

# 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 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, 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

The diagram is a UML class diagram for a 'Library' system. It features several classes and interfaces, including:

- Book**: A central class with attributes like title, author, publisher, and year. It has methods like getBookInfo() and displayBookInfo().
- Author**: A class representing the author of a book, with attributes like name and birthYear.
- Publisher**: A class representing the publisher, with attributes like name and address.
- BookInterface**: An interface that defines methods for book-related operations.
- AuthorInterface**: An interface that defines methods for author-related operations.
- PublisherInterface**: An interface that defines methods for publisher-related operations.
- BookManager**: A class that manages the library's collection of books.
- AuthorManager**: A class that manages the library's collection of authors.
- PublisherManager**: A class that manages the library's collection of publishers.
- BookSearcher**: A class that searches for books in the library.
- AuthorSearcher**: A class that searches for authors in the library.
- PublisherSearcher**: A class that searches for publishers in the library.

The diagram illustrates the relationships between these classes, including inheritance, association, and aggregation. It also shows the use of interfaces to define common methods and the implementation of these methods by the concrete classes.

### 2.1.1 State Pattern

2

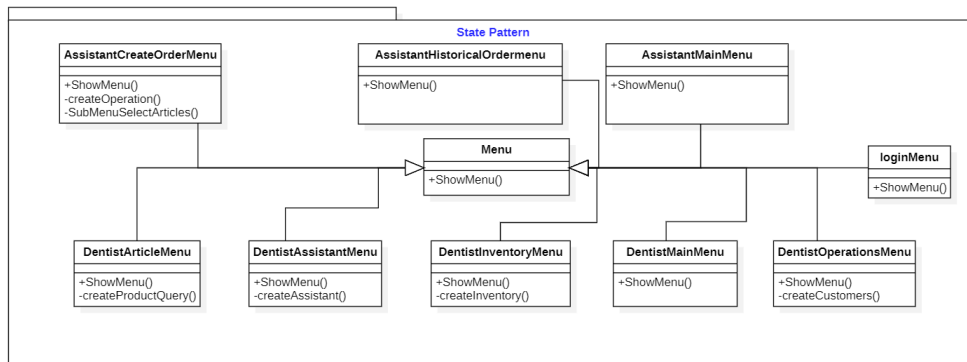


Figure 2: Diagramma delle classi (State pattern)

### 2.1.2 Observer Pattern

Questo pattern permette di definire una dipendenza  $1 \rightarrow N$  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*).

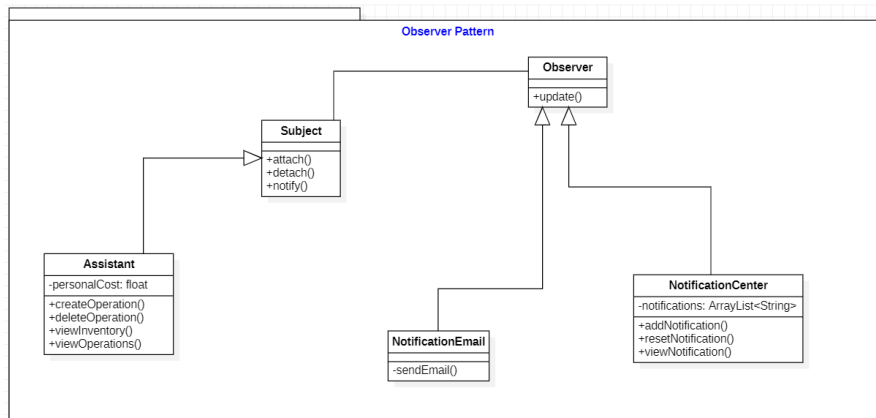


Figure 3: Diagramma delle classi (Observer pattern)

