

Dentist management system

Ingegneria del software

Leonardo Pampaloni, Filippo di Martino



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di
Ingegneria

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 operazioni da poter fare, oltre che poter vedere le informazioni sui clienti.
3. **Clienti**
4. **Operazioni**

L'applicazione ha come obiettivo quello di gestire e notificare l'admin, ovvero il dentista, di tutte le operazioni effettuate dagli assistenti, inoltre è compresa nell'applicazione il salvataggio dei clienti, degli articoli e delle operazioni 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.

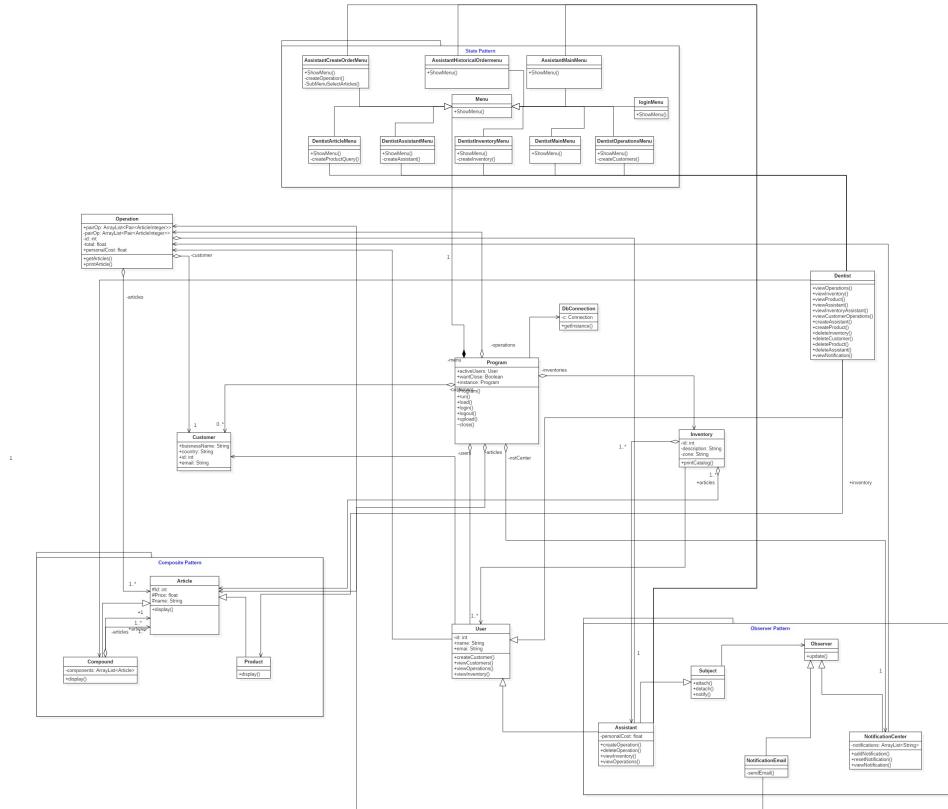


Figure 1: Diagramma delle classi

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.

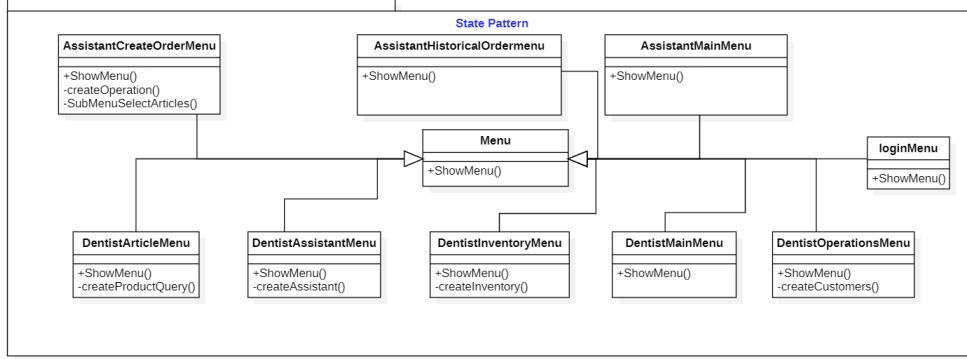


Figure 2: Diagramma delle classi (State pattern)

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*).

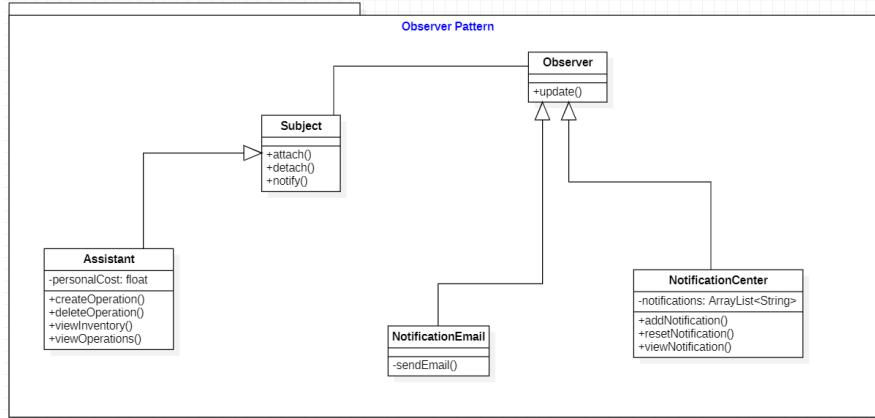


Figure 3: Diagramma delle classi (Observer pattern)

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 invece di un albero.

