Dentist management system

Ingegneria del software

Leonardo Pampaloni, Filippo di Martino

Computer Engineering
University Of Florence
Italy

1 Introduzione

L'idea del progetto nasce come applicazione per la gestione di interventi fatti da dentisti e assistenti su vari clienti. Nell'applicazione sono presenti principalmente:

1. ***Dentista***: Il dentista ha la possibilità di controllare tutte le informazioni riguardanti i clienti, gli assistenti e i set di interventi assegnati agli Assistenti.
2. ***Assistenti***: Ogni assistente ha a sua disposizione un set predefinito di interventi da poter fare, oltre che poter vedere le informazioni sui clienti.
3. ***Clienti***
4. ***Interventi***

L'applicazione ha come obiettivo quello di gestire e notificare l'admin, ovvero il dentista, ti tutti gli interventi effettuati dagli assistenti, inoltre è compresa nell'applicazione il salvataggio dei clienti, degli articoli e degli interventi all'interno di un database. Sono presenti due tipologie di notifica per l'admin, uno all'interno dell'applicazione, che tiene traccia dello storico delle operazioni fatte da tutti gli assistenti mentre l'altro è esterno all'applicazione e viene notificato l'admin tramite email ogni volta che un assistente effettua un intervento su un cliente.

2 Diagrammi UML

Abbiamo scelto di presentare tre diversi diagrammi UML, il diagramma delle classi (*Class Diagram*), il diagramma dei casi d'uso (*Use Case Diagram*), e il diagramma E/R (*Entity Relationship* del progetto.

2.1 Class Diagram

Dal *Class Diagram* possiamo vedere come effettivamente sono legate le varie classi del programma. Si può notare infatti che le operazioni di notifica sono effettuate da un *Observer pattern*, suddiviso in due diverse classi per le due tipologie differenti di notifica. Sono presenti anche altri due pattern: lo *State pattern* e il *Composite pattern*, rispettivamente per la gestione delle classi dei vari menù e per la gestione delle classi per le operazioni/interventi.
Il Dentista (*Admin*,ha il compito di creare Clienti, Articoli, e set di Operazioni/Interventi. Per le operazioni abbiamo incluso la possibilità di raccogliere più interventi in uno (creandolo tramite il *Composite pattern*), oltre ovviamente a poter selezionare l'uso di più articoli e strumenti (E.g. Creazione di un kit-monouso (Guanti,Bicchiere,Tovaglietta), e includerlo in tutte le operazioni).
La classe *Program* è il cuore della gestione dell'applicazione: rappresenta il

nodo centrale del sistema. *Program* è una classe ***Singleone*** in quanto si necessita di avere una singola istanza di essa e deve essere necessariamente reperibile, al suo interno sono presenti inoltre metodi come *load(Connection c* e *upload(Connection c)* che permettono la comunicazione con il DB esterno e *run()* per poter gestire il loop di sistema.

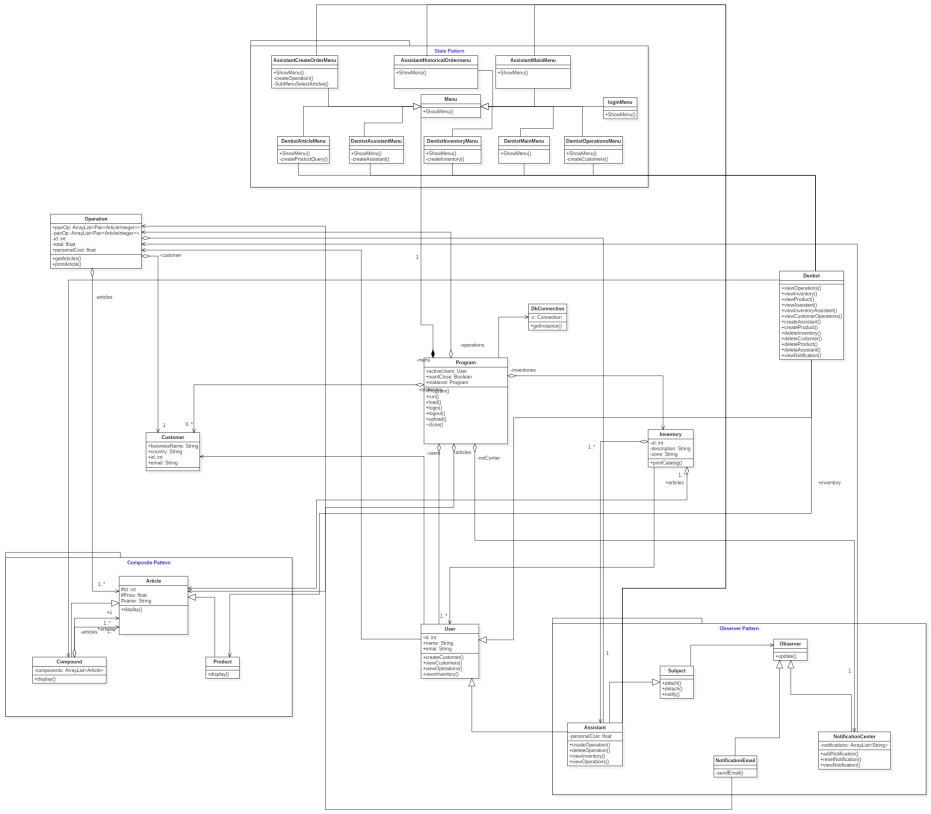


Figure 1: Diagramma delle classi

2.1.1 State Pattern

Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando il comportamento di un oggetto deve cambiare in base al suo stato. Questo pattern è spesso utilizzato per le macchine a stati finiti, il nostro caso è molto simile a quello scenario, infatti il menù passa da uno stato all'altro in base alla scelta dell'utente che lo sta utilizzando.

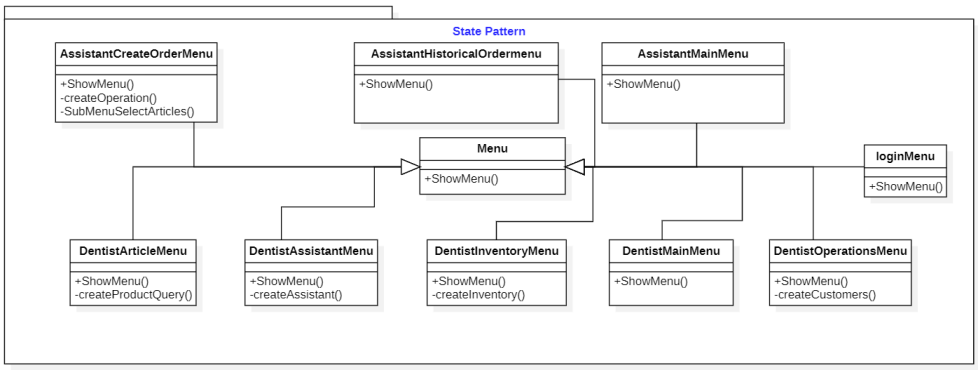


Figure 2: Diagramma delle classi (State pattern)

2.1.2 Observer Pattern

Questo pattern permette di definire una dipendenza 1⟶N fra oggetti, il suo compito è quello di notificare gli N oggetti ogni volta che un oggetto (Subject) cambia stato. Nel progetto sono inseriti due tipologie di Observer: uno che notifica internamente all'applicazione (*NotificationCenter*) mentre l'altro che manda una mail all'assistente desiderato e all'Admin (*NotificationEmail*).