### 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 anziché un albero.

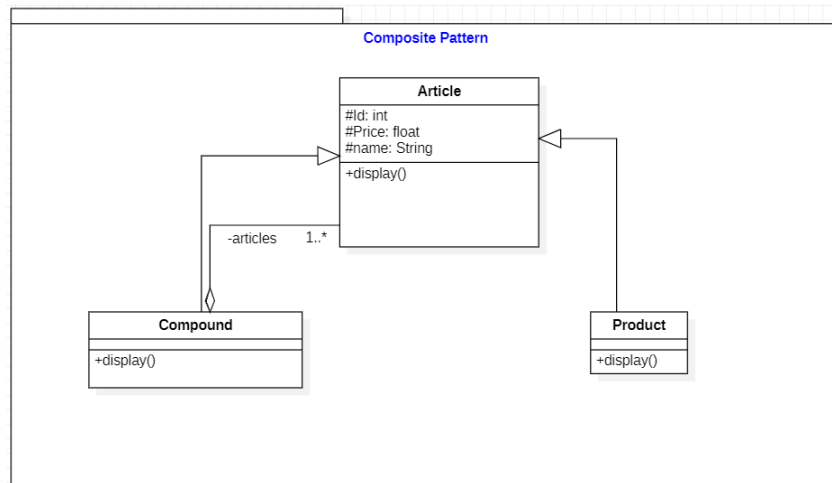


Figure 4: Diagramma delle classi (Composite pattern)

