di vendita.

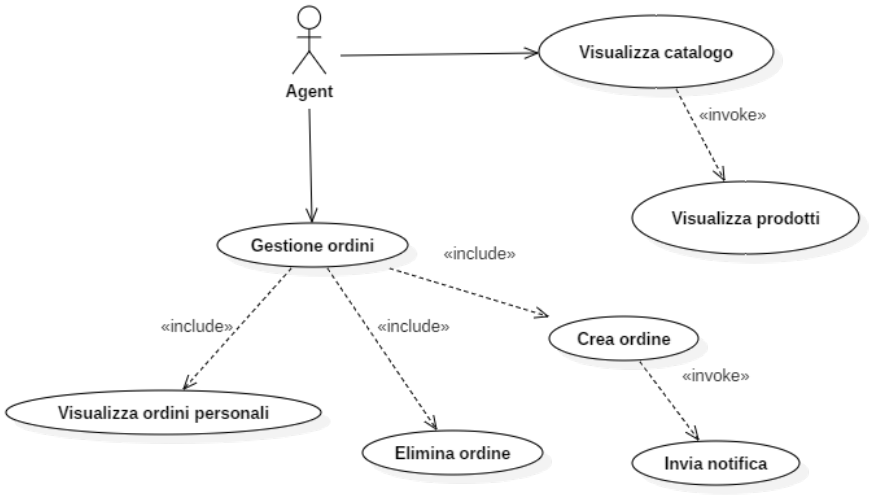

Figure 6: Use Case diagram Agente

2.3 E/R diagram

Di seguito viene riportato il diagramma E/R della base dati utilizzata.

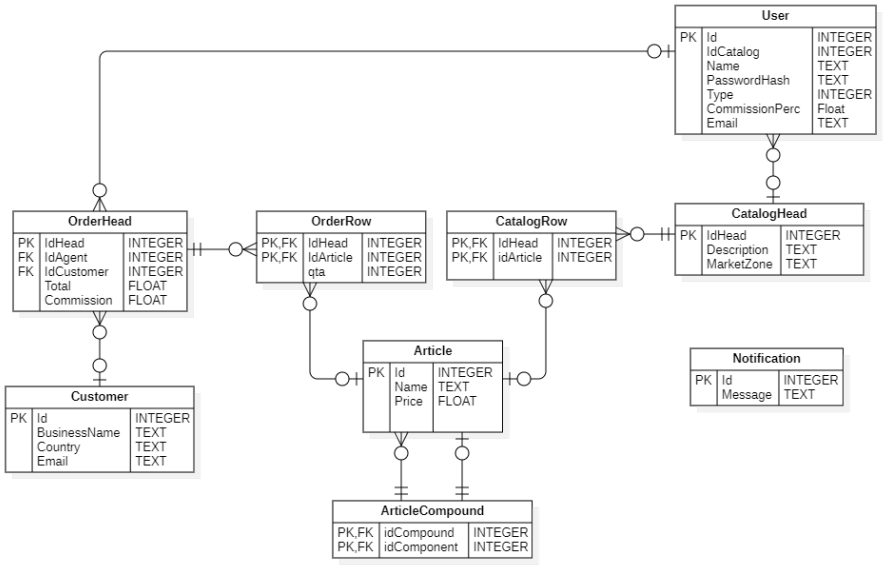

Figure 7: Diagramma ER

3 Implementazione

Di seguito vengono presentate le funzioni del progetto che meritano un approfondimento riguardo le scelte di implementative.

3.1 Articoli

La classe articoli è astratta, essa delinea l'interfaccia delle due tipologie possibili di prodotti, quelli semplici e quelli composti; quest'ultimi sono creati a partire da altri articoli, che a loro volta posso essere composti. Il composite ha lo scopo principale di fornire la stessa interfaccia per entrambi i tipi di articoli, concretamente permette, tramite la funzione display(), nel caso di articoli composti, di stampare anche i dati degli articoli componenti. Un'altra funzionalità della classe Compound è quella del calcolo automatico del prezzo in base a quello degli articoli che lo compongono. Di Seguito è presentata l'implementazione del metodo display nelle due classi.

```
1 public void display() {
2     System.out.println("--Id: "+this.id+" Article: " + name + " Price: " +
          price);
3 }
```

Listing 1: Implementazione di display in Product

```
1 public void display() {
2     System.out.println("--Id: "+this.id+" Compound: " + name + " Price:
          " + price);
3     for(Article i : components)
4         System.out.println("    || Component: " + i.getName() + " Price: " +
              i.getPrice() + (i instanceof Compound ? " (Compound)":""));
5 }
```

Listing 2: Implementazione di display in Compound

3.2 Utenti

La classe utente è astratta, ha lo scopo di fornite un interfaccia comune ai vari attori all'interno del programma, essa offre l'implementazione per alcuni metodi comuni ad entrambe le classi che la specializzano. I metodi *viewOrders()* e *viewCatalog()* sono astratti poiché consentono agli amministratori di visualizzare tutti gli ordini e tutti i cataloghi, invece gli agenti di visualizzeranno solo gli ordini effettuati o il catalogo a loro assegnato. Tra gli attributi dell'utente c'è l'hash della password che viene calcolato tramite il metodo statico *getHash(String p)* che restituisce il risultato dell'esecuzione della funzione hash MD5; in modo da non dover salvare la password in chiaro all'interno del programma e nel database.

3.2.1 Agenti

Agli agenti è consentita gestione dei propri ordini: è data la possibilità di inserirne di nuovi, con il vincolo che gli articoli contenuti all'interno siano presenti nel listino a loro assegnato. Inoltre è anche possibile eliminarli in caso sia stato effettuato un errore. Alla creazione dell'ordine corrisponde una notifica agli observer, che sarà discussa successivamente.

```
public void createOrder(Customer c, ArrayList<Pair<Article,Integer>>
    articles) {
    Order order = new Order(this,articles,c);
    Program.getInstance().getOrders().add(order);
    System.out.println("Created!");
    notify(order);
}
```

Listing 3: Implementazione della creazione dell'ordine

```
@Override
public void notify(Order order) {
    for(Observer o: observers)
        o.update(order);
}
```