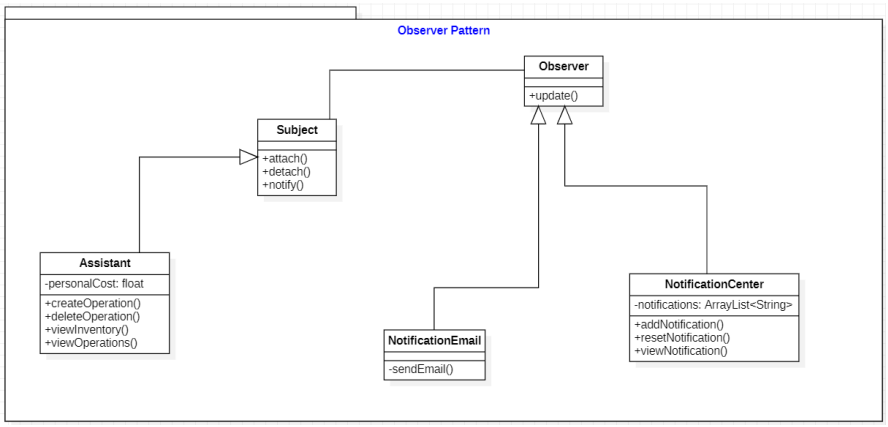


Figure 3: Diagramma delle classi (Observer pattern)

2.1.3 Composite Pattern

Il pattern serve per poter trattare un gruppo di oggetti come istanza di un oggetto singolo. Solitamente questo raggruppamento si può vedere come una struttura ad albero, nel progetto però il pattern è stato leggermente modificato per permettere l'annidamento delle classi composte, in questo caso infatti il grafico del pattern potrebbe essere riassunto con un grafo anzichè un albero.

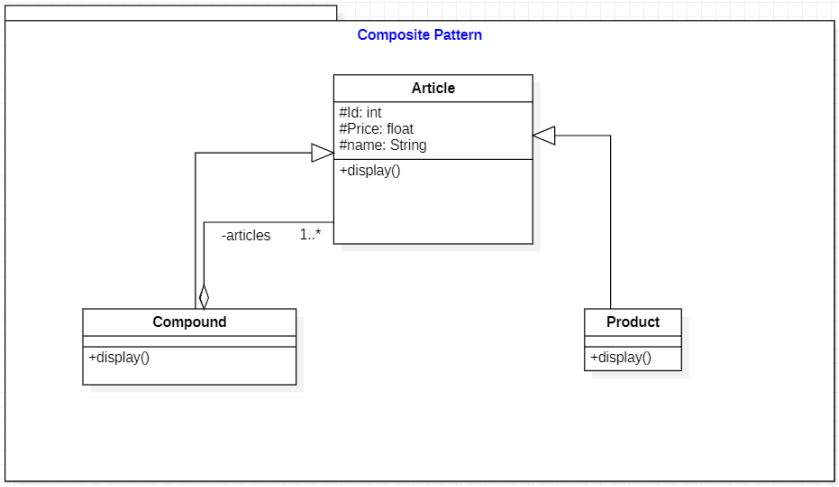


Figure 4: Diagramma delle classi (Composite pattern)

2.2 Use Case Diagram

Nello *Use Case Diagram* sono presenti due attori che interagiscono con il sistema, il Dentista e l'Assistente. I due attori hanno un caso d'uso comune, in quanto entrambi sono classi derivate di User.
Di seguito lo Use Case completo del progetto.

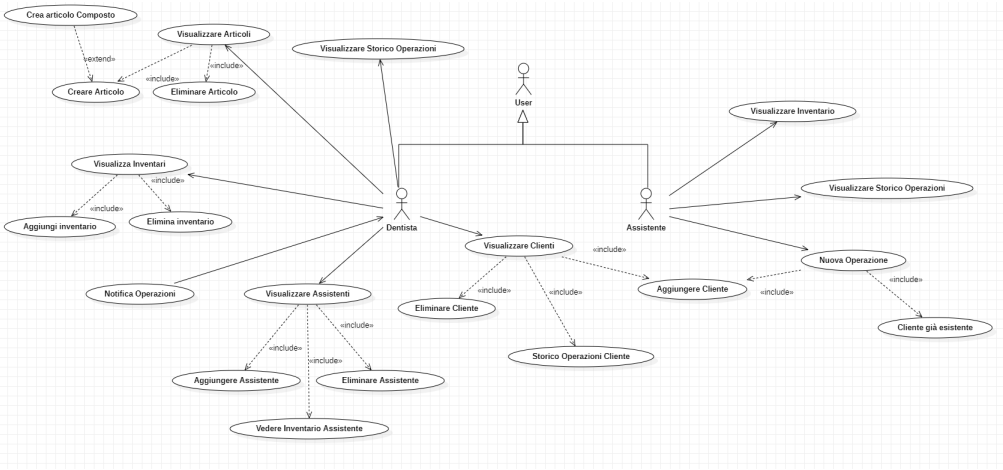


Figure 5: Diagramma dei casi d'uso

2.2.1 Dentist's Use Case

Si distinguono quattro macro gruppi di casi d'uso:

1. ***Articoli***: Il Dentista ha la possibilità di gestire gli articoli, può infatti scegliere se creare, eliminare o semplicemente visualizzare gli Articoli.

2. ***Inventari***: Solo il Dentista ha la possibilità di aggiungere e rimuovere eventuali Inventari contenenti i Set di Operazioni assegnati ai vari assistenti.

3. ***Assistenti***: Il Dentista ha la possibilità di gestire anchegli Assistenti, può infatti scegliere se aggiungere, eliminare o visualizzare gli Inveentari dei vari Assistenti.

4. ***Clienti***: Il Dentista ha la possibilità di gestire i Clienti, può infatti creare, eliminare o visualizzare lo storico delle operazioni avvenute su quel Cliente.