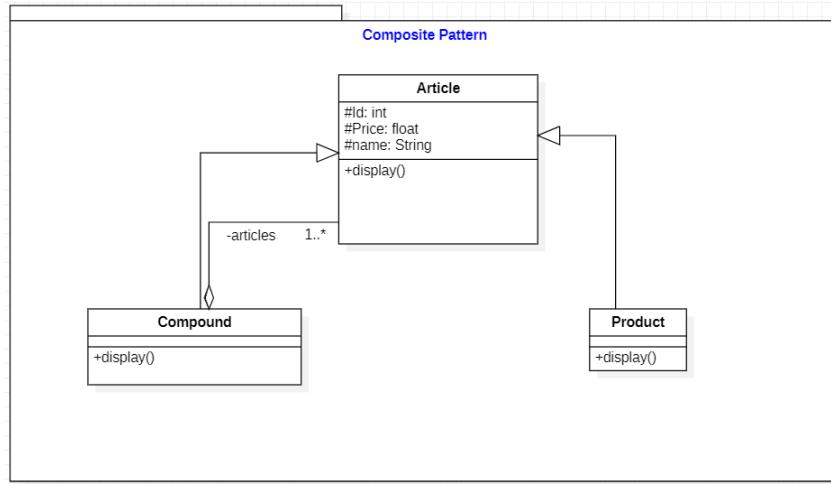


Figure 4: Diagramma delle classi (Composite pattern)

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.

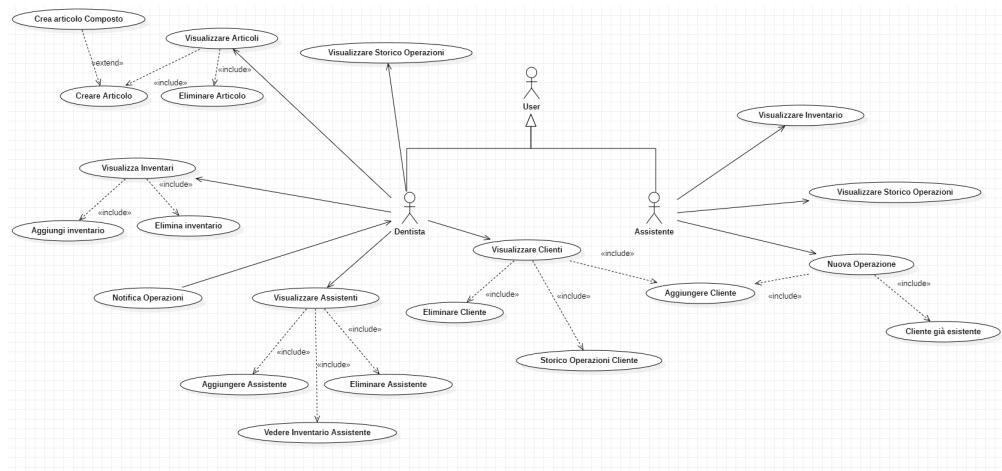


Figure 5: Diagramma dei casi d'uso

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 anche gli Assistenti, può infatti scegliere se aggiungere, eliminare o visualizzare gli Inventari 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.

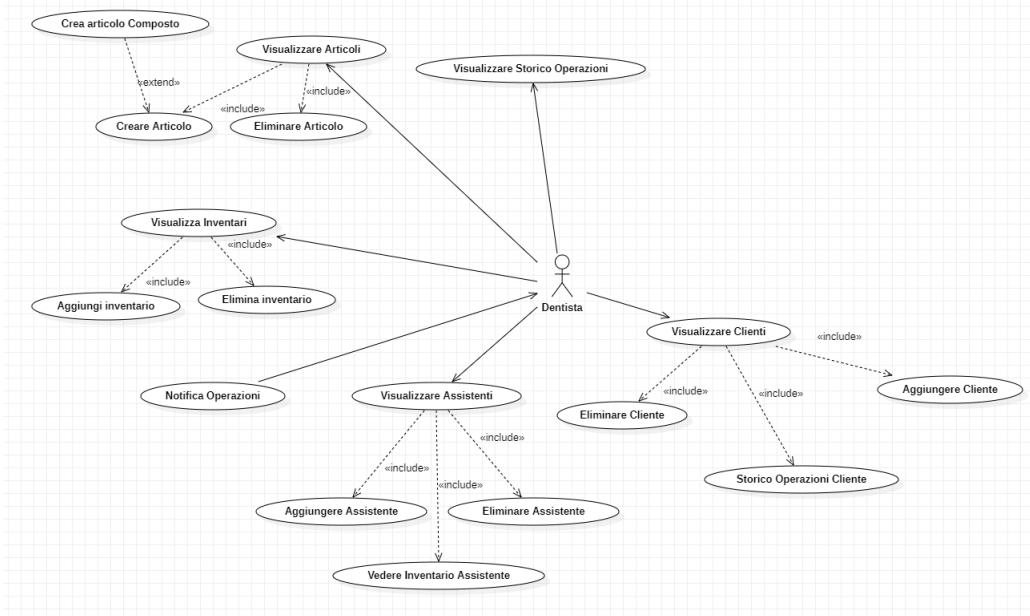


Figure 6: Diagramma dei casi d'uso del Dentista

2.2.2 Assistent's Use Case

Nello Use Case Diagram dell'Assistente invece sono presenti meno funzionalità, questo perchè è nostro intento dare maggior controllo all'admin del sistema, mentre un controllo limitato ai vari assistenti che hanno accesso al programma. Di seguito le seguenti 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.