Listing 4: Implementazione di Notify

Un'altra operazione consentita agli agenti è l'eliminazione dei propri ordini, essa è possibile fornendo il numero dell'ordine da eliminare.

```
public boolean deleteOrder(int id) {

    Order orderToDelete = null;

    for(Order i : Program.getInstance().getOrders()) {
        if(i.getId() == id && i.getAgent().getId() == this.getId()){
            orderToDelete = i;
        }
    }

    if(orderToDelete==null) {
        System.err.println("Wrong ID! Re-insert it");
        return false;
    }

    Program.getInstance().getOrders().remove(orderToDelete);

    return true;

}
```

Listing 5: Implementazione dell'eliminazione di un ordine

3.2.2 Amministratori

Gli amministratori possono gestire agenti, cataloghi e prodotti; per tutti è possibile l'eliminazione, la creazione e la visualizzazione. Alla creazione di un agente sarà necessario fornire, oltre ai dati anagrafici, il catalogo da associare ad essi, in modo che possa vedere e proporre ai clienti solo gli articoli presenti in esso. Lo stesso catalogo può essere assegnato a più agenti.

```
1 public void createAgent(String name, String password, float commission,
      Catalog catalog, String email) {
2     Program.getInstance().getUsers().add(new
          Agent(name,password,commission, catalog,email));
3     System.out.println("Created!");
4 }
```

Listing 6: Implementazione della creazione di Agenti

Nella creazione dei cataloghi è richiesto un ArrayList di articoli che rappresentano i prodotti che saranno contenuti al suo interno.

```
1 public void createCatalog(String description, String marketZone,
      ArrayList<Article> articles) {
2
3     Program.getInstance().getCatalogs().add(new
          Catalog(articles,description,marketZone));
4     System.out.println("Created!");
5 }
```

Listing 7: Implementazione di creazione di Cataloghi

Gli amministratori possono creare entrambi i tipi di prodotti, questa differenziazione è fatta tramite l'overload del metodo *createproduct()*, che in una versione ha come parametri formali il nome e il prezzo, nell'altra il nome ed un ArrayList degli Articoli componenti.

```
1     public void createProduct(String name, ArrayList<Article> a) {
2         Program.getInstance().getArticles().add(new Compound(name, a));
3     }
```

Listing 8: Implementazione della creazione di Prodotti Composti

```
1     public void createProduct(String name, float price) {
2         Program.getInstance().getArticles().add(new Product(name, price));
3     }
```

Listing 9: Implementazione della creazione di Prodotti Semplici

Gli amministratori posso anche cancellare tutte le altre tipologie ma con varie limitazioni per mantenere la consistenza della struttura. Nello specifico non si possono eliminare articoli già ordinati o componenti di composti; non

si possono eliminare cataloghi che sono collegati a degli agenti; non si possono eliminare clienti ai quali sono già associati degli ordini. Infine gli agenti possono essere sempre eliminati, negli ordini in cui comparivano sarà inserito null per mantenere coerente lo storico degli ordini.

```
public void deleteCatalog(int IdCatalog){
    Catalog tmp = null;

    for(User i: Program.getInstance().getUsers()){
        if(i instanceof Agent && ((Agent)
            i).getCatalog().getId()==IdCatalog){
            System.err.println("Catalog Can't be Deleted! It's Linked to an
                User!");
            return;
        }
    }

    for(Catalog i: Program.getInstance().getCatalogs()){
        if(i.getId()==IdCatalog){
            tmp = i;
        }
    }

    if (tmp == null){
        System.err.println("Wrong ID! Re-insert it");
        return;
    }

    Program.getInstance().getCatalogs().remove(tmp);
    System.out.println("Deleted!");
}
```

Listing 10: Implementazione dell'eliminazione di Cataloghi

```
public void deleteCustomer(int idClient) {
    Customer tmp = null;

    for(Order i: Program.getInstance().getOrders()){
        if(i.getClient().getId()==idClient){
            System.err.println("Client Can't be Deleted! It's Linked to an
                Order!");
            return;
        }
    }

    for(Customer i: Program.getInstance().getCustomers()){
        if(i.getId()==idClient){
            tmp = i;
        }
    }

    if (tmp == null){
        System.err.println("Wrong ID! Re-insert it");
        return;
    }
```

```
21
22      Program.getInstance().getCustomers().remove(tmp);
23      System.out.println("Deleted!");
24  }
```

Listing 11: Implementazione dell'eliminazione di Clienti

```
1   public void deleteProduct(int idArticle) {
2       Article tmp = null;
3
4       for(Order i: Program.getInstance().getOrders()){
5           for(Article j:i.getArticles()){
6               if (j.getId()==idArticle){
7                   System.err.println("This Article is Already Ordered! It
                        can't be Deleted!");
8                   return;
9               }
10          }
11      }
12
13      for(Article i: Program.getInstance().getArticles()){
14          if(i instanceof Compound){
15              for(Article j : ((Compound) i).getComponents()){
16                  if (j.getId()==idArticle){
17                      System.err.println("This Article is a Component Of
                            Another Article! It can't be Deleted!");
18                      return;
19                  }
20              }
21          }
22      }
23
24      for(Article i: Program.getInstance().getArticles()){
25          if(i.getId()==idArticle){
26              tmp = i;
27          }
28      }
29
30      if (tmp == null){
31          System.err.println("Wrong ID! Re-insert it");
32          return;
33      }
34
35      for(Catalog i: Program.getInstance().getCatalogs())
36          i.getArticles().removeIf(j -> j.getId() == idArticle);
37
38
39      Program.getInstance().getArticles().remove(tmp);
40      System.out.println("Deleted!");
41  }
```

Listing 12: Implementazione dell'eliminazione di Prodotti

```
1   public void deleteAgent(int idAgent){
2       Agent agent=null;
```

```
3       for(User i : Program.getInstance().getUsers()){
4           if (i instanceof Agent && i.getId()==idAgent){
5               agent = (Agent) i;
6               break;
7           }
8       }
9
10      if (agent==null){
11          System.err.println("Id Agent Doesn't Exist!");
12          return;
13      }
14
15      for(Order i : Program.getInstance().getOrders()){
16          if (i.getAgent()==agent){
17              i.agentDeleted();
18          }
19      }
20
21      Program.getInstance().getUsers().remove(agent);
22      System.out.println("Deleted!");
23  }
```

Listing 13: Implementazione dell'eliminazione di Agenti

3.3 Observer

Il pattern observer è utilizzato per fornire delle notifiche sia amministratori che ai clienti su gli ordini registrati dagli agenti. Il subject è quindi l'agente attivo nel programma, al momento del login gli observer vi verranno legati tramite il metodo *attach()* e al momento del logout verrà e seguito il *dettach()*. Abbiamo già visto l'implementazione di notify, gestito in maniera push, e la sua chiamata all'interno della creazione degli agenti.