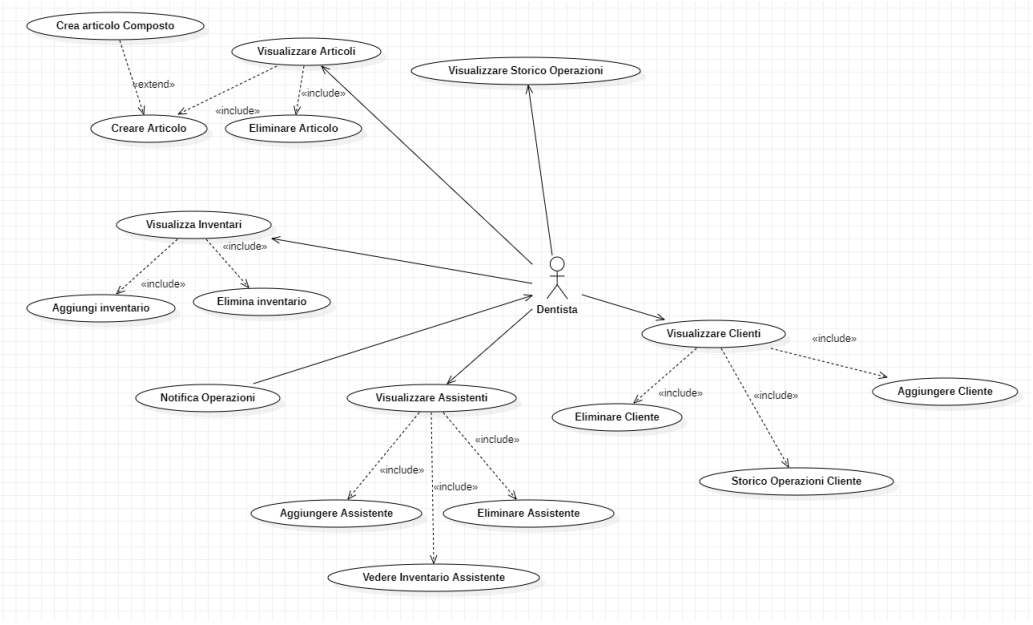


Figure 6: Diagramma dei casi d'uso del Dentista

2.2.2 Assistent's Use Case

Nello *Use Case Diagram* dell'Assistente invece sono presenti meno funzionalità:

1. ***Operazioni***: Tramite la creazione di una nuova operazione è possibile creare o selezionare il cliente alla quale verrà fatto l'intervento. Inoltre è possibile vedere lo storico delle operazioni fatte dall'Assistente stesso.

2. ***Inventario***: L'Assistente può vedere quali operazioni ha nel suo inventario.

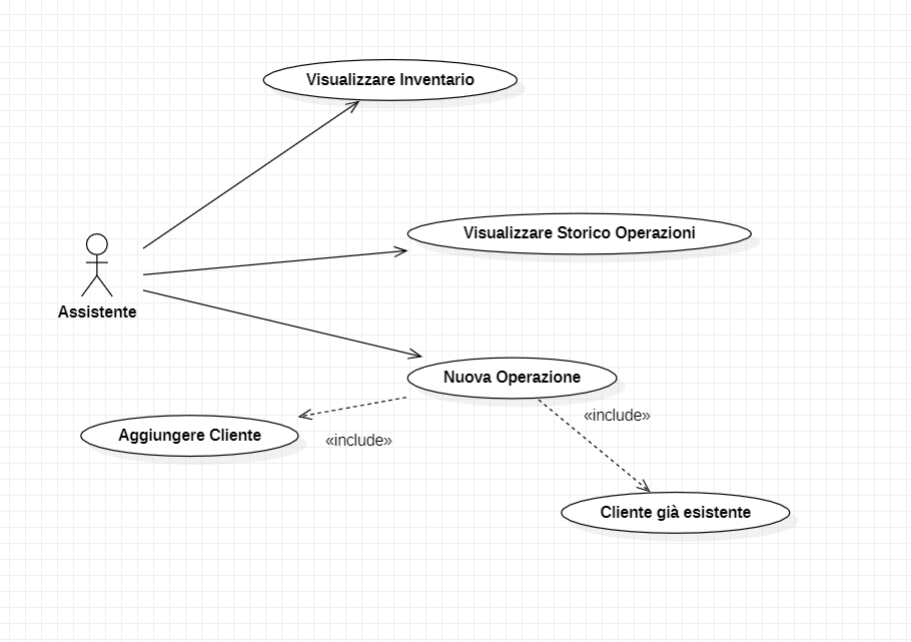


Figure 7: Diagramma dei casi d'uso dell'Assistente

2.3 E/R Diagram

Per avere più chiarezza su come è strutturato il DB abbiamo fatto il diagramma *Entity Relationship* della struttura dati.

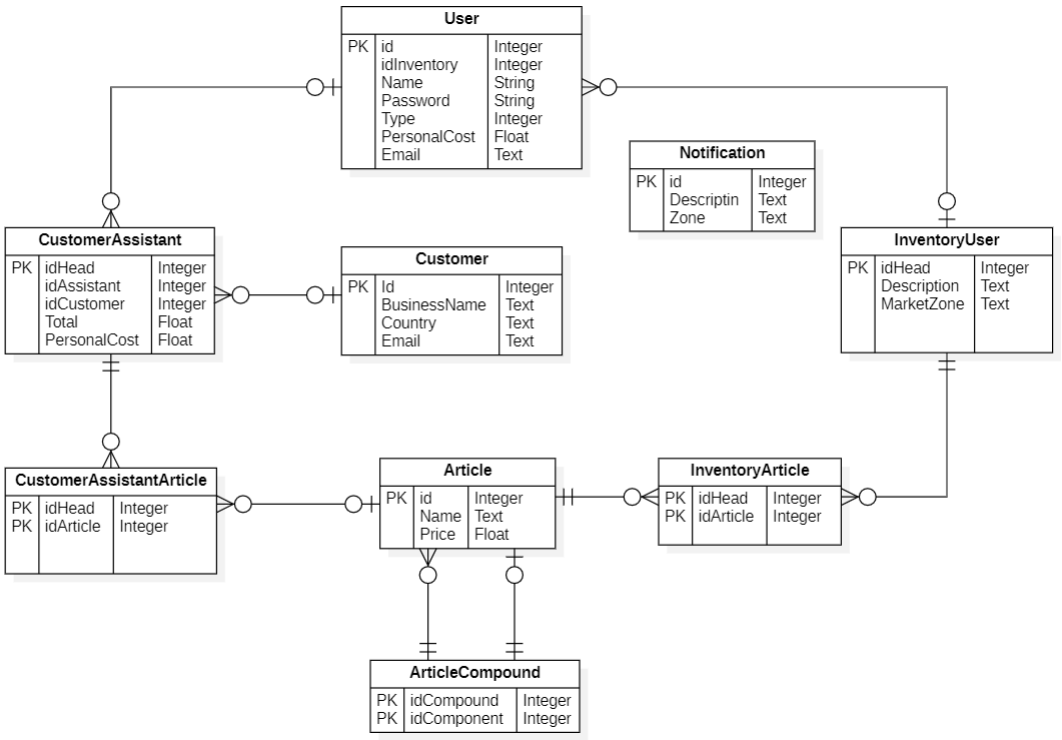


Figure 8: Diagramma E/R della struttura dati

2.3.1 Compound DB Structure

Di seguito riportiamo una scelta implementativa all'interno del database a nostro dire: 'interessante'. Durante la creazione della logica della struttura dati dovevamo creare un modo per archiviare gli articoli di tipo compound. Essendo questi degli articoli composti da articoli che potevano essere aggiunti ai vari inventari dovevamo travare un modo di creare una composizione all'interno del database. Come possiamo notare la tabella article è composta da un id, un nome e un prezzo.