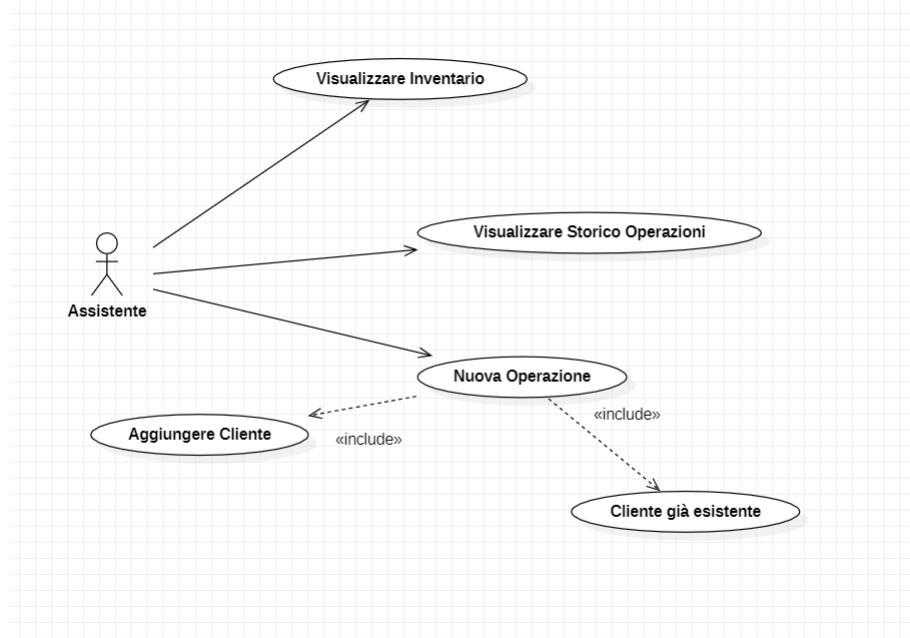


Figure 7: Diagramma dei casi d'uso dell'Assistente

2.3 E/R Diagram

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

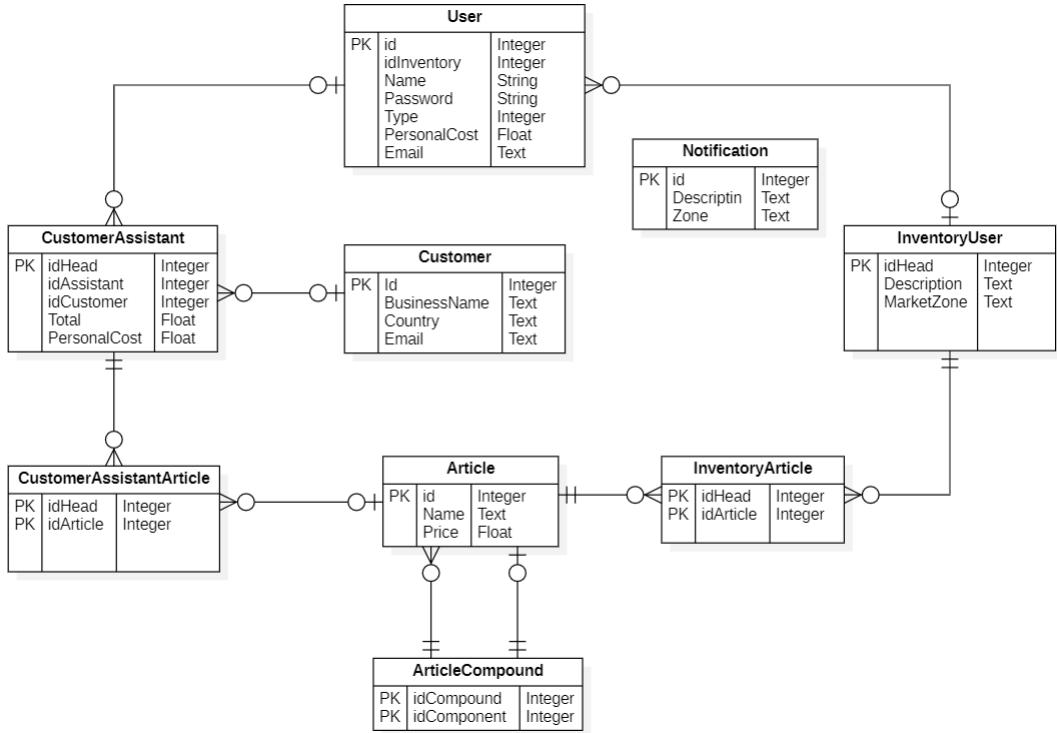


Figure 8: Diagramma E/R della struttura dati

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 trovare 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.

	Id	Name	Price
1	10	Guanti	2.0
2	11	Bicchiere	1.0
3	12	Resina	25.0
4	13	Alginato	30.0
5	14	Testina trapano	10.0
6	15	Anestetico	7.0
7	16	Bavaglio	1.0
8	17	Kit Monouso	4.0
9	18	Rimozione Dente	11.0
10	19	Impronta Arcata Dentale	34.0
11	20	Ricostruzione Dente	39.0

Figure 9: Struttura Article

La tabella ArticleCompound è composta da un idCompound 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.