## 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.

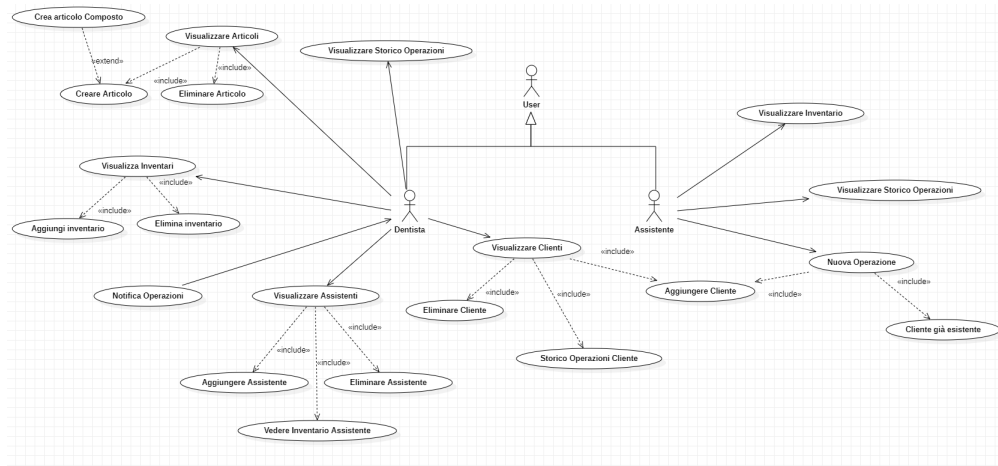


Figure 5: Diagramma dei casi d'uso

### 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.

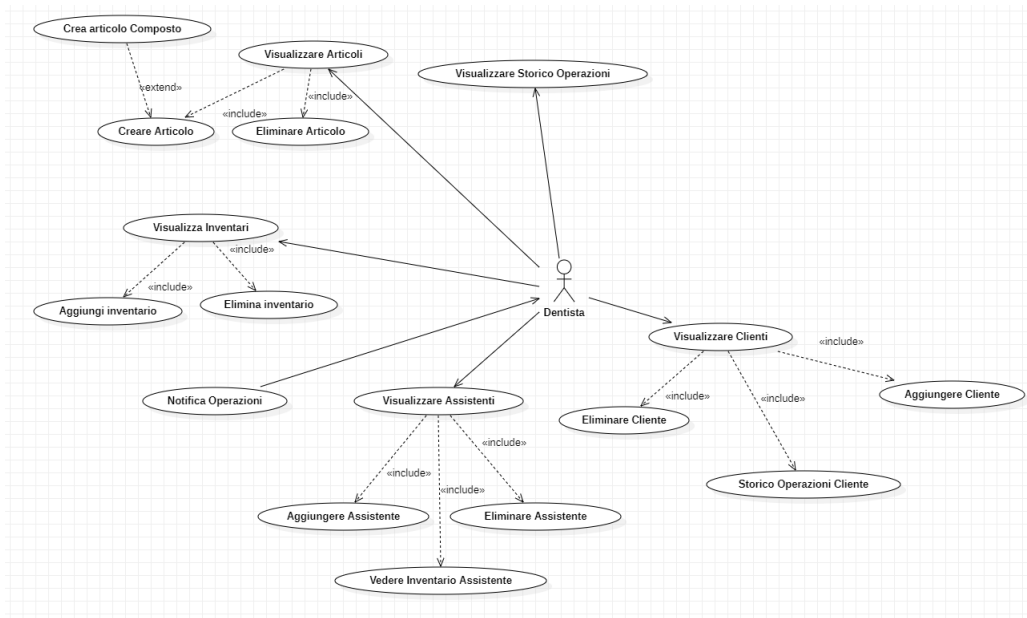


Figure 6: Diagramma dei casi d'uso del Dentista

### 2.2.2 Assistant'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.

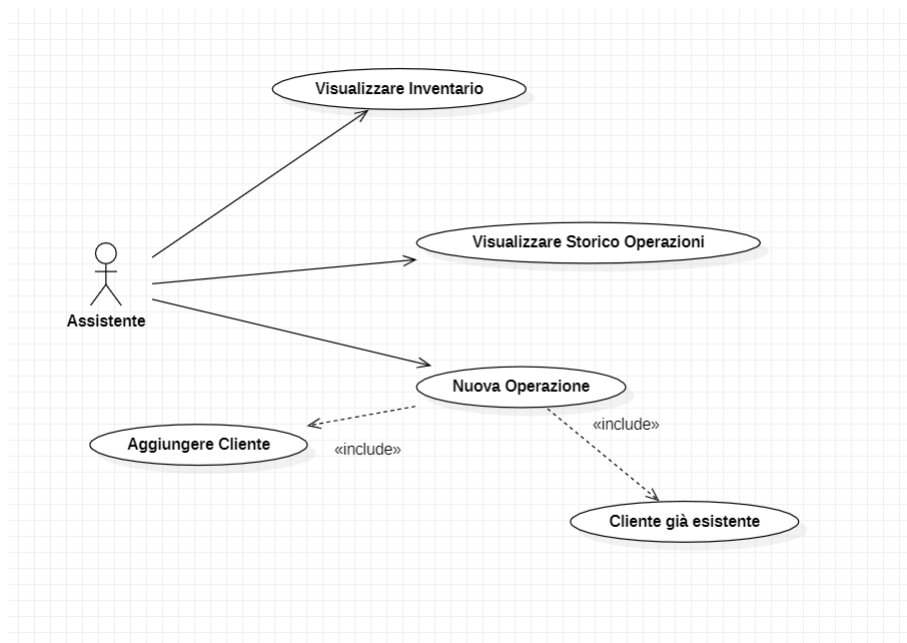


Figure 7: Diagramma dei casi d'uso dell'Assistente

### 2.3 E/R Diagram

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

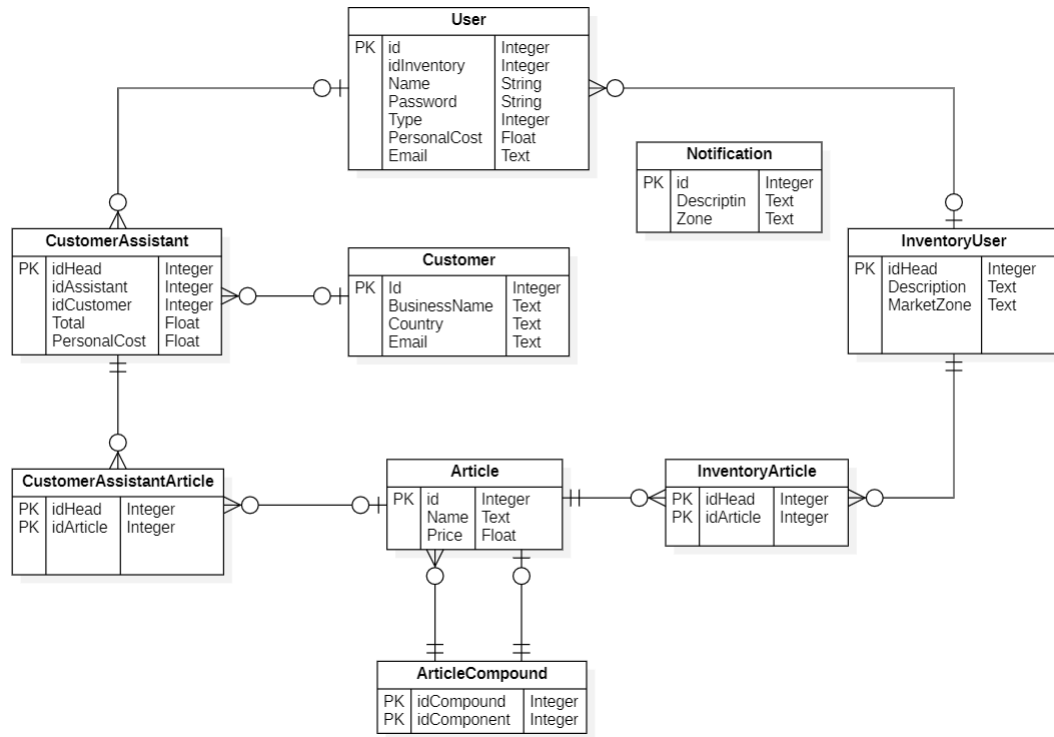


Figure 8: Diagramma E/R della struttura dati

### 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 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
	Filtro	Filtro	Filtro
1	10	Guanti	2.0
2	11	Bicchieri	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 sì 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
	Filtro	Filtro
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
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 " +
9             operation.getCustomer().getBusinessName() + " stata fatta da
10             " + operation.getAssistant().getName());
11     }
12 }
```

---

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
12        dell'admin
13    }
```

---

```

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     String test = "pippodima99@gmail.com";
23     String from = "ing.software.dimpa@gmail.com";
24
25     Properties properties = System.getProperties();
26     properties.put("mail.smtp.host", "smtp.gmail.com");
27     properties.put("mail.smtp.port", "465");
28     properties.put("mail.smtp.ssl.enable", "true");
29     properties.put("mail.smtp.auth", "true");
30
31     Session session = Session.getInstance(properties, new
32         javax.mail.Authenticator() {
33         protected PasswordAuthentication getPasswordAuthentication() {
34             return new
35                 PasswordAuthentication("ing.software.dimpa@gmail.com",
36                     "rkqvtlxwtcaczfjj\n"); //mbvmolrwpfmzcuoq
37                     neaüczeusvreoesu
38         }
39     });
40
41     session.setDebug(true);
42
43     try {
44         MimeMessage message = new MimeMessage(session);
45         message.setFrom(new InternetAddress(from));
46         message.addRecipient(Message.RecipientType.TO, new
47             InternetAddress(to));
48         message.setSubject(obj);
49         message.setContent(text, "text/html");
50
51         System.out.println("sending ...");
52         Transport.send(message);
53         System.out.println("inviato");
54     } catch (MessagingException mex) {
55         mex.printStackTrace();
56     }
57 }
58 }