Esistono due classi che implementano l'interfaccia di observer, NotificatioEmail e NotificationCenter. La prima invia una mail di notifica a tutti gli amministratori ed al cliente che ha richiesto l'ordine; l'altra consente di avvisare il primo amministratore ad accedere al programma, la notifica viene salvata nel database al momento della chiusura del programma e caricata alla sua riapertura.

```
1   @Override
2   public void update(Object obj) {
3       Order o = (Order)obj;
4       String to = "";
5       for (User u : Program.getInstance().getUsers()){
6           if(u instanceof Administrator)
7               to += u.getEmail() + ",";
8       }
9       to = to.substring(0, to.length() - 1);
10
11      String products="";
12      for (Pair<Article, Integer> a : o.getRows()){
```

```
13          products += "--"+a.getValue0().getName()+" qta:
                "+a.getValue1()+" <br>";
14      }
15
16      String text;
17      text = "<html><meta charset='UTF-8'>" +
18              "<p style='line-height: 2em; font-size: 16px; font-family:
                    Calibri;'>" +
19              "We require your attention regarding the following order:
                    <br>" +
20              "<table>" +
21              "    <tbody>" +
22              "        <tr style='background:#f5f2f2'>" +
23              "            <td style='font-weight: bold; width:20%'>Order
                    number:</td>" +
24              "            <td>"+o.getId()+"</td>" +
25              "        </tr>" +
26              "        <tr style='background:#e1e1e1'>" +
27              "            <td style='font-weight: bold;
                    width:20%'>Agent:</td>" +
28              "            <td>"+o.getAgent().getName()+" --
                    "+o.getAgent().getEmail()+"</td>" +
29              "        </tr>" +
30              "        <tr style='background:#f5f2f2'>" +
31              "            <td style='font-weight: bold;
                    width:20%'>Customer:</td>" +
32              "            <td>"+o.getClient().getBusinessName()+" --
                    "+o.getClient().getEmail()+"</td>" +
33              "        </tr>" +
34              "        <tr style='background:#e1e1e1'>" +
35              "            <td style='font-weight: bold;
                    width:20%'>Total:</td>" +
36              "            <td>"+o.getTotal()+"</td>" +
37              "        </tr>" +
38              "        <tr style='background:#f5f2f2'>" +
39              "            <td style='font-weight: bold;
                    width:20%'>Commission:</td>" +
40              "            <td>"+o.getCommissionTot()+"</td>" +
41              "        </tr>" +
42              "        <tr style='background:#e1e1e1'>" +
43              "            <td style='font-weight: bold;
                    width:20%'>Products:</td>" +
44              "            <td>"+products+"</td>" +
45              "        </tr>" +
46              "    </tbody>" +
47              "</table>" +
48              "</p>" +
49              "</html>";
50      sendEmail(to, "A new order has been issued!", text);
51      text = "<html><meta charset='UTF-8'>" +
52              "<p style='line-height: 2em; font-size: 16px; font-family:
                    Calibri;'>" +
53              "We require your attention regarding the following order:
                    <br>" +
54              "<table>" +
55              "    <tbody>" +
56              "        <tr style='background:#f5f2f2'>" +
```

```
57                  "            <td style='font-weight: bold; width:20%'>Order
                        number:</td>" +
58                  "            <td>"+o.getId()+"</td>" +
59                  "        </tr>" +
60                  "        <tr style='background:#e1e1e1'>" +
61                  "            <td style='font-weight: bold;
                        width:20%'>Agent:</td>" +
62                  "            <td>"+o.getAgent().getName()+" --
                        "+o.getAgent().getEmail()+"</td>" +
63                  "        </tr>" +
64                  "        <tr style='background:#f5f2f2'>" +
65                  "            <td style='font-weight: bold;
                        width:20%'>Customer:</td>" +
66                  "            <td>"+o.getClient().getBusinessName()+" --
                        "+o.getClient().getEmail()+"</td>" +
67                  "        </tr>" +
68                  "        <tr style='background:#e1e1e1'>" +
69                  "            <td style='font-weight: bold;
                        width:20%'>Total:</td>" +
70                  "            <td>"+o.getTotal()+"</td>" +
71                  "        </tr>" +
72                  "        <tr style='background:#f5f2f2'>" +
73                  "            <td style='font-weight: bold;
                        width:20%'>Products:</td>" +
74                  "            <td>"+products+"</td>" +
75                  "        </tr>" +
76                  "    </tbody>" +
77                  "</table>" +
78                  "</p>" +
79                  "</html>";
80      sendEmail(o.getClient().getEmail(), "Your order has been taken over!",
            text);
81  }
```

Listing 14: Implementazione di Update in NotificatioEmail

```
1   private void sendEmail(String to, String obj, String text){
2
3       final String username = "mirkomacaluso.swe@gmail.com";
4       final String password = "SWEMirkoMacaluso";
5
6       Properties prop = new Properties();
7       prop.put("mail.smtp.host", "smtp.gmail.com");
8       prop.put("mail.smtp.port", "465");
9       prop.put("mail.smtp.auth", "true");
10      prop.put("mail.smtp.socketFactory.port", "465");
11      prop.put("mail.smtp.socketFactory.class",
            "javax.net.ssl.SSLSocketFactory");
12
13      Session session = Session.getInstance(prop,
14              new javax.mail.Authenticator() {
15                  protected PasswordAuthentication
                        getPasswordAuthentication() {
16                      return new PasswordAuthentication(username,
                            password);
```

```
                }
            });

    try {

        Message message = new MimeMessage(session);
        message.setFrom(new InternetAddress("from@gmail.com"));
        message.setRecipients(
                Message.RecipientType.TO,
                InternetAddress.parse(to)
        );
        message.setSubject(obj);
        message.setContent(text, "text/html");

        Transport trans = session.getTransport("smtp");
        trans.connect("smtp.gmail.com", 465,
            "mirkomacaluso.swe@gmail.com", "SWEMirkoMacaluso");
        trans.sendMessage(message, message.getAllRecipients());

    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 15: Implementazione di SendEmail

Figure 8: Email inviata agli amministratori

```
@Override
public void update(Object obj) {
    Order order = (Order)obj;
    this.notification.add("A new order has been issued by for customer " +
        order.getClient().getBusinessName() + " from " +
        order.getAgent().getName());
}
```

Listing 16: Implementazione di Update in NotificationCenter

3.4 Program

La classe program permette il caricamento dei dati dal database attraverso la funzione *load(Connection c)*; i dati vengono recuperati attraverso delle query al database indicato nella connection e caricati all'interno degli arrayList della classe.