	IdCompound	IdComponent
1	17	10
2	17	11
3	17	16
4	18	15
5	18	17
6	19	13
7	19	17
8	20	12
9	20	14
10	20	17

Figure 10: Struttura ArticleCompound

3 Implementazione e approfondimento

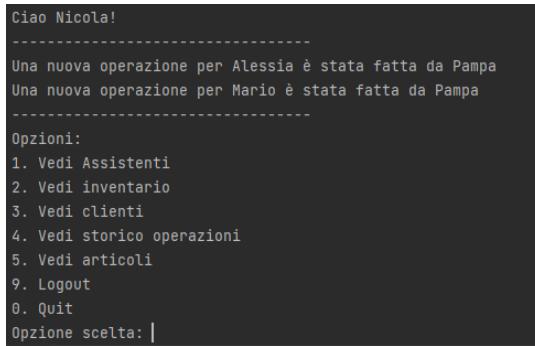
Di seguito riportiamo alcuni dei più importanti metodi utilizzati che necessitano di una spiegazione più approfondita.

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

Listing 1: Observer



The screenshot shows a terminal window with a dark background and white text. It displays a welcome message 'Ciao Nicola!', followed by two notifications: 'Una nuova operazione per Alessia è stata fatta da Pampa' and 'Una nuova operazione per Mario è stata fatta da Pampa'. Below these, there is a menu titled 'Opzioni:' with the following options: 1. Vedi Assistenti, 2. Vedi inventario, 3. Vedi clienti, 4. Vedi storico operazioni, 5. Vedi articoli, 9. Logout, 0. Quit. At the bottom, it says 'Opzione scelta: |'.

Figure 11: Notifica per il dentista all'interno del programma

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);
12
13         String products = "";
14         for (Pair<Article, Integer> a : o.getRows()) {
15             products += " -- " + a.getValue0().getName() + " qta: " +
16                 a.getValue1() + " <br>";
17         }
18
19         String text;
20         text = "... ";
21
22         private void sendEmail(String to, String obj, String text) {
23
24             String test = "pippodima99@gmail.com";
25             String from = "ing.software.dimpa@gmail.com";
26
27             Properties properties = System.getProperties();
28             properties.put("mail.smtp.host", "smtp.gmail.com");
29             properties.put("mail.smtp.port", "465");
30             properties.put("mail.smtp.ssl.enable", "true");
31             properties.put("mail.smtp.auth", "true");
32
33             Session session = Session.getInstance(properties, new
34                 javax.mail.Authenticator() {
35                     protected PasswordAuthentication getPasswordAuthentication() {
36                         return new
37                             PasswordAuthentication("ing.software.dimpa@gmail.com",
38                                 "rkqvtlxwtcaczfjji\n");
39                     }
40                 });
41
42             session.setDebug(true);
43
44             try {
45                 MimeMessage message = new MimeMessage(session);
46                 message.setFrom(new InternetAddress(from));
47                 message.addRecipient(Message.RecipientType.TO, new
48                     InternetAddress(to));
49                 message.setSubject(obj);
50                 message.setContent(text, "text/html");
51
52                 System.out.println("sending ... ");
53                 Transport.send(message);
```

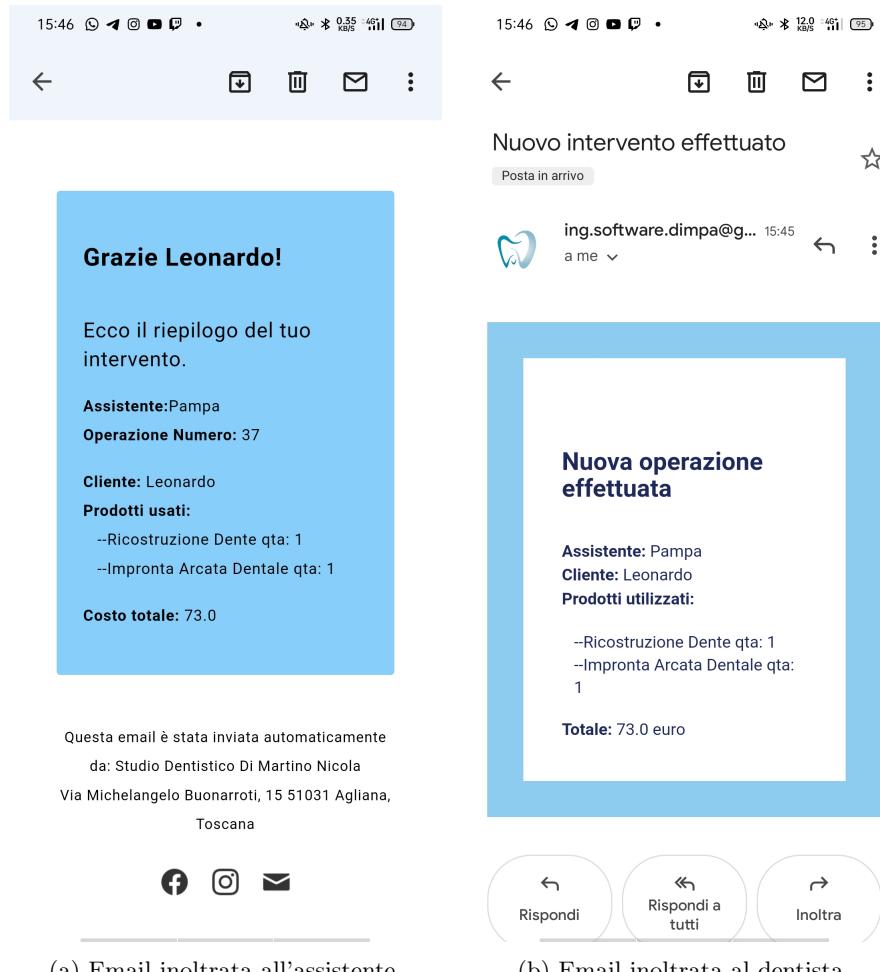
```