```

---

Listing 2: Email Observer

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

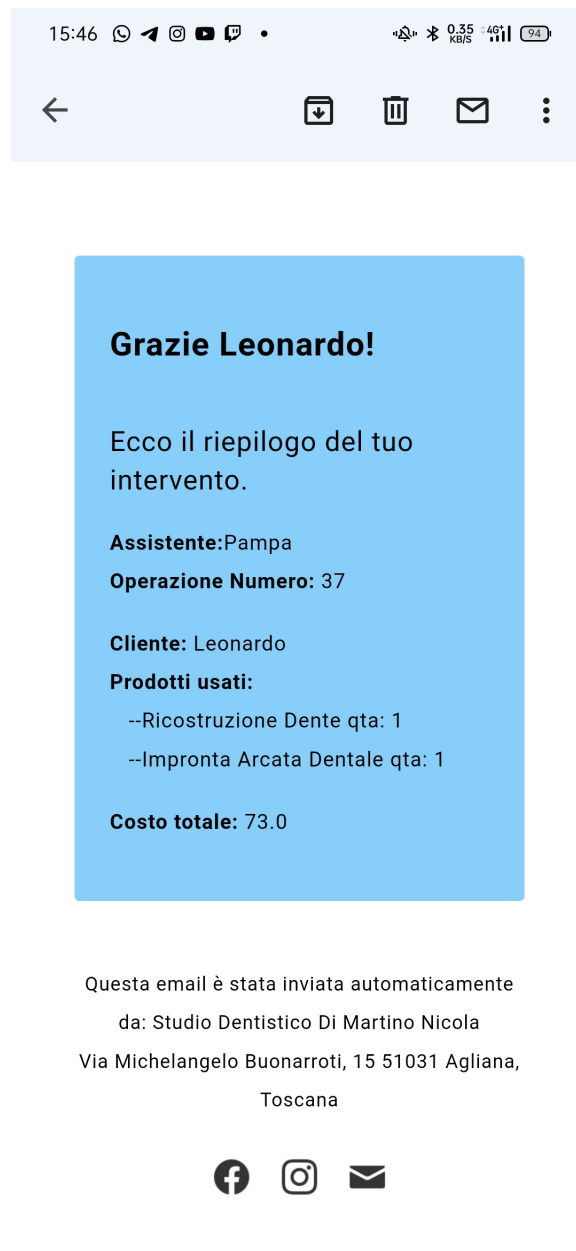


Figure 11: Email inoltrata all'assistente che ha effettuato l'operazione

### 3.1.2 Curiosita'

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

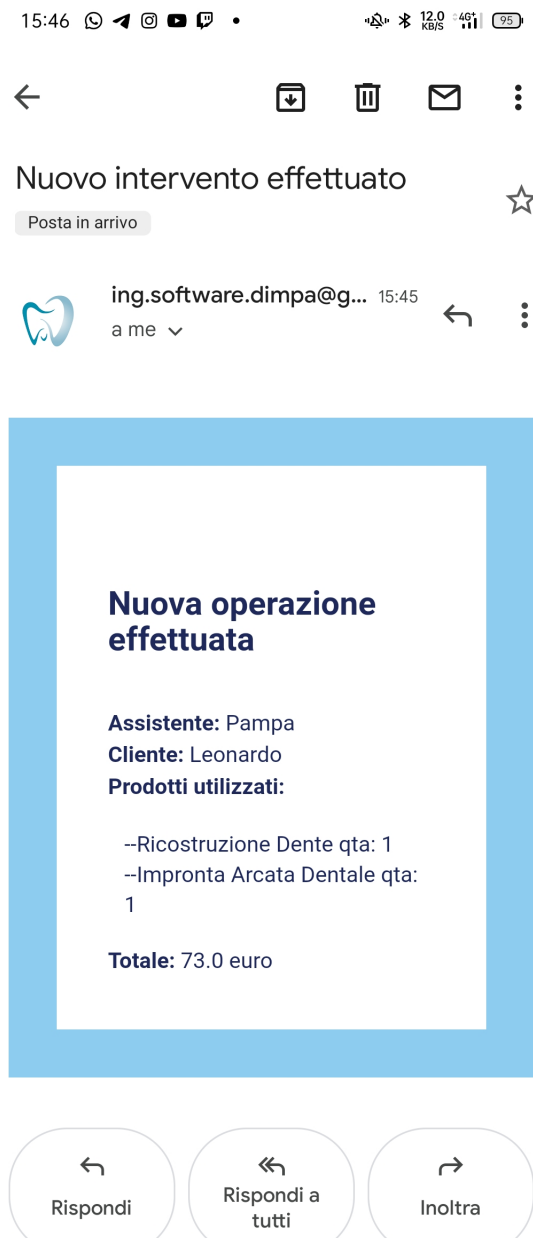


Figure 12: Email inoltrata al dentista.

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:

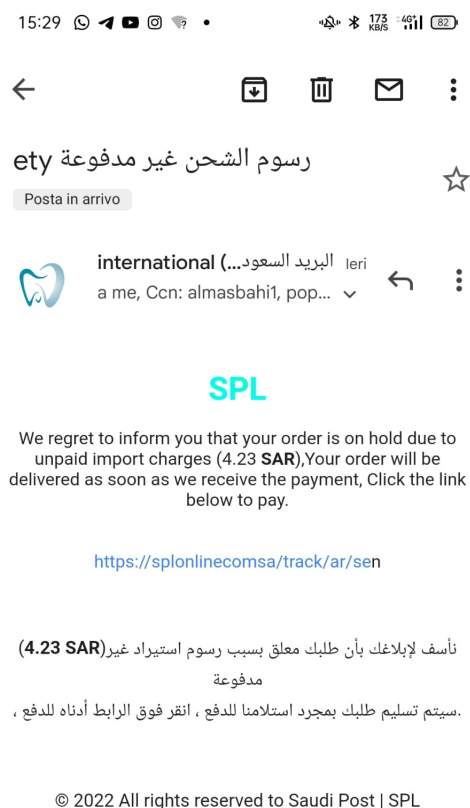


Figure 13: Email inoltrata al dentista.