```
public void load(Connection c) throws SQLException {

    Statement stmt = c.createStatement();
    Statement stmt1 = c.createStatement();
    ResultSet rs, rs1;

    rs = stmt.executeQuery("SELECT * FROM Customer;");
    while (rs.next()) {
        int id = rs.getInt("id");
        String businessName = rs.getString("BusinessName");
        String country = rs.getString("Country");
        String email = rs.getString("Email");
        customers.add(new Customer(id, businessName, country, email));
    }

    rs = stmt.executeQuery("SELECT * FROM Notification;");
    while (rs.next()) {
        String message = rs.getString("message");
        notCenter.addNotification(message);
    }

    rs = stmt.executeQuery("SELECT * FROM Article WHERE id not in
        (SELECT IdCompound FROM ArticleCompound );");
    while (rs.next()) {
        int id = rs.getInt("id");
        String name = rs.getString("name");
        float price = rs.getFloat("Price");
        articles.add(new Product(name, price, id));
    }
    rs = stmt.executeQuery("SELECT * FROM Article WHERE id in
        (SELECT IdCompound FROM ArticleCompound );");
    while (rs.next()) {
        int id = rs.getInt("id");
        String name = rs.getString("name");
        ArrayList<Article> components = new ArrayList<>();
```

```
34          rs1 = stmt1.executeQuery("SELECT * FROM ArticleCompound
                WHERE IdCompound = " + id + " ;");
35          while (rs1.next()) {
36              int idComponent = rs1.getInt("idComponent");
37
38              for (Article a : articles) {
39                  if (a.getId() == idComponent) {
40                      components.add(a);
41                      break;
42                  }
43              }
44          }
45          articles.add(new Compound(name, components, id));
46      }
47
48      rs = stmt.executeQuery("SELECT * FROM CatalogHead;");
49      while (rs.next()) {
50          int id = rs.getInt("idHead");
51          String description = rs.getString("Description");
52          String marketZone = rs.getString("MarketZone");
53          ArrayList<Article> tmp = new ArrayList<>();
54          rs1 = stmt1.executeQuery("SELECT * FROM CatalogRow
                WHERE IdHead = " + id + " ;");
55          while (rs1.next()) {
56              int idArticle = rs1.getInt("idArticle");
57              for (Article a : articles) {
58                  if (a.getId() == idArticle) {
59                      tmp.add(a);
60                      break;
61                  }
62              }
63          }
64          catalogs.add(new Catalog(tmp, description, marketZone, id));
65      }
66      rs = stmt.executeQuery("SELECT * FROM User;");
67      while (rs.next()) {
68          int id = rs.getInt("id");
69          String name = rs.getString("Name");
70          String passHash = rs.getString("Passwordhash");
71          int type = rs.getInt("Type");
72          int idCatalog = rs.getInt("IdCatalog");
73          float commissionPercentage = rs.getFloat("CommissionPerc");
74          String email = rs.getString("email");
75
76          if (type == 1) {
77              Catalog tmp = null;
78              for(Catalog i: catalogs) {
79                  if (i.getId()==idCatalog){
80                      tmp = i;
81                  }
82              }
83
84              if(tmp == null){
85                  System.err.println("Catalog don't exist!");
86                  break;
87              }
```

```
88
89              users.add(new Agent(name, passHash, commissionPercentage,
                    tmp,email,id));
90          } else {
91              users.add(new Administrator(name, passHash, email,id));
92          }
93      }
94      rs = stmt.executeQuery("SELECT * FROM OrderHead;");
95      while (rs.next()) {
96          int id = rs.getInt("idHead");
97          int idAgent = rs.getInt("idAgent");
98          int idCustomers = rs.getInt("IdCustomer");
99          float total = rs.getFloat("Total");
100         float commission = rs.getFloat("Commission");
101
102         Agent tmpAgent = null;
103         for (User i : users) {
104             if (i.getId() == idAgent) {
105                 tmpAgent = (Agent) i;
106                 break;
107             }
108         }
109
110         Customer tmpCustomer = null;
111         for (Customer i : customers) {
112             if (i.getId() == idCustomers) {
113                 tmpCustomer = i;
114                 break;
115             }
116         }
117
118         if(tmpCustomer == null){
119             System.err.println("Customer don't exist!");
120             break;
121         }
122
123         ArrayList<Pair<Article, Integer>> tmp = new ArrayList<>();
124         rs1 = stmt1.executeQuery("SELECT * FROM OrderRow WHERE
                IdHead = " + id + " ;");
125         while (rs1.next()) {
126             int idArticle = rs1.getInt("idArticle");
127             int qta = rs1.getInt("qta");
128             for (Article a : articles ) {
129                 if (a.getId() == idArticle) {
130                     tmp.add(new Pair<>(a,qta));
131                     break;
132                 }
133             }
134         }
135         orders.add(new Order(total, commission, tmpAgent, tmp,
                tmpCustomer, id));
136     }
137
138 }
```

Listing 17: Implementazione di Load

Program consente anche l'operazione opposta, upload(), che elimina tutte le *entry* nel DB ed inserisce le nuove tuple corrispondenti ai dati salvati negli arrayList dentro la classe.