51     System.out.println("invia");
52
53 } catch (MessagingException mex) {
54     mex.printStackTrace();
55 }
56
57 }

```

Listing 2: Email Observer

Una volta che il sistema registra le nuove operazioni queste sono le email inviate 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:

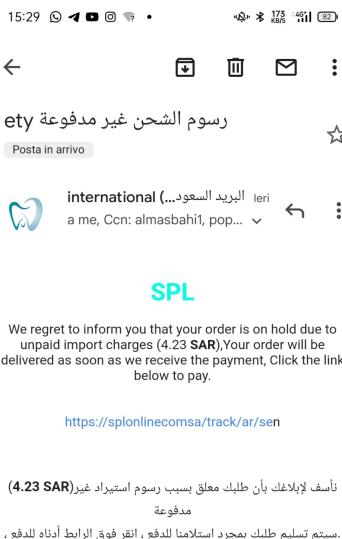


Figure 13: Email hackerate.

3.2 User

La classe User é 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 però 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 é in grado di vedere tutte le operazioni in quanto é '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. Sono in grado di aggiungere i clienti, effettuare le operazioni all'interno dell'inventario a loro associato. Sono 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.

```
Id: 4 Inventario: Inventario Assistente Zona: Firenze
--Id: 18 Composizione: Rimozione Dente Prezzo: 11.0
  ||Componente: Anestetico Prezzo: 7.0
  ||Componente: Kit Monouso Prezzo: 4.0 (Composizione)
--Id: 19 Composizione: Impronta Arcata Dentale Prezzo: 34.0
  ||Componente: Alginato Prezzo: 30.0
  ||Componente: Kit Monouso Prezzo: 4.0 (Composizione)
--Id: 20 Composizione: Ricostruzione Dente Prezzo: 39.0
  ||Componente: Resina Prezzo: 25.0
  ||Componente: Testina trapano Prezzo: 10.0
  ||Componente: Kit Monouso Prezzo: 4.0 (Composizione)

-----
Inserisci un ID di un articolo da assegnare per la sua operazione o 0 per terminare l'operazione
```

Figure 14: Esempio operazione assistente

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

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 è in grado di visualizzare tutte le informazioni di tutte le classi. È inoltre il responsabile della creazione di nuovi inventari, aggiunta di articoli e articoli composti. Può 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 può 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
7              (i.getAssistant() != null & & i.getAssistant().getId() == idCustomer)
8                  {
9                      System.out.println("Operazione -> ID: " + i.getId() +
10                         " TOTALE: " + i.getTotal() + " COSTO PERSONALE:
11                         " + i.getOPersonalCost() + " CLIENTE: " +
12                         i.getCustomer().getBusinessName());
13                     i.printArticle();
14                     check = true;
15                 }
16             }
17         if (!check)
18             System.out.println("Non ci sono ordini!.");
19         System.out.println("-----");
20     }
```

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 affinché non si potesse eliminare un inventario se un assistente avesse usato delle componenti proprie. Successivamente però, anche da un confronto con la realtà, abbiamo deciso di ignorare il problema perché se scegliamo di eliminare un determinato inventario è perché decidiamo di effettuare una ristrutturazione della gestione delle risorse in quanto possono cambiare prezzi e disponibilità.

```

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

```

Listing 8: Eliminazione di un inventario

3.3 Program

La classe Program è 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 all'interno degli ArrayList della classe. Durante questa fase di sviluppo, StackOverflow si è 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
23         in (SELECT IdCompound FROM ArticleCompound );");
24     while (rs.next()) {
25         int id = rs.getInt("id");
26         String name = rs.getString("name");
27         float price = rs.getFloat("Price");
28         articles .add(new Product(name, price, id));
29     }
30     rs = stmt.executeQuery("SELECT * FROM Article WHERE id in
31         (SELECT IdCompound FROM ArticleCompound );");
32     while (rs.next()) {
33         int id = rs.getInt("id");
34         String name = rs.getString("name");
35         ArrayList<Article> components = new ArrayList<>();
36         rs1 = stmt1.executeQuery("SELECT * FROM
37             ArticleCompound WHERE IdCompound = " + id + " ;");
38         while (rs1.next()) {
39             int idComponent = rs1.getInt("idComponent");
40             for (Article a : articles ) {
41                 if (a.getId() == idComponent) {
42                     components.add(a);
43                     break;
44                 }
45             }
46             articles .add(new Compound(name, components, id));
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
55     WHERE IdHead = " + id + " ;");
56     while (rs1.next()) {
57         int idArticle = rs1.getInt("idArticle");
58         for (Article a : articles) {
59             if (a.getId() == idArticle) {
60                 tmp.add(a);
61                 break;
62             }
63         }
64         inventories .add(new Inventory(tmp, description, marketZone,
65                                         id));
66     }
67
68     rs = stmt.executeQuery("SELECT * FROM User;");
69     while (rs.next()) {
70         int id = rs.getInt("id"); //1 agent
71         String name = rs.getString("Name");
72         String passHash = rs.getString("Password");
73         int type = rs.getInt("Type");
74         int idInventory = rs.getInt("IdInventory");
75         float personalCost = rs.getFloat("PersonalCost");
76         String email = rs.getString("email");
77
78         if (type == 1) {
79             Inventory tmp = null;
80             for (Inventory i : inventories) {
81                 if (i.getId() == idInventory) {
82                     tmp = i;
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,
91                               tmp, email, id));
92     } else {
93         users.add(new Dentist(name, passHash, email, id));
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) {
114             if (i.getId() == idCustomers) {
115                 tmpCustomer = i;
116                 break;
117             }
118         }
119         if (tmpCustomer == null) {
120             System.err.println("Customer don't exist!");
121             break;
122         }
123     }
124
125     ArrayList<Pair<Article, Integer>> tmp = new ArrayList<>();
126     rs1 = stmt1.executeQuery("SELECT * FROM
127                             CustomerAssistantArticle WHERE IdHead = " + id + " ;");
128     while (rs1.next()) {
129         int idArticle = rs1.getInt("idArticle");
130         int qta = rs1.getInt("qta");
131         for (Article a : articles) {
132             if (a.getId() == idArticle) {
133                 tmp.add(new Pair<>(a, qta));
134                 break;
135             }
136         }
137         operations.add(new Operation(total, cost, tmpAssistant, tmp,
138                                     tmpCustomer, id));
139     }
140 }
```

Listing 9: Implementazione Load

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