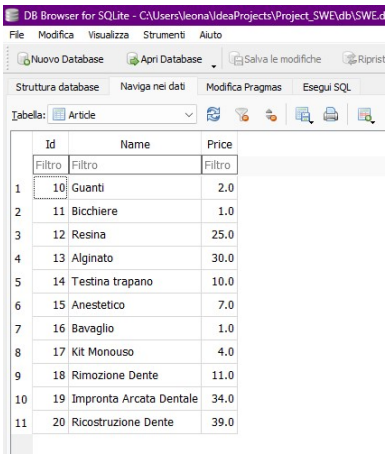


Figure 9: Struttura Article

La tabella ArticleCompound è composta da un idCoumpound e un idComponent. Facendo si che entrambe le colonne siano chiave primaria possiamo 'duplicare' i vari idCompound e associarvi diversi idComponent. Questo stratagemma conclude con l'associazione di vari articoli ad un articolo indipendente.

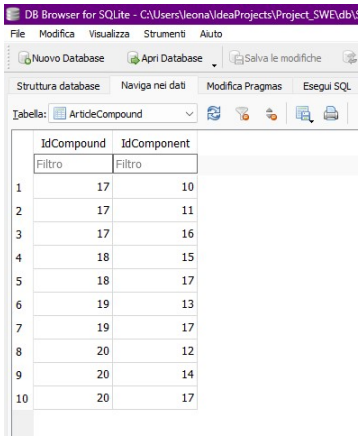


Figure 10: Struttura ArticleCompound

3 Implementazione e approfondimento

Di seguito riportiamo alcuni dei più importanti metodi utilizzati che necessitano di una spiegazione più approfondita.

3.1 Observer

Il pattern Observer è utilizzato per notificare sia il dentista che i vari assistenti. Il nostro intento era di fornire all'assistente un resoconto dell'operazione con i vari articoli utilizzati e l'importo dei materiali. Per il dentista invece funge da 'allarme' non appena un assistente dichiara di aver utilizzato alcuni articoli. Più nello specifico il nostro subject non è altro che l'assistente in quanto è responsabile di effettuare le operazioni. Durante il login gli observer sono legati con il metodo *attach()* e al momento del logout viene chiamato il metodo *detach()*. Il pattern observer è implementato attraverso due interfacce ovvero il NotificationEmail e il NotificationCenter. La prima è responsabile di inviare una mail ai diretti interessati col resoconto delle operazioni. L'altra invece è responsabile di notificare il dentista alla sua entrata con le operazioni effettuate in sua assenza.

```
1  public final class NotificationCenter implements Observer {
2
3      private ArrayList<String> notification;
4
5   @Override
6      public void update(Object obj) {
7          Operation operation = (Operation)obj;
8          this.notification.add("Una nuova operazione per " +
               operation.getCustomer().getBusinessName() + "  stata fatta da
               " + operation.getAssistant().getName());
9      }
10
11 }
```

Listing 1: Observer

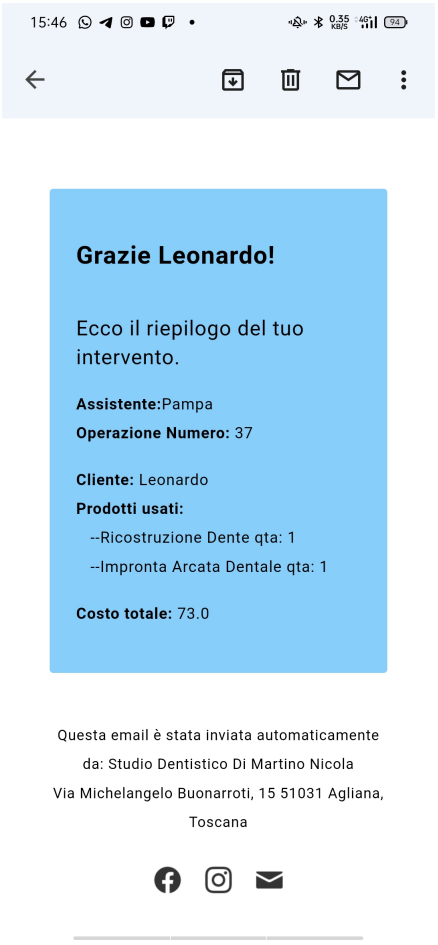
3.1.1 Email Observer

```
1  public final class NotificationEmail implements Observer {
2
3      @Override
4      public void update(Object obj) {
5          Operation o = (Operation) obj;
6          String to = "";
7          for (User u : Program.getInstance().getUsers()) {
8              if (u instanceof Dentist)
9                  to += u.getEmail() + ",";
10         }
11         to = to.substring(0, to.length() - 1); //TODO mettere email
               dell'admin
12
```

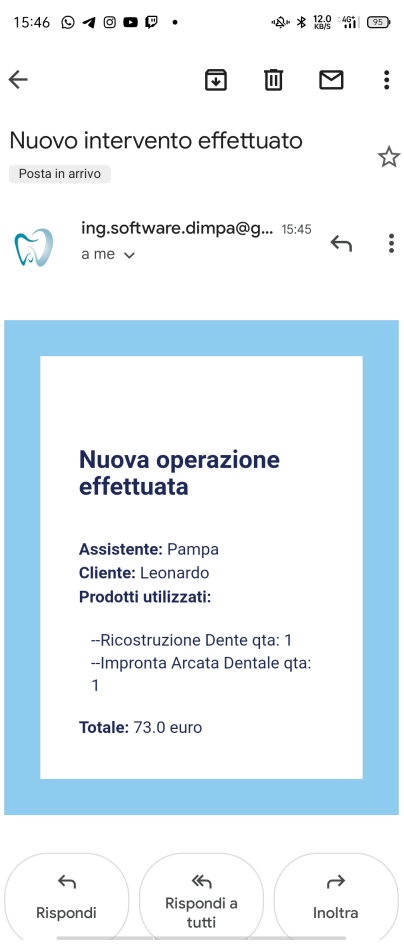
```
13          String products = "";
14          for (Pair<Article, Integer> a : o.getRows()) {
15              products += "--" + a.getValue0().getName() + " qta: " +
                    a.getValue1() + " <br>"; //TODO WOWOWOWOW
16          }
17
18          String text;
19          text = " ... ";
20
21  private void sendEmail(String to, String obj, String text) {
22
23          String test = "pippodima99@gmail.com";
24          String from = "ing.software.dimpa@gmail.com";
25
26          Properties properties = System.getProperties();
27          properties.put("mail.smtp.host", "smtp.gmail.com");
28          properties.put("mail.smtp.port", "465");
29          properties.put("mail.smtp.ssl.enable", "true");
30          properties.put("mail.smtp.auth", "true");
31
32          Session session = Session.getInstance(properties, new
                javax.mail.Authenticator() {
33              protected PasswordAuthentication getPasswordAuthentication() {
34                  return new
                        PasswordAuthentication("ing.software.dimpa@gmail.com",
                        "rkqvtlxwtcaczfjj\n"); //mbvmolrwfpmzcuoq
                        neauczeusvreoesu
35
36              }
37          });
38
39          session.setDebug(true);
40
41          try {
42              MimeMessage message = new MimeMessage(session);
43
44              message.setFrom(new InternetAddress(from));
45              message.addRecipient(Message.RecipientType.TO, new
                    InternetAddress(to));
46              message.setSubject(obj);
47              message.setContent(text, "text/html");
48
49              System.out.println("sending...");
50              Transport.send(message);
51              System.out.println("inviato");
52
53          } catch (MessagingException mex) {
54              mex.printStackTrace();
55          }
56      }
57
58  }
```

Listing 2: Email Observer

Una volta che il sistema registra le nuove operazioni queste sono le email inoltrate automaticamente dal sistema.



