

Flower Shop

Ingegneria del software

Caterina Giardi, Beatrice Vangi



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Ingegneria Informatica
Università di Firenze

Indice

1	Introduzione	2
1.1	Statement	2
1.1.1	Funzionalità Customer	2
1.1.2	Funzionalità Florist	2
1.2	Strumenti utilizzati	3
2	Analisi dei requisiti	4
2.1	Use Case Diagram	4
3	Progettazione	5
3.1	Class Diagram	5
3.2	Template	6
3.3	Mock Up	7
4	Implementazione	13
4.1	Design Pattern	13
4.1.1	Observer	13
4.1.2	State	14
4.1.3	Singleton	17
4.1.4	Composite	17
4.2	Gestione dei CSV	17
4.3	Metodi Principali	19
4.3.1	Fill Order	19
4.3.2	Place Order	20
4.3.3	Login e Sign Up	20
5	Unit Test	23
5.1	Customer Test	23
5.2	Florist Test	24
5.3	OrderList Test	25
5.4	Program Test	25
5.5	Storage Test	27

1 Introduzione

1.1 Statement

Lo scopo del nostro progetto è quello di gestire un negozio di fiori, con la possibilità di ricevere e effettuare ordini, e al contempo poter prepararli e spedirli.

La fase iniziale prevede l'accesso dell'utente. Al momento dell'autenticazione, l'utente inserisce la propria mail: qualora fosse registrato, procede col login, altrimenti verrà reindirizzato alla registrazione nel sistema dove dovrà identificarsi come cliente o come fioraio. Dopo aver fatto il login, l'utente si troverà nel menù corrispondente alla sua categoria: se è un cliente sarà nel *CustomerMenu*, se è un fioraio nel *FloristMenu*.

- Una volta in *CustomerMenu* il cliente può effettuare tre azioni diverse: piazzare un ordine, vedere la sua lista di ordini, aprire la sua inbox oppure uscire dal programma.
- Da *FloristMenu* il fioraio, tramite input da tastiera, può scegliere tra le seguenti opzioni: vedere la lista di ordini, completare un ordine o vedere le disponibilità nel magazzino.

I dati su utenti, catalogo, ordini, messaggi, disponibilità del magazzino sono conservati all'interno di documenti CSV, dai quali sono caricati all'avvio e aggiornati durante l'esecuzione del programma.

1.1.1 Funzionalità Customer

Il cliente per poter **piazzare un ordine** deve prendere visione del catalogo, che viene stampato prima di scegliere gli elementi da inserire all'interno dell'ordine. Dopo che avrà scelto i prodotti, riceverà un messaggio nell'inbox: al suo interno si notifica che l'ordine è confermato e che verrà processato. Per **visualizzare il messaggio** basta accedere all'*InboxMenu* dal *CustomerMenu* e scegliere l'opzione "View messages". Esiste anche la possibilità di **ripulire l'inbox** tramite l'opzione "Empty inbox". Il cliente può infine **ri-vedere i suoi ordini passati**, visualizzando non solo l'id dell'ordine, il suo contenuto e prezzo, ma anche il suo stato (Processing, Shipped, Delivered).

1.1.2 Funzionalità Florist

Per **completare un ordine** lo si prende in carico prelevandolo dalla lista. Dopodiché il fioraio deve sapere come comporre l'ordine e le composizioni floreali al suo interno, così da poterle ricreare e inserirle nella *Box* che verrà spedita; per fare ciò si consulta l'ordine che implicitamente specifica come è composto lo stesso e per scomporre l'eventuale Bouquet si chiama il metodo *getItem()*. Al fine di poter considerare l'ordine come un insieme di prodotti e il Bouquet come un composto di questi, sfruttiamo il pattern Composite. Ogni fiore ha un costo, di conseguenza il prezzo dei bouquet è dato dalla somma di tutti i fiori che lo compongono e dal prezzo delle decorazioni eventuali, in aggiunta a un costo fisso di manodopera.

Dal momento che si fa in modo che in magazzino ci siano sempre le risorse necessarie alla creazione dei prodotti presenti sul catalogo, il cliente riceverà sempre il prodotto richiesto. Una volta finito di prelevare gli elementi dallo *Storage* e composto il prodotto richiesto, il fioraio inserisce questo nella *Box*, che al termine dell'operazione verrà chiusa e spedita.

La classe ***Supplier*** rappresenta i fornitori, il cui ruolo è quello di istanziare i fiori e inviarli allo *Storage* così che possano essere immagazzinati. Tutto ciò avviene tramite un pattern Observer nel quale i *Supplier* svolgono il ruolo di ConcreteObserver, infatti quest'ultimi invieranno i fiori una volta notificati dallo *Storage*, che è un ConcreteSubject. La notifica viene inviata nel momento in cui il fioraio constata che le quantità di fiori presenti in magazzino sono inferiori a una determinata soglia.