```

1 public void upload(Connection c) {
2     String sql;
3     Statement stmt = null;
4     try {
5         stmt = c.createStatement();
6         for (String s : Arrays.asList("DELETE FROM User;",
7                                       "DELETE FROM CustomerAssistant;", "DELETE FROM
8                                         CustomerAssistantArticle;", "DELETE FROM
9                                         Notification;", "DELETE FROM Customer;", "DELETE
10                                        FROM InventoryArticle;", "DELETE FROM
11                                         InventoryUser;", "DELETE FROM Article;", "DELETE
12                                         FROM ArticleCompound;")) {
13             sql = s;
14             stmt.executeUpdate(sql);
15         }
16     } catch (SQLException e) {
17         e.printStackTrace();
18     }
19 }
```

```

9             c.commit();
10            }
11        } catch (Exception e) {
12            System.err.println(e.getClass().getName() + ":" +
13                e.getMessage());
14            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
24                        (Id,Name,Password,Type,PersonalCost,email) " +
25                            "VALUES (" + user.getId() + ", " +
26                                user.getName() + ", " + user.getPassword() + ",
27                                " + type + ", " + perch + ", " + user.getEmail() +
28                                + ")";}
29            } else {
30                type = 1;
31                Assistant tmp = (Assistant) user;
32                perch = tmp.getPersonalCost();
33                sql = "INSERT INTO User
34                    (Id,Name,Password,Type,PersonalCost,idInventory,email)
35                        " + "VALUES (" + user.getId() + ", " +
36                            user.getName() + ", " + user.getPassword() + ",
37                                " + type + ", " + perch + ", " +
38                                tmp.getInventory().getId() + ", " + user.getEmail() +
39                                + ")";}
40            }
41            stmt = c.createStatement();
42            stmt.executeUpdate(sql);
43            c.commit();
44        } catch (Exception e) {
45            System.err.println(e.getClass().getName() + ":" +
46                e.getMessage());
47        }
48        for (Customer customer : customers) {
49            try {
50                sql = "INSERT INTO Customer
51                    (id,BusinessName,Country,Email) " + "VALUES (" +
52                        customer.getId() + ", " + customer.getBusinessName() +
53                            + ", " + customer.getCountry() + ", " +
54                                customer.getEmail() + ")";
55                stmt = c.createStatement();
56                stmt.executeUpdate(sql);
57                c.commit();
58            } catch (Exception e) {
59                System.err.println(e.getClass().getName() + ":" +
60                    e.getMessage());
61            }
62        }

```

```

47     }
48 }
49
50 for (Operation operation : operations) {
51     try {
52         if (operation.getAssistant() != null)
53             sql = "INSERT INTO CustomerAssistant
54                 (idHead,idAssistant,IdCustomer,Total,PersonalCost)
55                 " + "VALUES (" + operation.getId() + ", " +
56                 operation.getAssistant().getId() + ", " +
57                 operation.getCustomer().getId() + ", " +
58                 operation.getTotal() + ", " +
59                 operation.getOPersonalCost() + ");";
60     } catch (Exception e) {
61         System.err.println(e.getClass().getName() + ":" +
62             e.getMessage());
63     } try {
64         for (Pair<Article, Integer> i : operation.getRows()) {
65             sql = "INSERT INTO CustomerAssistantArticle
66                 (idHead,idArticle,qta) " + "VALUES (" +
67                 operation.getId() + ", " + i.getValue0().getId() +
68                 ", " + i.getValue1() + ");";
69         } catch (Exception e) {
70             System.err.println(e.getClass().getName() + ":" +
71             e.getMessage());
72     }
73 }
74 for (Inventory inventory : inventories) {
75     try {
76         sql = "INSERT INTO InventoryUser
77             (idHead,Description,MarketZone) " + "VALUES (" +
78             inventory.getId() + ", " + inventory.getDescription() +
79             ", " + inventory.getZone() + ");";
80     } catch (Exception e) {
81         System.err.println(e.getClass().getName() + ":" +
82             e.getMessage());
83 }

```