(a) Email inoltrata all'assistente.



(b) Email inoltrata al dentista.

3.1.2 Cursiosita'

Durante le numerose prove di testing per il funzionamento del sistema di notifica e-mail, siamo andati incontro a un imprevisto. Avendo disattivato tutte le possibili protezioni sull'account per poterci accedere dall'applicazione, pensiamo che il fatto di aver fatto tante prove in poco tempo abbia "attirato l'attenzione" di persone di terze parti che sono entrati nell'account o hanno fatto entrare l'indirizzo email in loop di promozioni e spam vari. Ce ne siamo accorti quando nel lanciare il metodo per mandare l'email, nel terminale era notificato un errore per aver "finito" le mail giornaliere da poter mandare. Una volta andati sul nostro indirizzo email abbiamo visto che sono state inviate c.a. 300 mail in un minuto, con un testo scritto in arabo a proposito di un costo di spedizione per qualche ordine. Ancora oggi non sappiamo cosa sia successo ma ci faceva piacere condividere questa strana esperienza che ci capitata.

Di seguito un esempio delle tante mail arrivate:

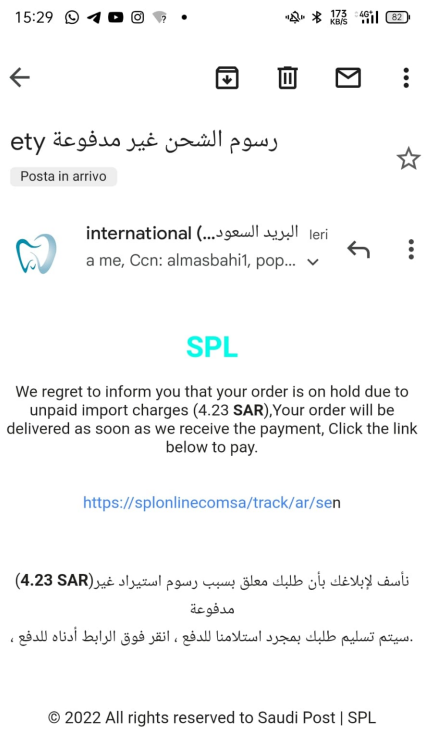


Figure 12: Email inoltrata al dentista.

3.2 User

La classe User dichiarata astratta in modo che possa fornire una base di partenza per tutti i tipi di utenti previsti e non, nel nostro programma. La classe astratta presenta metodi base come il *viewOperations()* e il *viewInventory()* che sono comuni alle classi derivate di utente (*Assistente* e *Dentista*). Questi due metodi sono implementati per in due modi diversi in quanto un assistente pu visualizzare le operazioni fatte soltanto da se stesso e non quelle degli altri. Il dentista invece in grado di vedere tutte le operazioni in quanto '*Admin*' del sistema. La classe User possiede inoltre un attributo comune quale la password in quanto ogni utente per loggare nel programma avr bisogno delle sue credenziali d'accesso.

3.2.1 Assistant

Come in un vero studio dentistico gli assistenti sono in grado di effettuare le proprie operazioni supervisionati dal dentista effettivo. Sono in grado di aggiungere i clienti, effettuare le operazioni all'interno dell'inventario a loro associato. Sonon inoltre in grado di eliminare clienti nel caso in cui questi vogliano cambiare studio dentistico. Quando gli assistenti effettuano un'operazione l'observer notifica attraverso il sistema di mail sia lui stesso con il resoconto degli articoli utilizzati sia il dentista.

```
1  public void createOperation(Customer c,
       ArrayList<Pair<Article,Integer>> articles) {
2          Operation operation = new Operation(this,articles,c);
3          Program.getInstance().getOperations().add(operation);
4          System.out.println("Creato!");
5          notify(new Operation(operation));
6      }
```

Listing 3: Crezione operazione

```
1    public void notify(Object obj) {
2          for(Observer o: observers)
3              o.update(obj);
4      }
```

Listing 4: Notify

3.2.2 Dentist

Il dentista, ovvero l'Admin del sistema in grado di visualizzare tutte le informazioni di tutte le classi. E' inoltre il responsabile della creazione di nuovi inventari, aggiunta di articoli e articoli composti. Pu inoltre cancellare qualsiasi elemento del sistema, persino gli assistenti. Quando assume un nuovo assistente deve associare una mail esistente in modo che il sistema di notifica possa

funzionare correttamente.

```
1  public void createAssistant(String name, String password, float
       commission, Inventory inventory, String email) {
2      Program.getInstance().getUsers().add(new
           Assistant(name,password,commission, inventory,email));
3      System.out.println("Creato!");
4  }
```

Listing 5: Crezione assistente

Come vediamo nel seguente esempio pu vedere tutte le operazioni di uno specifico cliente.

```
1  public void viewCustomerOperations(int idCustomer){
2
3      System.out.println("--------------------------------------");
4      boolean check = false;
5      for(Operation i : Program.getInstance().getOperations()){
6          if
               (i.getAssistant()!=null&&i.getAssistant().getId()==idCustomer)
               {
7              System.out.println("Operazione -> ID: " + i.getId() + "
                   TOTALE: " + i.getTotal() + " COSTO PERSONALE:
                   " + i.getOPersonalCost() + " CLIENTE: " +
                   i.getCustomer().getBusinessName());
8              i.printArticle();
9              check = true;
10         }
11     }
12     if(!check)
13         System.out.println("Non ci sono ordini!.");
14     System.out.println("--------------------------------------");
15
16 }
```

Listing 6: Controllo operazioni di un cliente specifico

```
1  public void deleteAssistant(int idAgent){
2      Assistant assistant =null;
3      for(User i : Program.getInstance().getUsers()){
4          if (i instanceof Assistant && i.getId()==idAgent){
5              assistant = (Assistant) i;
6              break;
7          }
8      }
```