## 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,  
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));  
7 }
```

---

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



attento 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 per, anche da un confronto con la realtà, abbiamo deciso di ignorare il problema perché se scegliamo di eliminare un determinato inventario 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
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 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
41             for (Article a : articles ) {
42                 if (a.getId() == idComponent) {
43                     components.add(a);
44                     break;
45                 }
46             }
47         }
48         articles .add(new Compound(name, components, id));
49     }
50
51     rs = stmt.executeQuery("SELECT * FROM InventoryUser;");
52     while (rs.next()) {
53         int id = rs.getInt("idHead");
54         String description = rs.getString("Description");
55         String marketZone = rs.getString("MarketZone");
56         ArrayList<Article> tmp = new ArrayList<>();
57         rs1 = stmt1.executeQuery("SELECT * FROM InventoryArticle
58             WHERE IdHead = " + id + " ");
59         while (rs1.next()) {
60             int idArticle = rs1.getInt("idArticle");
61             for (Article a : articles ) {
62                 if (a.getId() == idArticle) {
63                     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");
70     //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     if (type == 1) {
78         Inventory tmp = null;
79         for (Inventory i : inventories) {
80             if (i.getId() == idInventory) {
81                 tmp = i;
82             }
83         }
84         if (tmp == null) {
85             System.err.println("Inventory don't exist!");
86             break;
87         }
88     }
89     users.add(new Assistant(name, passHash, personalCost,
90         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     Customer tmpCustomer = null;
112     for (Customer i : customers) {
113

```

```

114         if (i.getId() == idCustomers) {
115             tmpCustomer = i;
116             break;
117         }
118     }
119
120     if (tmpCustomer == null) {
121         System.err.println("Customer don't exist!");
122         break;
123     }
124
125     ArrayList<Pair<Article, Integer>> tmp = new ArrayList<>();
126     rs1 = stmt1.executeQuery("SELECT * FROM
CustomerAssistantArticle WHERE IdHead = " + id + " ");
127     while (rs1.next()) {
128         int idArticle = rs1.getInt("idArticle");
129         int qta = rs1.getInt("qta");
130         for (Article a : articles) {
131             if (a.getId() == idArticle) {
132                 tmp.add(new Pair<>(a, qta));
133                 break;
134             }
135         }
136     }
137     operations.add(new Operation(total, cost, tmpAssistant, tmp,
tmpCustomer, id));
138 }
139 }
140 }

```

---

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.

---

```

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;",
"DELETE FROM CustomerAssistant;", "DELETE FROM
CustomerAssistantArticle;", "DELETE FROM
Notification;", "DELETE FROM Customer;", "DELETE
FROM InventoryArticle;", "DELETE FROM
InventoryUser;", "DELETE FROM Article;", "DELETE
FROM ArticleCompound;")) {
7             sql = s;
8             stmt.executeUpdate(sql);
9             c.commit();
10        }
11    } catch (Exception e) {
12        System.err.println(e.getClass().getName() + ": " +
e.getMessage());
13        System.exit(0);
14    }

```

```

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.getIdInventory().getId() + ", " + user.getEmail()
39                 + ")";
40         }
41
42         stmt = c.createStatement();
43         stmt.executeUpdate(sql);
44         c.commit();
45     } catch (Exception e) {
46         System.err.println(e.getClass().getName() + ": " +
47             e.getMessage());
48     }
49 }
50
51 for (Customer customer : customers) {
52     try {
53         sql = "INSERT INTO Customer
54             (id,BusinessName,Country,Email) " + "VALUES (" +
55             customer.getId() + ", " + customer.getBusinessName()
56             + ", " + customer.getCountry() + ", " +
57             customer.getEmail() + ")";
58
59         stmt = c.createStatement();
60         stmt.executeUpdate(sql);
61         c.commit();
62     } catch (Exception e) {
63         System.err.println(e.getClass().getName() + ": " +
64             e.getMessage());
65     }
66 }
67
68 for (Operation operation : operations) {
69     try {
70         if (operation.getAssistant() != null)
71             sql = "INSERT INTO CustomerAssistant