```
public void upload(Connection c) {
    String sql;
    Statement stmt = null;
    try {
        stmt = c.createStatement();
        for (String s : Arrays.asList("DELETE FROM User;", "DELETE
            FROM OrderHead;", "DELETE FROM OrderRow;", "DELETE
            FROM Notification;", "DELETE FROM Customer;", "DELETE
            FROM CatalogRow;", "DELETE FROM CatalogHead;",
            "DELETE FROM Article;", "DELETE FROM
            ArticleCompound;")) {
            sql = s;
            stmt.executeUpdate(sql);
            c.commit();
        }
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ": " + e.getMessage());
        System.exit(0);
    }

    int type;
    float perch;
    for (User user : users) {
        try {
            if (!(user instanceof Agent)) {
                type = 0;
                perch = 0;
                sql = "INSERT INTO User
                    (Id,Name,PasswordHash,Type,CommissionPerc,email) "
                    + "VALUES (" + user.getId() + ", '" + user.getName()
                    + "', '" + user.getPasswordHash() + "', " + type + ", "
                    + perch + " ,'"+user.getEmail()+"');";
            } else {
                type = 1;
                Agent tmp = (Agent) user;
                perch = tmp.getCommissionPercentage();
                sql = "INSERT INTO User
                    (Id,Name,PasswordHash,Type,CommissionPerc,IdCatalog,email)
                    " + "VALUES (" + user.getId() + ", '" +
                    user.getName() + "', '" + user.getPasswordHash() + "',
                    " + type + ", " + perch + "
                    ,"+tmp.getCatalog().getId()+"
                    ,'"+user.getEmail()+"');";
            }

            stmt = c.createStatement();
            stmt.executeUpdate(sql);
            c.commit();
        } catch (Exception e) {
            System.err.println(e.getClass().getName() + ": " +
                e.getMessage());
        }
```

```
37      }
38
39      for (Customer customer : customers) {
40          try {
41              sql = "INSERT INTO Customer
                    (id,BusinessName,Country,Email) " + "VALUES (" +
                    customer.getId() + ", '" + customer.getBusinessName() +
                    "', '" + customer.getCountry() + "', '" +
                    customer.getEmail() + "');";
42              stmt = c.createStatement();
43              stmt.executeUpdate(sql);
44              c.commit();
45          } catch (Exception e) {
46              System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
47          }
48      }
49
50      for (Order order : orders) {
51          try {
52              if(order.getAgent()!=null)
53                  sql = "INSERT INTO OrderHead
                        (idHead,idAgent,IdCustomer,Total,Commission) " +
                        "VALUES (" + order.getId() + ", '" +
                        order.getAgent().getId() + "',
                        "+order.getClient().getId()+" ,'" + order.getTotal() +
                        "', '" + order.getCommissionTot() + "');";
54              else
55                  sql = "INSERT INTO OrderHead
                        (idHead,idAgent,IdCustomer,Total,Commission) " +
                        "VALUES (" + order.getId() + ", '" + -1 + "',
                        "+order.getClient().getId()+" ,'" + order.getTotal() +
                        "', '" + order.getCommissionTot() + "');";
56              stmt = c.createStatement();
57              stmt.executeUpdate(sql);
58              c.commit();
59          } catch (Exception e) {
60              System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
61          }
62          try {
63              for (Pair<Article,Integer> i: order.getRows()) {
64                  sql = "INSERT INTO OrderRow (idHead,idArticle,qta) " +
                        "VALUES (" + order.getId() + ", " +
                        i.getValue0().getId() + ","+i.getValue1()+");";
65                  stmt = c.createStatement();
66                  stmt.executeUpdate(sql);
67                  c.commit();
68              }
69          } catch (Exception e) {
70              System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
71          }
72      }
73
74      for (Catalog catalog : catalogs) {
```

```
75          try {
76              sql = "INSERT INTO CatalogHead
                    (idHead,Description,MarketZone) " + "VALUES (" +
                    catalog.getId() + ", '" + catalog.getDescription() + "', '" +
                    catalog.getMarketZone() + "');";
77              stmt = c.createStatement();
78              stmt.executeUpdate(sql);
79              c.commit();
80          } catch (Exception e) {
81              System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
82          }
83          try {
84              for (Article article : catalog.getArticles()) {
85                  sql = "INSERT INTO CatalogRow (idHead,idArticle) " +
                        "VALUES (" + catalog.getId() + ", " + article.getId()
                        + ");";
86                  stmt = c.createStatement();
87                  stmt.executeUpdate(sql);
88                  c.commit();
89              }
90          } catch (Exception e) {
91              System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
92          }
93
94      }
95
96      for (Article article : articles) {
97          if (article instanceof Compound) {
98              Compound tmp = (Compound) article;
99              for (Article a : tmp.getComponents()) {
100                 try {
101                     sql = "INSERT INTO ArticleCompound
                            (IdCompound,IdComponent) " + "VALUES (" +
                            article.getId() + ", " + a.getId() + ");";
102                     stmt = c.createStatement();
103                     stmt.executeUpdate(sql);
104                     c.commit();
105                 } catch (Exception e) {
106                     System.err.println(e.getClass().getName() + ": " +
                            e.getMessage());
107                 }
108             }
109         }
110         try {
111             sql = "INSERT INTO Article (Id,Name,Price) " + "VALUES ("
                    + article.getId() + ", '" + article.getName() + "', '" +
                    article.getPrice() + "');";
112             stmt = c.createStatement();
113             stmt.executeUpdate(sql);
114             c.commit();
115         } catch (Exception e) {
116             System.err.println(e.getClass().getName() + ": " +
                    e.getMessage());
```

```
117             }
118         }
119
120         for (String notify : notCenter.getNotification()) {
121             try {
122                 sql = "INSERT INTO Notification (Message) " + "VALUES ('"
                        + notify + "');";
123                 stmt = c.createStatement();
124                 stmt.executeUpdate(sql);
125                 c.commit();
126             } catch (Exception e) {
127                 System.err.println(e.getClass().getName() + ": " +
                        e.getMessage());
128             }
129         }
130
131         try {
132             stmt.close();
133             c.close();
134         } catch (Exception e2) {
135             e2.printStackTrace();
136         }
137
138         instance = null;
139     }
```

Listing 18: Implementazione di Upload

Il metodo *run()* rappresenta il loop di sistema, è utilizzato esternamente per eseguire il programma attraverso l'interfaccia da console, esso provvede ad instaurare la connessione al database tramite DBConnection, chiede i dati per il login ed effettua un ciclo sulla variabile booleana *wantClose* nel quale chiama *showMenu()* del menù attivo, che chiederà input dalla console consentendo all'utente di interagire con il programma. Quando sarà selezionata la voce del menù che chiama *close()* la variabile *wantClose* diventerà true interrompendo il ciclo. L'ultima operazione eseguita da *run()* è la chiamata al metodo upload().