Listing 7: Eliminazione di un assistente

Durante l'eliminazione di un relativo inventario ci siamo trovati davanti ad una difficile scelta implementativa. La problematica era che eliminando un inventario si eliminava pure lo storico delle operazioni di un assistente che ha

attinto a quei vari articoli. Inizialmente avevamo creato dei controlli affinch non si potesse eliminare un inventario se un assistente avesse usato delle componenti proprie. Successivamente per, anche da un confronto con la realt, abbiamo deciso di ignorare il problema perch se scegliamo di eliminare un determinato inventario perch decidiamo di effettuare una ristrutturazione della gestione delle risorse in quanto possono cambiare prezzi e disponibilit.

```
1  public void deleteInventory(int IdCatalog){
2          Inventory tmp = null;
3
4
5
6          for(Inventory i: Program.getInstance().getInventories()){
7              if(i.getId()==IdCatalog){
8                  tmp = i;
9              }
10         }
11
12         if (tmp == null){
13             System.err.println("ID sbagliato! Riprovare");
14             return;
15         }
16
17         Program.getInstance().getInventories().remove(tmp);
18         System.out.println("Cancellato!");
19     }
```

Listing 8: Eliminazione di un inventario

Listing 9: Controllo operazioni di un cliente specifico

Listing 10: Controllo operazioni di un cliente specifico

3.3 Program

La classe Program responsabile del download e upload dei dati attraverso la funzione *load(Connection c)*, i dati vengono recuperati attraverso delle query al database indicato nella connection e caricati allinterno degli ArrayList della classe. Durante questa fase di sviluppo, StackOverflow si rivelato il nostro migliore amico.

```
1  public void load(Connection c) throws SQLException {
2
3          Statement stmt = c.createStatement();
4          Statement stmt1 = c.createStatement();
5          ResultSet rs, rs1;
6
7          rs = stmt.executeQuery("SELECT * FROM Customer;");
```

```
8           while (rs.next()) {
9               int id = rs.getInt("id");
10              String businessName = rs.getString("BusinessName");
11              String country = rs.getString("Country");
12              String email = rs.getString("Email");
13              customers.add(new Customer(id, businessName, country, email));
14          }
15
16          rs = stmt.executeQuery("SELECT * FROM Notification;");
17          while (rs.next()) {
18              String message = rs.getString("message");
19              notCenter.addNotification(message);
20          }
21
22          rs = stmt.executeQuery("SELECT * FROM Article WHERE id not
                in (SELECT IdCompound FROM ArticleCompound );");
23          while (rs.next()) {
24              int id = rs.getInt("id");
25              String name = rs.getString("name");
26              float price = rs.getFloat("Price");
27              articles.add(new Product(name, price, id));
28          }
29          rs = stmt.executeQuery("SELECT * FROM Article WHERE id in
                (SELECT IdCompound FROM ArticleCompound );");
30          while (rs.next()) {
31              int id = rs.getInt("id");
32              String name = rs.getString("name");
33              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 InventoryUser;");
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 InventoryArticle
                    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          inventories.add(new Inventory(tmp, description, marketZone,
                id));
65      }
66
67      rs = stmt.executeQuery("SELECT * FROM User;");
68      while (rs.next()) {
69          int id = rs.getInt("id");                                   //1 agent
                - 0 administrator
70          String name = rs.getString("Name");
71          String passHash = rs.getString("Password");
72          int type = rs.getInt("Type");
73          int idInventory = rs.getInt("IdInventory");
74          float personalCost = rs.getFloat("PersonalCost");
75          String email = rs.getString("email");
76
77          if (type == 1) {
78              Inventory tmp = null;
79              for (Inventory i : inventories) {
80                  if (i.getId() == idInventory) {
81                      tmp = i;
82                  }
83              }
84
85              if (tmp == null) {
86                  System.err.println("Inventory don't exist!");
87                  break;
88              }
89
90              users.add(new Assistant(name, passHash, personalCost,
                    tmp, email, id));
91          } else {
92              users.add(new Dentist(name, passHash, email, id));
93          }
94      }
95
96      rs = stmt.executeQuery("SELECT * FROM CustomerAssistant;");
97      while (rs.next()) {
98          int id = rs.getInt("idHead");
99          int Assistant = rs.getInt("idAssistant");
100         int idCustomers = rs.getInt("IdCustomer");
101         float total = rs.getFloat("Total");
102         float cost = rs.getFloat("PersonalCost");
103
104         Assistant tmpAssistant = null;
105         for (User i : users) {
106             if (i.getId() == Assistant) {
107                 tmpAssistant = (Assistant) i;
108                 break;
109             }
110         }
111
112         Customer tmpCustomer = null;
113         for (Customer i : customers) {
```

```
                    if (i.getId() == idCustomers) {
                        tmpCustomer = i;
                        break;
                    }
                }

                if (tmpCustomer == null) {
                    System.err.println("Customer don't exist!");
                    break;
                }

                ArrayList<Pair<Article, Integer>> tmp = new ArrayList<>();
                rs1 = stmt1.executeQuery("SELECT * FROM
                    CustomerAssistantArticle WHERE IdHead = " + id + " ;");
                while (rs1.next()) {
                    int idArticle = rs1.getInt("idArticle");
                    int qta = rs1.getInt("qta");
                    for (Article a : articles) {
                        if (a.getId() == idArticle) {
                            tmp.add(new Pair<>(a, qta));
                            break;
                        }
                    }
                }
                operations.add(new Operation(total, cost, tmpAssistant, tmp,
                    tmpCustomer, id));
            }

        }
```