```

83    try {
84        for (Article article : inventory.getArticles ()) {
85            sql = "INSERT INTO InventoryArticle
86                (idHead,idArticle) " + "VALUES (" +
87                    inventory.getId() + "," + article.getId() + ");";
88            stmt = c.createStatement();
89            stmt.executeUpdate(sql);
90            c.commit();
91        }
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
102                        (IdCompound,IdComponent) " + "VALUES (" +
103                            + article.getId() + "," + a.getId() + ");";
104                    stmt = c.createStatement();
105                    stmt.executeUpdate(sql);
106                    c.commit();
107                }
108            }
109        }
110        try {
111            sql = "INSERT INTO Article (Id,Name,Price) " +
112                "VALUES (" + article.getId() + "," + "
113                    article.getName() + "," + article.getPrice() + ");";
114            stmt = c.createStatement();
115            stmt.executeUpdate(sql);
116            c.commit();
117        }
118    }
119
120    for (String notify : notCenter.getNotification ()) {
121        try {
122            sql = "INSERT INTO Notification (Message) " +
123                "VALUES (" + notify + ");";
124            stmt = c.createStatement();
125            stmt.executeUpdate(sql);
126            c.commit();
127        }
128        catch (Exception e) {
129            System.err.println(e.getClass().getName() + ":" +
130                e.getMessage());
131        }
132    }

```

```

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 10: 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 verrá messa a true interrompendo il loop.

Listing 11: Implementazione Run()

4 UnitTest

Prendiamo adesso in analisi la parte dei test effettuati per verificare la correttezza del codice. La base dati utilizzata é 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 quantitá 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
8         Pair<>(assistant.getInventory().getArticles().get(1), 20));
9     articles .add(new
10        Pair<>(assistant.getInventory().getArticles().get(2), 50));
11
12     Customer customer = p.getCustomers().get(2);
13
14     assistant .createOperation(customer, articles );
15
16     Operation createdOperation =
17         p.getOperations().get(p.getOperations().size() - 1);
18
19     assertAll("Order's Data",
20             () -> assertEquals(createdOperation.getAssistant(),
21                     assistant),
22             () -> assertEquals(createdOperation.getCustomer(),
23                     customer),
24             () -> assertEquals(createdOperation.getArticles().get(0),
25                     articles .get(0).getValue0()),
26             () -> assertEquals(createdOperation.getArticles().get(1),
27                     articles .get(1).getValue0())
28         );
29
30 }

```

Listing 12: 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() <
19                     operationCountBefore1),
20             () -> assertFalse(p.getOperations().contains(finalOp))
21         );
22
23 }

```

Listing 13: 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(
6              (int)((Math.random() * (p.getInventories().size() - 1) + 1)
7                  );
8          admin.createAssistant("UnitTest",
9              "studiomartino", 5.5F, inventory, "unitTest@gmail.com");
10         User createUser = p.getUsers().get(p.getUsers().size() - 1);
11
12         assertTrue(createUser instanceof Assistant);
13
14         Assistant createAssistant = (Assistant) createUser;
15
16         assertEquals("Test create assistant",
17             () -> assertEquals(inventory,
18                 createAssistant.getInventory()),
19             () -> assertEquals(createAssistant.getId(),
20                 p.getUsers().get(p.getUsers().size() - 1).getId())
21         );
22     }
23 }
```

Listing 14: Implementazione creazione assistente

Cancelliamo adesso un assistente e verifichiamo che non sia più 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          assertFalse(p.getUsers().contains(id));
15
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()){
24                     if(u instanceof Assistant){
25                         if(((Assistant)u).getInventory().equals(t)){
26                             return true;
27                         }
28                     }
29                 }
30             }
31         }
32     }
33 }
```

Listing 15: Implementazione eliminazione assistente

4.2.2 Creazione/eliminazione inventario

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

```

1 @Test
2     @DisplayName("Create InventoryTest")
3     void testCreateInventory() {
4
5         ArrayList<Article> articles = new ArrayList<>();
6         articles.add(p.getArticles().get(1));
7         articles.add(p.getArticles().get(2));
8         articles.add(p.getArticles().get(3));
9         int preSize = p.getInventories().size();
10        admin.createInventory("description", "Italy", articles);
11
12        assertEquals("Test create assistant",
13                     () -> assertEquals(preSize + 1, p.getInventories().size()),
14                     () -> assertEquals(articles,
15                                     p.getInventories().get(p.getInventories().size() - 1).getArticles())
16    );
17 }
```

Listing 16: Implementazione creazione inventario

Adesso lo eliminiamo e verifichiamo che non esista più.

```

1 @Test
2     @DisplayName("Delete InventoryTest")
3     void testDeleteInventory() {
4
5         int preSize = p.getInventories().size();
6         boolean check;
7 }
```

```

8     ArrayList<Article> articles = new ArrayList<>();
9     articles .add(p.getArticles().get(1));
10    articles .add(p.getArticles().get(2));
11    articles .add(p.getArticles().get(3));
12    admin.createInventory("description", "Italy", articles );
13    preSize = p.getInventories().size();
14
15    int lastCat =
16        p.getInventories().get(p.getInventories().size() -1).getId();
17    check = checkInventory(lastCat);
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 17: 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             () ->
10                assertTrue(p.getArticles().get(p.getArticles().size() -1)
11                    instanceof Product),
12                () ->
13                    assertEquals(p.getArticles().get(p.getArticles().size() -1).getPrice(),
14                        3.5F)
15            );
16
17         int preSize2 = p.getArticles().size();
18         ArrayList<Article> articles = new ArrayList<>();
19         articles .add(p.getArticles().get(1));
20         articles .add(p.getArticles().get(2));
21
22         float tmp = 0 ;
23         for ( Article a : articles )
24             tmp += a.getPrice();
25         float prePrice = tmp;
26
27         admin.createProduct("Compound article",articles);
```

```

24     assertAll("Compound Article",
25             () -> assertEquals(preSize2 + 1, p.getArticles().size()),
26             () ->
27                 assertTrue(p.getArticles().get(p.getArticles().size() - 1)
28                             instanceof Compound),
29             () ->
30                 assertEquals(p.getArticles().get(p.getArticles().size() - 1).getPrice(),
31                             prePrice)
32         );

```

Listing 18: Implementazione creazione prodotti

Analogamente eliminiamo i prodotti. Controlliamo inoltre che non possa cancellarsi un articolo che è 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     }

```

Listing 19: Implementazione eliminazione prodotti

4.2.4 Eliminazione cliente

Verifichiamo che possa essere correttamente rimosso un cliente dello studio.