```
1   public void run() {
2
3       try {
4           this.load(DBConnection.getInstance());
5       } catch (SQLException e) {
6           System.err.println("Unable to load!");
7           return ;
8       }
9       this.setMenu(new LoginMenu());
10
11      while (!wantClose)
12          menu.showMenu();
13      System.out.println("Bye Bye!");
14      this.upload(DBConnection.getInstance());
15
16  }
```

Listing 19: Implementazione di Run

4 Unit Test

In questa sezione saranno presi in analisi i test effettuati; è stata creata una base dati apposita per eseguire i test. Essi sono stati suddivisi in tre categorie, test sulle funzioni di amministratore, di agente e test generali. Tutti condividono la funzione *prepare()*, che esegue il setup della connessione con il database di test, attraverso *DBConnectionTest*. I test sono stati eseguiti con la libreria Junit 5 .

4.1 General test

I test in questa sezione hanno lo scopo di verificare il funzionamento delle operazioni principali della classe *Program*.

4.1.1 Test Login

Questo test mira a verificare la procedura di login sia nel caso di amministratore e che di agente.

```
@Test
@DisplayName("Login user Test")
void testLoginUser() {

    p.login("Agent1","111");

    User user = null;

    for (User i:p.getUsers()){
        if (i.getName().equals("Agent1")){
            user = i;
        }
    }

    User expectedUser1 = user;

    assertAll("Agent Login",
            () -> assertEquals(expectedUser1, p.getActiveUser()),
            () -> assertTrue(p.getActiveUser() instanceof Agent)
    );

    p.logout();

    p.login("Admin","111");

    user = null;

    for (User i:p.getUsers()){
        if (i.getName().equals("Admin")){
            user = i;
        }
    }

    User expectedUser2 = user;

    assertAll("Admin Login",
            () -> assertEquals(expectedUser2, p.getActiveUser()),
```

```
38              () -> assertTrue(p.getActiveUser() instanceof Administrator)
39      );
40  }
```

Listing 20: Implementazione del test Login

4.1.2 Test Load/Upload

Questo test verifica il corretto caricamento dei dati da e verso Database.

Vengono eseguite delle query di inserimento manualmente ed in seguito viene chiamato il metodo di caricamento dei dati dal DB *load(Connection c)* e si verifica che i nuovi dati inseriti siano stati ricaricati correttamente all'interno delle strutture dati utilizzate nel programma.

Dopo aver quindi verificato il corretto funzionamento del metodo che invia i dati al DB viene inserito manualmente all'interno del programma alcuni articoli ed alcuni clienti per poi poter chiamare il metodo *upload(Connection c)* e successivamente andare ad effettuare sul DB delle query di ricerca per verificarne il corretto funzionamento.

```
1   @Test
2   @DisplayName("Upload/Load data Test")
3   void testUploadLoadData() throws SQLException {
4       String sql;
5       Statement stmt;
6       ResultSet rs;
7
8       Connection c = DBConnectionTest.getInstance();
9       p.upload(c);
10      p = Program.getInstance();
11
12      c = DBConnectionTest.getInstance();
13
14      String customerName = "testCustomer";
15      String customerCountry = "testCountry";
16      String customerEmail = "testemail";
17
18      sql = "INSERT INTO Customer (BusinessName,Country,Email) " +
            "VALUES ('"+customerName + "', '" + customerCountry+ "', '" +
            customerEmail+ "');";
19
20      stmt = c.createStatement();
21      stmt.executeUpdate(sql);
22      c.commit();
23
24      String articleName = "testCustomer";
25      float articlePrice = 10.2F;
26
27      sql = "INSERT INTO Article (Name,Price) " + "VALUES ('"
            +articleName+ "', '" + articlePrice+ "');";
28      stmt = c.createStatement();
29      stmt.executeUpdate(sql);
30      c.commit();
31
32      p.load(c);
```

```
33
34      Customer newCustomer = null;
35      Article newArticle = null;
36
37      for (Customer i:p.getCustomers()){
38          if (i.getBusinessName().equals(customerName) &&
                i.getCountry().equals(customerCountry) &&
                i.getEmail().equals(customerEmail)){
39              newCustomer=i;
40          }
41      }
42
43      for (Article i:p.getArticles()){
44          if (i.getName().equals(articleName) &&i.getPrice()==articlePrice){
45              newArticle=i;
46          }
47      }
48
49      Customer finalNewCustomer = newCustomer;
50      Article finalNewArticle = newArticle;
51      assertAll("Load From DB",
52              () -> assertNotNull(finalNewCustomer),
53              () -> assertNotNull(finalNewArticle)
54      );
55
56      p.login("Admin","111");
57      Administrator admin = (Administrator) p.getActiveUser();
58
59      admin.createProduct("TestProductUpload",12.2F);
60      admin.createCustomer("TestCustomerUpload", "TestCustomerUpload",
            "TestCustomerUpload" );
61
62      p.upload(c);
63
64      c = DBConnectionTest.getInstance();
65      stmt = c.createStatement();
66
67      int risArticle = 0;
68      rs = stmt.executeQuery("SELECT COUNT(*) as ris FROM Article
            WHERE name='TestProductUpload';");
69      while (rs.next()) {
70          risArticle = rs.getInt("ris");
71      }
72
73      int risCustomer = 0;
74      rs = stmt.executeQuery("SELECT COUNT(*) as ris FROM Customer
            WHERE businessname='TestCustomerUpload' AND
            country='TestCustomerUpload' AND
            email='TestCustomerUpload';");
75      while (rs.next()) {
76          risCustomer = rs.getInt("ris");
77      }
78
79      int finalRisArticle = risArticle;
80      int finalRisCustomer = risCustomer;
81      assertAll("Upload To DB",
82              () -> assertTrue(finalRisArticle>=1),
83              () -> assertTrue(finalRisCustomer>=1)
```

```
84      );
85
86      sql = "DELETE FROM Customer WHERE LOWER(businessname)
            LIKE '%test%';";
87      stmt.executeUpdate(sql);
88      c.commit();
89
90      sql = "DELETE FROM Article WHERE LOWER(name) LIKE
            '%test%';";
91      stmt.executeUpdate(sql);
92      c.commit();
93  }
```

Listing 21: Implementazione del test Upload/Load

4.2 Administrator Test

I test in questa sezione hanno lo scopo di verificare il funzionamento delle operazioni principali della classe *Administrator*.

4.2.1 Test creazione/eliminazione di un agente

In questo test viene creato manualmente un agente ed inserito all'interno del programma e successivamente si verifica il corretto inserimento di esso.