Listing 11: Implementazione Load

Analogamente abbiamo la parte che consente l'operazione opposta che elimina tutte le entry nel DB e inserisce le nuove tuple.

```
public void upload(Connection c) {
        String sql;
        Statement stmt = null;
        try {
            stmt = c.createStatement();
            for (String s : Arrays.asList("DELETE FROM User;",
                "DELETE FROM CustomerAssistant;", "DELETE FROM
                CustomerAssistantArticle;", "DELETE FROM
                Notification;", "DELETE FROM Customer;", "DELETE
                FROM InventoryArticle;", "DELETE FROM
                InventoryUser;", "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);
        }
```

```
15
16          int type;
17          float perch;
18          for (User user : users) {
19              try {
20                  if (!(user instanceof Assistant)) {
21                      type = 0;
22                      perch = 0;
23                      sql = "INSERT INTO User
                            (Id,Name,Password,Type,PersonalCost,email) " +
                            "VALUES (" + user.getId() + ", '" +
                            user.getName() + "', '" + user.getPassword() + "',
                            " + type + ", " + perch + " ,'" + user.getEmail()
                            + "');";
24                  } else {
25                      type = 1;
26                      Assistant tmp = (Assistant) user;
27                      perch = tmp.getPersonalCost();
28                      sql = "INSERT INTO User
                            (Id,Name,Password,Type,PersonalCost,idInventory,email)
                            " + "VALUES (" + user.getId() + ", '" +
                            user.getName() + "', '" + user.getPassword() + "',
                            " + type + ", " + perch + " ," +
                            tmp.getInventory().getId() + " ,'" + user.getEmail()
                            + "');";
29                  }
30
31                  stmt = c.createStatement();
32                  stmt.executeUpdate(sql);
33                  c.commit();
34              } catch (Exception e) {
35                  System.err.println(e.getClass().getName() + ": " +
                        e.getMessage());
36              }
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 (Operation operation : operations) {
51              try {
52                  if (operation.getAssistant() != null)
53                      sql = "INSERT INTO CustomerAssistant
```

```
                        (idHead,idAssistant,IdCustomer,Total,PersonalCost)
                        " + "VALUES (" + operation.getId() + ", '" +
                        operation.getAssistant().getId() + "', " +
                        operation.getCustomer().getId() + " ,'" +
                        operation.getTotal() + "', '" +
                        operation.getOPersonalCost() + "');";
54                  else
55                      sql = "INSERT INTO CustomerAssistant
                        (idHead,idAssistant,IdCustomer,Total,PersonalCost)
                        " + "VALUES (" + operation.getId() + ", '" + -1
                        + "', " + operation.getCustomer().getId() + " ,'" +
                        operation.getTotal() + "', '" +
                        operation.getOPersonalCost() + "');";
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 : operation.getRows()) {
64                      sql = "INSERT INTO CustomerAssistantArticle
                            (idHead,idArticle,qta) " + "VALUES (" +
                            operation.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 (Inventory inventory : inventories) {
75              try {
76                  sql = "INSERT INTO InventoryUser
                            (idHead,Description,MarketZone) " + "VALUES (" +
                            inventory.getId() + ", '" + inventory.getDescription() +
                            "', '" + inventory.getZone() + "');";
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 : inventory.getArticles()) {
85                      sql = "INSERT INTO InventoryArticle
                            (idHead,idArticle) " + "VALUES (" +
                            inventory.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 12: Implementazione Upload

Il metodo *Run()* rappresenta il loop del sistema. Instaura la connessione col Database tramite la DBConnection, chiede i dati per il login ed effettua il ciclo sulla variabile booleana *WantClose()* da cui si chiama lo *ShowMenu()*. Quando si desidera uscire dal programma questa variabile verr messa a true interrompendo il loop.

Listing 13: Implementazione Run()

4 UnitTest

Prendiamo adesso in analisi la parte dei test effettuati per verificare la correttezza del codice. La base dati utilizzata semplicemente una copia di quella usata nel programma principale. I test effettuati ricoprono la parte di creazione ed eliminazione di assistenti, inventari, articoli e clienti. La funzione *prepare()* collega il database tramite la DBConnectionTest tramite la libreria Junit5.

4.1 Assistant test

Vediamo il funzionamento dei test per la classe *Assistant*.

4.1.1 Creazione/eliminazione di un'operazione

Verifichiamo il corretto funzionamento di creazione di una operazione da parte di un assistente. Creiamo una lista di articoli con la loro relativa quantit e viene creato anche un cliente test a cui assegnare tale operazione. Verifichiamo poi che l'ordine sia stato effettuato correttamente.

```
1  @Test
2      @DisplayName("Create Operation Test")
3      void testOperationCreation() {
4
5          ArrayList<Pair<Article, Integer>> articles = new ArrayList<>();
6
7          articles .add(new
               Pair<>(assistant.getInventory().getArticles().get(1), 20));
8          articles .add(new
               Pair<>(assistant.getInventory().getArticles().get(2), 50));
9
10         Customer customer = p.getCustomers().get(2);
```

```
11
12          assistant.createOperation(customer, articles);
13
14          Operation createdOperation =
                p.getOperations().get(p.getOperations().size() - 1);
15
16
17          assertAll("Order's Data",
18                  () -> assertEquals(createdOperation.getAssistant(),
                        assistant),
19                  () -> assertEquals(createdOperation.getCustomer(),
                        customer),
20                  () -> assertEquals(createdOperation.getArticles().get(0),
                        articles.get(0).getValue0()),
21                  () -> assertEquals(createdOperation.getArticles().get(1),
                        articles.get(1).getValue0())
22          );
23
24      }
```

Listing 14: Implementazione creazione operazione

Analogamente adesso vogliamo che l'operazione appena creata venga rimossa dal database.

```
1       @DisplayName("Delete Operation Test")
2       void testDeleteOperation() {
3
4           Operation operation = p.getOperations().get(0);
5
6           for (Operation i : p.getOperations()) {
7               if (i.getAssistant() == assistant) {
8                   operation = i;
9               }
10          }
11
12          int operationCountBefore1 = p.getOperations().size();
13
14          assistant.deleteOperation(operation.getId());
15
16          Operation finalOp = operation;
17          assertAll("Operation deleted",
18                  () -> assertTrue(p.getOperations().size() <
                        operationCountBefore1),
19                  () -> assertFalse(p.getOperations().contains(finalOp))
20          );
21
22      }
```

Listing 15: Implementazione eliminazione operazione

4.2 Denstist test

Controlliamo adesso il corretto funzionamento della classe dentista, ovvero il nostro amministratore.

4.2.1 Creazione/eliminazione di un assistente

Inseriamo manualmente un assistente e verifichiamo che effettivamente esista.

```
1   @Test
2     @DisplayName("Create AssistantTest")
3     void testCreateAssistant() {
4
5         Inventory inventory = p.getInventories().get(
              (int)((Math.random() * (p.getInventories().size()-1 - 1)) + 1)
              );
6         admin.createAssistant("UnitTest",
              "studiomartino",5.5F,inventory,"unitTest@gmail.com");
7         User createUser = p.getUsers().get(p.getUsers().size()-1);
8
9         assertTrue(createUser instanceof Assistant);
10
11        Assistant createAssistant = (Assistant) createUser;
12
13        assertAll("Test create assistant",
14                () -> assertEquals(inventory,
                      createAssistant.getInventory()),
15                () -> assertEquals(createAssistant.getId(),
                      p.getUsers().get(p.getUsers().size()-1).getId())
16        );
17
18    }
```

Listing 16: Implementazione creazione assistente

Cancelliamo adesso un assistente e verifichiamo che non sia pi presente. Abbiamo aggiunto un controllo che ci permette di controllare che all'eliminazione dell'assistente anche il suo inventario associato non sia stato cancellato.

```
1   @Test
2     @DisplayName("Delete AssistantTest")
3     void testDeleteAssistant() {
4
5         int id = 0;
6         int id2 = 3;
7
8         for (User u : p.getUsers()) {
9             if (u.getId() == id) {
10                admin.deleteAssistant(id);
11            }
12        }
13
14
15        assertFalse(p.getUsers().contains(id));
16
17
18    }
19
20    private boolean checkInventory(int id){
21        for (Inventory t : p.getInventories()){
22            if(t.getId()==id){
23                for (User u : p.getUsers()){
```

```
                    if(u instanceof Assistant){
                        if(((Assistant)u).getInventory().equals(t)){
                            return true;
                        }
                    }
                }
            }
        }
        return false;
    }
```

Listing 17: Implementazione eliminazione assistente

4.2.2 Creazione/eliminazione inventario

Creiamo manualmente un inventario con relativi articoli e controlliamo che sia stato effettivamente inserito.

```
@Test
    @DisplayName("Create InventoryTest")
    void testCreateInventory() {

        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.getInventories().size();
        admin.createInventory("description","Italy", articles);

        assertAll("Test create assistant",
                () -> assertEquals(preSize + 1, p.getInventories().size()),
                () -> assertEquals(articles,
                    p.getInventories().get(p.getInventories().size()-1).getArticles())
        );

    }
```

Listing 18: Implementazione creazione inventario

Adesso lo eliminiamo e verifichiamo che non esista pi.