Al completamento dell'ordine il fioraio pianifica il ritiro da parte della compagnia di spedizione (*ShippingCompany*). A questa azione corrisponde una **notifica** inviata al cliente che comunica l'affidamento del pacco al corriere. Anche questa procedura è implementata seguendo lo scheletro del pattern Observer. La notifica è inviata dal *CustomerNotifier* (ConcreteObserver) e consiste nell'invio di un messaggio al cliente. Infine, non appena il pacco risulta essere giunto a destinazione, viene inviato l'ultimo messaggio per confermare l'avvenuta consegna.

1.2 Strumenti utilizzati

L'applicazione è sviluppata in Java. Per i diagrammi di classi e di casi d'uso è stato utilizzato il software StarUML. È stato realizzato un livello di persistenza tramite file CSV, gestiti in lettura e scrittura tramite la libreria OpenCSV [1]. Infine, i test sono stati realizzati tramite le librerie di JUnit 5 [3].

2 Analisi dei requisiti

2.1 Use Case Diagram

Nell'analisi dei requisiti si rende utile lo Use Case Diagram (Figura 1) che illustra come si strutturano le relazioni tra attori e casi d'uso.

Dal diagramma si nota la centralità del login, che è un'azione preliminare a qualsiasi funzionalità del programma di cui si voglia usufruire, e inoltre vengono evidenziate le azioni che è possibile compiere e i sotto-casi di ogni caso d'uso.

Nel caso in esame gli attori sono quattro: Florist, Customer, Supplier e Shipping Company. I primi due, in particolare svolgono un ruolo chiave, in quanto costituiscono i due principali interlocutori.

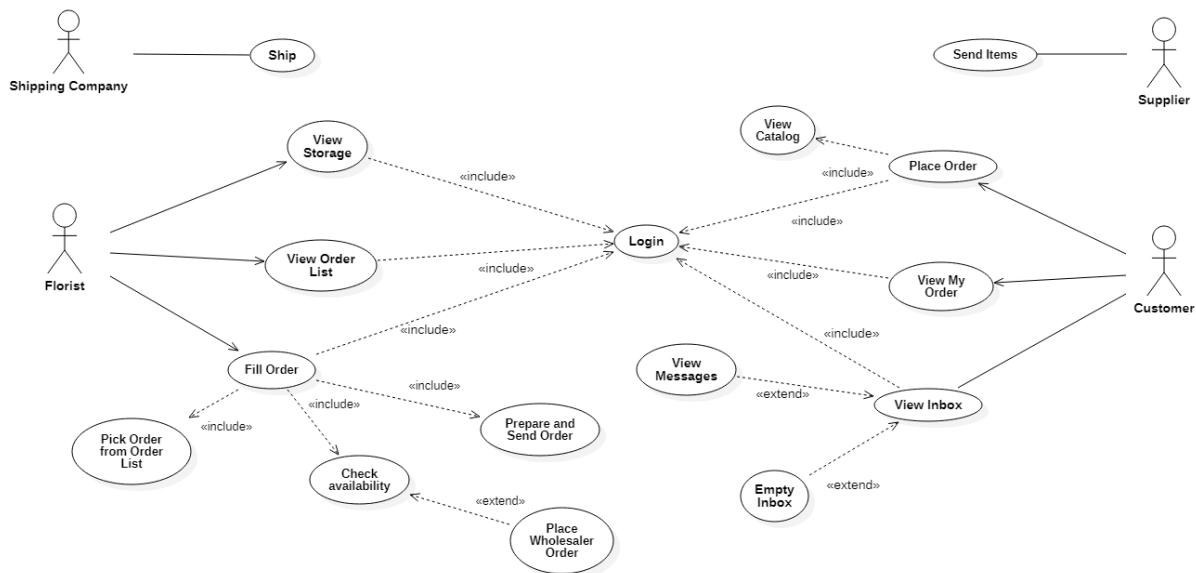


Figura 1: Use Case Diagram.

3 Progettazione

3.1 Class Diagram

Come si può vedere nella seguente figura 2, nel Class Diagram sono presenti 4 pattern differenti:

- Observer: sfruttando questo pattern è stato possibile creare un sistema di dipendenza tra *Storage* e *Supplier*, dove automaticamente lo *Storage* notifica il *Supplier* non appena le quantità di un prodotto presenti in magazzino sono minori di una certa soglia. Inoltre tramite l'Observer pattern abbiamo implementato un sistema di messaggi di aggiornamento di cui il cliente può usufruire. Questo è ciò che avviene con le classi *Order* e *CustomerNotifier*.
- Singleton: abbiamo creato *Program*, *Catalog*, e *OrderList* come Singleton.
- Composite: si è rivelato utile l'uso del pattern composite disponendo di una classe *Bouquet* composto da più elementi della classe *Flower* e *Decoration*. In questo modo si accede facilmente agli elementi di *Bouquet* e possiamo considerare queste tre classi come *Product*.
- State: lo abbiamo scelto per realizzare i menù, così da avere una sorta di interfaccia per facilitare l'accesso alle funzionalità offerte dal programma.