```
1   @Test
2   @DisplayName("Create Agent Test")
3   void testCreateAgent() {
4
5       Catalog catalog = p.getCatalogs().get( (int)((Math.random() *
            (p.getCatalogs().size()-1 - 1)) + 1) );
6       admin.createAgent("UnitTest",
            "111",5.5F,catalog,"unitTest@gmail.com");
7       User createUser = p.getUsers().get(p.getUsers().size()-1);
8
9       assertTrue(createUser instanceof Agent);
10
11      Agent createAgent = (Agent)createUser;
12
13      assertAll("Test create agent",
14              () -> assertEquals(catalog, createAgent.getCatalog()),
15              () -> assertEquals(createAgent.getId(),
                    p.getUsers().get(p.getUsers().size()-1).getId())
16      );
17  }
```

Listing 22: Implementazione di test create Agent

In questo test si verifica la corretta eliminazione di un agente da parte di un amministratore e nel caso in cui l'agente abbia già effettuato degli ordini essi dovranno perdere il riferimento all'agente in quanto devono rimanere nello storico globale degli ordini.

```java
@Test
@DisplayName("Delete Agent Test")
void testDeleteAgent() {
    int id = 4;
    User userDel = null;
    admin.deleteAgent(id);
    boolean check = false;

    for (User u : p.getUsers()){
        if (u.getId() == id)
            userDel = u;
        check = u instanceof Administrator;
    }
    if (check)
        assertTrue(p.getUsers().contains(userDel));
    else {
        assertFalse(p.getUsers().contains(userDel));
        ArrayList<Order> localOrders = new ArrayList<>();
        for (Order o : p.getOrders()){
            if (o.getAgent() == userDel){
                localOrders.add(o);
            }
        }
        for (Order o : localOrders)
            assertNull(o.getAgent());
    }
}
```

Listing 23: Implementazione del test delete Agent

4.2.2 Test creazione/eliminazione di un catalogo

In questo test viene creato manualmente un catalogo, con annessa una lista di articoli, all'interno del programma e successivamente si verifica il corretto inserimento di esso.

```java
@Test
@DisplayName("Create Catalog Test")
void testCreateCatalog() {
    ArrayList<Article> articles = new ArrayList<>();
    articles.add(p.getArticles().get(1));
    articles.add(p.getArticles().get(2));
    articles.add(p.getArticles().get(3));
    int preSize = p.getCatalogs().size();
    admin.createCatalog("description","Italy", articles);
    assertAll("Test create agent",
            () -> assertEquals(preSize + 1, p.getCatalogs().size()),
            () -> assertEquals(articles,
                p.getCatalogs().get(p.getCatalogs().size()-1).getArticles())
    );
}
```

Listing 24: Implementazione del test create catalog

In questo test si verifica la corretta eliminazione di un catalogo. Ci sono due casi possibili: il primo è che il catalogo sia già stato assegnato ad un'agente e quindi non può essere eliminato, il secondo invece è il caso in cui il catalogo sia completamente svincolato da un agente e che quindi possa essere eliminato. Il test verifica il corretto funzionamento delle due casistiche.

```
@Test
@DisplayName("Delete Catalog Test")
void testDeleteCatalog() {

    int preSize = p.getCatalogs().size();
    boolean check;

    check = checkCatalog(1);
    admin.deleteCatalog(1);

    if(check)
        assertEquals(preSize, p.getCatalogs().size());
    else
        assertEquals(preSize - 1, p.getCatalogs().size());

    ArrayList<Article> articles = new ArrayList<>();
    articles.add(p.getArticles().get(1));
    articles.add(p.getArticles().get(2));
    articles.add(p.getArticles().get(3));
    admin.createCatalog("description","Italy", articles);
    preSize = p.getCatalogs().size();

    int lastCat = p.getCatalogs().get(p.getCatalogs().size()-1).getId();
    check = checkCatalog(lastCat);

    admin.deleteCatalog(lastCat);

    if(check)
        assertEquals(preSize, p.getCatalogs().size());
    else
        assertEquals(preSize - 1, p.getCatalogs().size());

}
```

Listing 25: Implementazione del test delete catalog

4.2.3 Test creazione/eliminazione di un prodotto

In questo test vengono creati ed aggiunti manualmente due tipi di articoli. Il primo è un articolo semplice e viene controllato se esso è stato aggiunto correttamente. Il secondo è un articolo composto da più prodotti (*composite*) viene quindi eseguito un controllo sul corretto inserimento di esso e sul prezzo che deve essere calcolato come somma dei prezzi degli articoli che lo compongono.

```
@Test
@DisplayName("Create Product Test")
void testCreateProduct() {
    int preSize = p.getArticles().size();
    admin.createProduct("ProductTestSingle",3.5F);

    assertAll("Single Article",
            () -> assertEquals(preSize + 1, p.getArticles().size()),
            () -> assertTrue(p.getArticles().get(p.getArticles().size()-1)
                instanceof Product),
            () ->
                assertEquals(p.getArticles().get(p.getArticles().size()-1).getPrice(),
                3.5F)
    );

    int preSize2 = p.getArticles().size();
    ArrayList<Article> articles = new ArrayList<>();
    articles.add(p.getArticles().get(1));
    articles.add(p.getArticles().get(2));

    float tmp = 0;
    for (Article a : articles)
        tmp += a.getPrice();
    float prePrice = tmp;

    admin.createProduct("Compound article",articles);

    assertAll("Compound Article",
            () -> assertEquals(preSize2 + 1, p.getArticles().size()),
            () -> assertTrue(p.getArticles().get(p.getArticles().size()-1)
                instanceof Compound),
            () ->
                assertEquals(p.getArticles().get(p.getArticles().size()-1).getPrice(),
                prePrice)

    );

}
```

Listing 26: Implementazione del test create product

In questo test si verifica la corretta eliminazione di un prodotto. Possono verificarsi 3 condizioni durante l'eliminazione.

La prima condizione è che Il prodotto che si cerca di eliminare è già stato ordinato almeno una volta da un cliente ed in questa situazione non è possibile eliminare il prodotto.

La seconda condizione si verifica nel momento in cui si cerca di cancellare un articolo che ne compone un altro, se questa condizione si verifica l'eliminazione di tale articolo deve essere bloccata.

La terza condizione si verifica nel momento in cui non si verifichino le prime due e quindi l'articolo che stiamo cercando di eliminare può essere effettivamente eliminato.

Il test verifica il corretto funzionamento di tutte e tre le condizioni.