```
@Test
    @DisplayName("Delete InventoryTest")
    void testDeleteInventory() {

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

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

```
14
15          int lastCat =
                p.getInventories().get(p.getInventories().size()-1).getId();
16          check = checkInventory(lastCat);
17
18          admin.deleteInventory(lastCat);
19
20          if(check)
21              assertEquals(preSize - 1, p.getInventories().size());
22          else
23              assertEquals(preSize - 1, p.getInventories().size());
24
25      }
```

Listing 19: Implementazione eliminazione inventario

4.2.3 Creazione/Eliminazione di un prodotto

Aggiungiamo adesso prima un semplice articolo e verifichiamo che sia stato inserito, poi creiamo un articolo di tipo composto e controlliamo che sia stato generato correttamente.

```
1   @Test
2       @DisplayName("Create ProductTest")
3       void testCreateProduct() {
4           int preSize = p.getArticles().size();
5           admin.createProduct("ProductTestSingle",3.5F);
6
7           assertAll("Single Article",
8                   () -> assertEquals(preSize + 1, p.getArticles().size()),
9                   () ->
                        assertTrue(p.getArticles().get(p.getArticles().size()-1)
                        instanceof Product),
10                  () ->
                        assertEquals(p.getArticles().get(p.getArticles().size()-1).getPrice(),
                        3.5F)
11          );
12
13          int preSize2 = p.getArticles().size();
14          ArrayList<Article> articles = new ArrayList<>();
15          articles.add(p.getArticles().get(1));
16          articles.add(p.getArticles().get(2));
17
18          float tmp = 0;
19          for (Article a : articles)
20              tmp += a.getPrice();
21          float prePrice = tmp;
22
23          admin.createProduct("Compound article",articles);
24
25          assertAll("Compound Article",
26                  () -> assertEquals(preSize2 + 1, p.getArticles().size()),
27                  () ->
                        assertTrue(p.getArticles().get(p.getArticles().size()-1)
                        instanceof Compound),
```

```
28                  () ->
                        assertEquals(p.getArticles().get(p.getArticles().size()-1).getPrice(),
                        prePrice)
29
30          );
31
32      }
```

Listing 20: Implementazione creazione prodotti

Analogamente eliminiamo i prodotti. Controlliamo inoltre che non possa cancellarsi un articolo che presente in un articolo composto.

```
1   @Test
2       @DisplayName("Delete ArticleTest")
3       void testDeleteArticle() {
4
5           admin.createProduct("testProduct1 - can_delete",5.5F);
6           Article P1 = p.getArticles().get(p.getArticles().size()-1);
7           int A1 = p.getArticles().get(p.getArticles().size()-1).getId();
8           admin.deleteProduct(A1);
9
10          assertFalse(p.getArticles().contains(P1));
11
12          int A2 = 1;
13          Article P2 = null;
14
15          for(Article a : p.getArticles()){
16              if(a.getId()==1) {
17                  P2 = a;
18              }
19          }
20          admin.deleteProduct(A2);
21          assertFalse(p.getArticles().contains(P2));
22
23          ArrayList<Article> articles = new ArrayList<>();
24          articles.add(p.getArticles().get(2));
25          articles.add(p.getArticles().get(3));
26          admin.createProduct("testProduct2", articles);
27          Article P3 = p.getArticles().get(2);
28          int A3 = p.getArticles().get(2).getId();
29          admin.deleteProduct(A3);
30
31          assertFalse(p.getArticles().contains(P3));
32
33      }
```

Listing 21: Implementazione eliminazione prodotti

4.2.4 Eliminazione cliente

Verifichiamo che possa essere correttamente rimosso un cliente dello studio.

```
1   @Test
2       @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));

    }
```

Listing 22: Implementazione eliminazione cliente

4.3 CoreTest

Controlliamo che le connessioni tra database si effettuino correttamente.

4.3.1 Login test

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

        p.login("Pampa", "studiomartino");

        User user = null;

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

        User expectedUser1 = user;

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

        p.logout();

        p.login("Nicola", "ilmiostudio");

        user = null;

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

```
        }

        User expectedUser2 = user;

        assertAll("Dentist Login",
                () -> assertEquals(expectedUser2, p.getActiveUser()),
                () -> assertTrue(p.getActiveUser() instanceof Dentist)
        );
    }
```

Listing 23: Implementazione login

4.4 Test Load/Upload

Controlliamo adesso che il collegamento tra database funzioni e che *Upload* e *Load*siano effettive. Si utilizzano delle query manuali dove inseriamo alcuni personaggi di Topolino come test. Attraverso il metodo *load* si verifica che i nuovi dati siano stati inseriti correttamente. Inseriamo poi alcuni articoli in modo che si possa chiamare il metodo *upload(Connection)*

```
@Test
    @DisplayName("Upload/Load data Test")
    void testUploadLoadData() throws SQLException {
        String sql;
        Statement stmt;
        ResultSet rs;

        Connection c = DBConnectionTest.getInstance();
        p.upload(c);
        p = Program.getInstance();

        c = DBConnectionTest.getInstance();

        String customerName = "Paperino";
        String customerCountry = "Paperinolandia";
        String customerEmail = "paperino.senzacognome@gmail.com";

        sql = "INSERT INTO Customer (BusinessName,Country,Email) "
              + "VALUES ('" + customerName + "', '" + customerCountry
              + "', '" + customerEmail + "');";

        stmt = c.createStatement();
        stmt.executeUpdate(sql);
        c.commit();

        String articleName = "Penny";
        float  articlePrice  = 10.2F;

        sql = "INSERT INTO Article (Name,Price) " + "VALUES ('" +
              articleName + "', '" + articlePrice + "');";
        stmt = c.createStatement();
        stmt.executeUpdate(sql);
        c.commit();

        p.load(c);

```

```
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("Nicola", "ilmiostudio");
57          Dentist admin = (Dentist) p.getActiveUser();
58
59          admin.createProduct("Berretto", 12.2F);
60          admin.createCustomer("Minnie", "paperinolandiaSud",
                "minnie.senzacognome@gmail.com");
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='Berretto';");
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='Minnie' AND
                country='paperinolandiaSud' AND
                email='minnie.senzacognome@gmail.com';");
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 24: Implementazione Load/Upload

5 Conclusione

Siamo partiti con questo progetto circa 1 anno prima della sua conclusione. Durante la sua stesura non avevamo molto chiaro cosa effettivamente fare ma durante questo tempo con ulteriore esperienza abbiamo trovato la via. Inizialmente pensavamo di fare una gestione di un magazzino ma successivamente il mio collega mi ha accennato al fatto che suo padre avesse uno studio dentistico e abbiamo deciso di adattare la struttura gi esistente a questa situazione. Pensavamo di fare un interfaccia grafica affinch+ si potesse navigare nel menu, adesso semplice console, in modo pi 'attuale'. Tuttavia abbiamo dovuto ridimensionare il tutto in quanto ci avrebbe portato via molto tempo ulteriore.