```
1  @Test
2      @DisplayName("Delete CustomerTest")
3      void testDeleteCustomer() {
4
5          Customer C1 = null;
6          Customer C2 = null;
7
8          for (Customer cli : p.getCustomers()){
9              if( cli.getId() == 1)
10                  C1 = cli;
11              if( cli.getId() == 2)
12                  C2 = cli;
13          }
14
15          admin.deleteCustomer(1);
16          assertFalse(p.getCustomers().contains(C1));
17
18      }
```

Listing 20: Implementazione eliminazione cliente

4.3 CoreTest

Controlliamo che le connessioni tra database si effettuino correttamente.

4.3.1 Login test

```
1  @Test
2      @DisplayName("Login user Test")
3      void testLoginUser() {
4
5          p.login("Pampa", "studiomartino");
6
7          User user = null;
8
9          for (User i : p.getUsers()) {
10              if (i.getName().equals("Pampa")) {
11                  user = i;
12              }
13          }
14
15          User expectedUser1 = user;
16
17          assertAll("Assistant Login",
18                  () -> assertEquals(expectedUser1, p.getActiveUser()),
19                  () -> assertTrue(p.getActiveUser() instanceof Assistant)
20          );
21
22          p.logout();
23
24          p.login("Nicola", "ilmiostudio");
```

```

25     user = null;
26
27     for (User i : p.getUsers()) {
28         if (i.getName().equals("Nicola")) {
29             user = i;
30         }
31     }
32
33     User expectedUser2 = user;
34
35     assertEquals("Dentist Login",
36                 () -> assertEquals(expectedUser2, p.getActiveUser()),
37                 () -> assertTrue(p.getActiveUser() instanceof Dentist)
38             );
39
40 }
```

Listing 21: 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)*

```

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 = "Paperino";
15     String customerCountry = "Paperinolandia";
16     String customerEmail = "paperino.senzacognome@gmail.com";
17
18     sql = "INSERT INTO Customer (BusinessName,Country,Email) "
19          + "VALUES (" + customerName + ", " + customerCountry
20          + ", " + customerEmail + ");";
21
22     stmt = c.createStatement();
23     stmt.executeUpdate(sql);
24     c.commit();
25
26     String articleName = "Penny";
27     float articlePrice = 10.2F;
```

```

27     sql = "INSERT INTO Article (Name,Price) " + "VALUES (" +  

28         articleName + ", " + articlePrice + ");";  

29     stmt = c.createStatement();  

30     stmt.executeUpdate(sql);  

31     c.commit();  

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) &&  

39             i.getCountry().equals(customerCountry) &&  

40             i.getEmail().equals(customerEmail)) {  

41                 newCustomer = i;  

42             }  

43         }  

44         for (Article i : p.getArticles()) {  

45             if (i.getName().equals(articleName) && i.getPrice() ==  

46                 articlePrice) {  

47                 newArticle = i;  

48             }  

49         }  

50         Customer finalNewCustomer = newCustomer;  

51         Article finalNewArticle = newArticle;  

52         assertAll("Load From DB",  

53             () -> assertNotNull(finalNewCustomer),  

54             () -> assertNotNull(finalNewArticle)  

55         );  

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

57         Dentist admin = (Dentist) p.getActiveUser();  

58  

59         admin.createProduct("Berretto", 12.2F);  

60         admin.createCustomer("Minnie", "paperinolandiaSud",  

61             "minnie.senzacognome@gmail.com");  

62  

63         p.upload(c);  

64         c = DBCollectionTest.getInstance();  

65         stmt = c.createStatement();  

66  

67         int risArticle = 0;  

68         rs = stmt.executeQuery("SELECT COUNT(*) as ris FROM Article  

69             WHERE name='Berretto';");  

70         while (rs.next()) {  

71             risArticle = rs.getInt("ris");  

72         }  

73         int risCustomer = 0;  

74         rs = stmt.executeQuery("SELECT COUNT(*) as ris FROM  

75             Customer WHERE businessname='Minnie' AND  

76                 country='paperinolandiaSud' AND  

77                     email='minnie.senzacognome@gmail.com';");  

78         while (rs.next()) {  

79

```

```

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)
87       LIKE '%test%';";
88 stmt.executeUpdate(sql);
89 c.commit();
90
91 sql = "DELETE FROM Article WHERE LOWER(name) LIKE
92       '%test%';";
93 stmt.executeUpdate(sql);
94 c.commit();
95 }
```

Listing 22: 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 già esistente a questa situazione. Pensavamo di fare un interfaccia grafica affinché si potesse navigare nel menù, adesso semplice console, in modo più 'attuale'. Tuttavia abbiamo dovuto ridimensionare il tutto in quanto ci avrebbe portato via molto tempo ulteriore.