```
@Test
@DisplayName("Delete Product Test")
void testDeleteArticle() {

    admin.createProduct("testProduct1 - can_delete",5.5F);
    Article P1 = p.getArticles().get(p.getArticles().size()-1);
    int A1 = p.getArticles().get(p.getArticles().size()-1).getId();
    admin.deleteProduct(A1);

    assertFalse(p.getArticles().contains(P1));

    int A2 = 1;
    Article P2 = null;

    for(Article a : p.getArticles()){
        if(a.getId()==1) {
            P2 = a;
        }
    }
    admin.deleteProduct(A2);
    assertTrue(p.getArticles().contains(P2));

    ArrayList<Article> articles = new ArrayList<>();
    articles.add(p.getArticles().get(2));
    articles.add(p.getArticles().get(3));
    admin.createProduct("testProduct2", articles);
    Article P3 = p.getArticles().get(2);
    int A3 = p.getArticles().get(2).getId();
    admin.deleteProduct(A3);

    assertTrue(p.getArticles().contains(P3));

}
```

Listing 27: Implementazione del test delete product

4.2.4 Test eliminazione di un cliente

In questo test si verifica la corretta eliminazione di un cliente da parte di un amministratore, possono verificarsi due condizioni.

La prima si verifica nel momento in cui un cliente che si vuole eliminare ha effettuato almeno un ordine ed in questo caso l'eliminazione deve essere bloccata.

La secondo si verifica nel momento in cui il cliente che si vuole eliminare non ha mai effettuato un ordine ed in questo caso può essere eliminato.

```

@Test
@DisplayName("Delete CustomerTest")
void testDeleteCustomer() {

    Customer C1 = null;
    Customer C2 = null;

    for (Customer cli : p.getCustomers()){
        if(cli.getId() == 1)
```

```
            C1 = cli;
        if(cli.getId() == 2)
            C2 = cli;
    }

    admin.deleteCustomer(1);
    assertFalse(p.getCustomers().contains(C1));

    admin.deleteCustomer(2);
    assertTrue(p.getCustomers().contains(C2));

}
```

Listing 28: Implementazione del test create customer

4.3 Agent test

I test in questa sezione hanno lo scopo di verificare il funzionamento delle operazioni principali della classe *Agent*.

4.3.1 Test creazione/eliminazione di un ordine

Il test verifica il corretto funzionamento del metodo di creazione di un ordine da parte di un agente. Viene creata un lista di articoli con annessa la quantità ordinata e viene anche creato un cliente di test per poter creare un ordine ed aggiungerlo manualmente. Viene quindi verificato successivamente che l'ordine sia stato creato correttamente e che sia stata inviata una notifica agli amministratori.

```
@Test
@DisplayName("Create Order Test")
void testOrderCreation() {

    ArrayList<Pair<Article,Integer>> articles = new ArrayList<>();

    articles.add(new Pair<>(agent.getCatalog().getArticles().get(1),20));
    articles.add(new Pair<>(agent.getCatalog().getArticles().get(2),50));

    Customer customer = p.getCustomers().get(2);

    agent.createOrder(customer,articles);

    Order createdOrder = p.getOrders().get(p.getOrders().size()-1);

    String messageNotification =
        Program.getInstance().getNotificationCenter().getNotification().get(
    Program.getInstance().getNotificationCenter().getNotification().size()-1);

    assertAll("Order's Data",
            () -> assertEquals(createdOrder.getAgent(), agent),
            () -> assertEquals(createdOrder.getClient(), customer),
            () -> assertEquals(createdOrder.getArticles().get(0),
                articles.get(0).getValue0()),
```

```
23              () -> assertEquals(createdOrder.getArticles().get(1),
                    articles .get(1).getValue0()),
24              () ->
                    assertTrue(messageNotification.contains(agent.getName()))
25      );
26
27  }
```

Listing 29: Implementazione del test create order

In questo test si verifica la corretta eliminazione di un ordine da parte di un agente. Possono essere eliminati solamente gli ordini che appartengono all'agente che sta effettuando l'operazione in caso contrario l'operazione deve essere bloccata.

In questo test vengono verificate entrambe le condizioni.

```
1   @Test
2   @DisplayName("Delete Order Test")
3   void testDeleteOrder() {
4
5       Order order = p.getOrders().get(0);
6
7       for (Order i: p.getOrders()){
8           if (i.getAgent()==agent){
9               order = i;
10          }
11      }
12
13      int orderCountBefore1 = p.getOrders().size();
14
15      agent.deleteOrder(order.getId());
16
17      Order finalOrder1 = order;
18      assertAll("Order deleted",
19              () -> assertTrue(p.getOrders().size()<orderCountBefore1),
20              () -> assertFalse(p.getOrders().contains(finalOrder1))
21      );
22
23      for (Order i: p.getOrders()){
24          if (i.getAgent()!=agent){
25              order = i;
26          }
27      }
28
29      int orderCountBefore2 = p.getOrders().size();
30
31      agent.deleteOrder(order.getId());
32
33      Order finalOrder2 = order;
34      assertAll("Order deleted",
35              () -> assertEquals(orderCountBefore2, p.getOrders().size()),
36              () -> assertTrue(p.getOrders().contains(finalOrder2))
37      );
38
39  }
```

Listing 30: Implementazione del test delete order

5 Conclusioni

L'idea originale del progetto era quella di sviluppare una applicazione mobile che permettesse di automatizzare la registrazione degli ordini da parte degli agenti commerciali, canalizzandone il flusso informativo all'interno del programma. Dopo il necessario ridimensionamento del progetto e il passaggio all'architettura desktop abbiamo deciso di preservare l'essenza del progetto originale mantenendone l'usabilità, seppur con un interfaccia molto semplice, ma efficace. Ovviamente sono stati necessari alcuni adattamenti che non hanno però minato le funzionalità originali del progetto, ad esempio la lettura del barcode. Un obiettivo trasversale del nostro progetto è stato quello di rendere fattibile la transizione verso un architettura mobile, ad esempio attraverso un utilizzo alternativo dello state pattern.

6 Librerie utilizzate

Per la realizzazione del progetto sono state utilizzate le seguenti librerie:

- SQLite-jdbc 3.8.11.2, per la gestione del database.
- JavaTuples-1.2, per le associazioni articolo-quantità tramite la struttura Pair.
- javax.mail e javax.activation 1.2.0, per l'invio delle notifiche tramite mail.
- JUnit 5, per lo unit-testing.