```

```

54         (idHead,idAssistant,IdCustomer>Total,PersonalCost)
55         " + "VALUES (" + operation.getId() + "," +
        operation.getAssistant().getId() + "," +
        operation.getCustomer().getId() + "," +
        operation.getTotal() + "," +
        operation.getOPersonalCost() + ");";
56     else
57         sql = "INSERT INTO CustomerAssistant
58         (idHead,idAssistant,IdCustomer>Total,PersonalCost)
59         " + "VALUES (" + operation.getId() + "," + -1
60         + "," + operation.getCustomer().getId() + "," +
        operation.getTotal() + "," +
        operation.getOPersonalCost() + ");";
61     stmt = c.createStatement();
62     stmt.executeUpdate(sql);
63     c.commit();
64 } catch (Exception e) {
65     System.err.println(e.getClass().getName() + ": " +
66     e.getMessage());
67 }
68 try {
69     for (Pair<Article, Integer> i : operation.getRows()) {
70         sql = "INSERT INTO CustomerAssistantArticle
71         (idHead,idArticle,qta) " + "VALUES (" +
72         operation.getId() + "," + i.getValue0().getId() +
73         "," + i.getValue1() + ");";
74         stmt = c.createStatement();
75         stmt.executeUpdate(sql);
76         c.commit();
77     }
78 } catch (Exception e) {
79     System.err.println(e.getClass().getName() + ": " +
80     e.getMessage());
81 }
82 }
83 for (Inventory inventory : inventories) {
84     try {
85         sql = "INSERT INTO InventoryUser
86         (idHead,Description,MarketZone) " + "VALUES (" +
        inventory.getId() + "," + inventory.getDescription() +
        "," + inventory.getZone() + ");";
        stmt = c.createStatement();
        stmt.executeUpdate(sql);
        c.commit();
    } catch (Exception e) {
        System.err.println(e.getClass().getName() + ": " +
        e.getMessage());
    }
    try {
        for (Article article : inventory.getArticles()) {
            sql = "INSERT INTO InventoryArticle
            (idHead,idArticle) " + "VALUES (" +
            inventory.getId() + "," + article.getId() + ");";
            stmt = c.createStatement();

```

```

87         stmt.executeUpdate(sql);
88         c.commit();
89     }
90     } catch (Exception e) {
91         System.err.println(e.getClass().getName() + ": " +
92             e.getMessage());
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             } catch (Exception e) {
108                 System.err.println(e.getClass().getName() + ": " +
109                     e.getMessage());
110             }
111         }
112     }
113     try {
114         sql = "INSERT INTO Article (Id,Name,Price) " +
115             "VALUES (" + article.getId() + ", " +
116             article.getName() + ", " + article.getPrice() + ")";
117         stmt = c.createStatement();
118         stmt.executeUpdate(sql);
119         c.commit();
120     } catch (Exception e) {
121         System.err.println(e.getClass().getName() + ": " +
122             e.getMessage());
123     }
124 }
125
126 for (String notify : notCenter.getNotification()) {
127     try {
128         sql = "INSERT INTO Notification (Message) " +
129             "VALUES (" + notify + ")";
130         stmt = c.createStatement();
131         stmt.executeUpdate(sql);
132         c.commit();
133     } catch (Exception e) {
134         System.err.println(e.getClass().getName() + ": " +
135             e.getMessage());
136     }
137 }
138
139 try {
140     stmt.close();
141     c.close();

```

```
134         } catch (Exception e2) {  
135             e2.printStackTrace();  
136         }  
137         instance = null;  
138     }  
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 verrà messa a true interrompendo il loop.

---

---

Listing 13: Implementazione Run()

## 4 UnitTest

mettere apposto i db