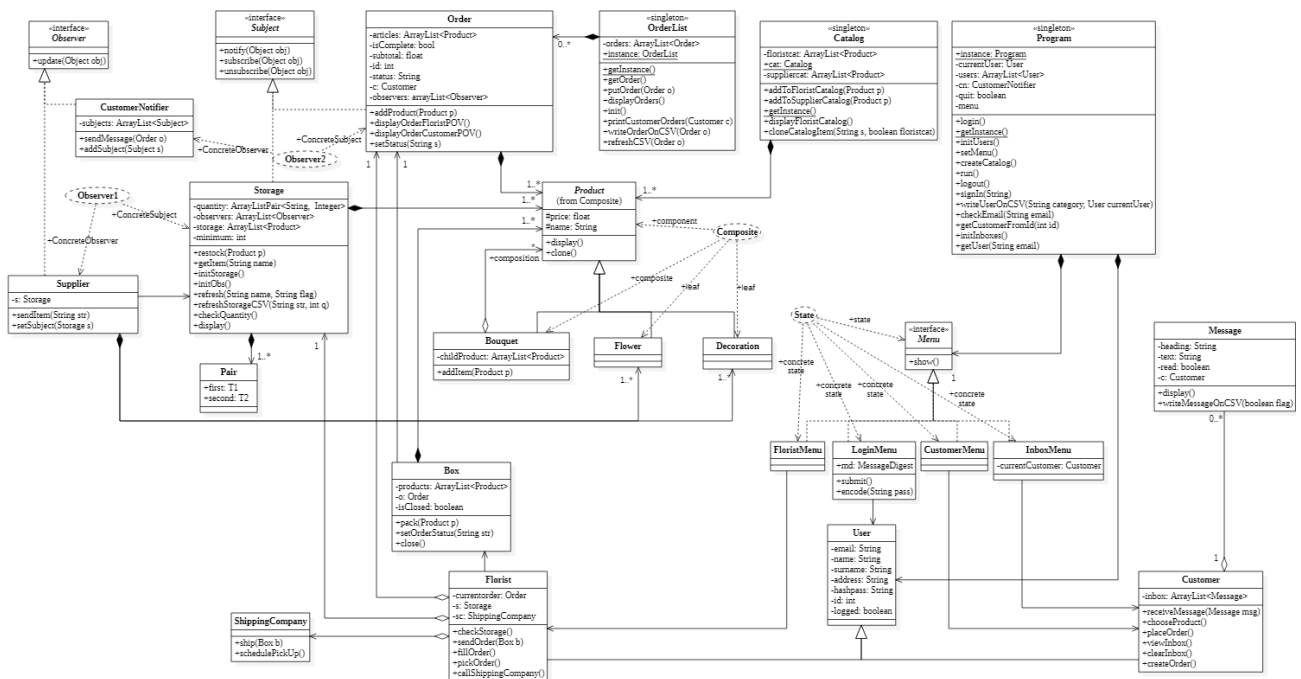


Figura 2: Class Diagram

3.2 Template

I template esposti di seguito definiscono le funzionalità che stanno alla base del software.

UC	Place Order
Level	User Goal
Actor	Customer
Basic Course	<ol style="list-style-type: none"> 1. Il cliente si trova nella schermata del MenuCustomer e clicca su "Place Order" (Figura 6). 2. Il sistema mostra il Catalogo (Figura 8). 3. Il cliente seleziona i prodotti che vuole ordinare. 4. Una volta selezionati i prodotti, clicca su Order. 5. Il sistema inserisce l'ordine nell'OrderList(Figura 5). 6. Il sistema ritorna alla schermata del MenuCustomer (Figura 6).

UC	View messages and clear inbox
Level	User Goal
Actor	Customer
Basic Course	<ol style="list-style-type: none"> 1. Il cliente si trova nella schermata del MenuCustomer e clicca su "View Inbox" (Figura 6). 2. Il sistema mostra il menu in Figura 9. 3. L'utente clicca su "View messages" 4. Il sistema mostra la lista di messaggi associati all'utente. (Figura 9). 5. Dopo aver preso visione dei messaggi, il cliente clicca sulla freccia per tornare al menu precedente. 6. Il cliente clicca su "Empty Inbox". 7. Il sistema cancella tutti i messaggi associati all'utente. 8. Il cliente clicca su "Back" (Figura 9). 9. Il sistema mostra la schermata del MenuCustomer (Figura 6).
Alternative Course	<ol style="list-style-type: none"> 4.a L'utente non ha messaggi. <ol style="list-style-type: none"> 4.a.1 Il sistema mostra una schermata con scritto "There are no messages" (Figura 6).

UC	View Order List
Level	User Goal
Actor	Florist
Basic Course	<ol style="list-style-type: none"> 1. Il fioraio si trova nella schermata del MenuFlorist e clicca su "View orders" (Figura 6). 2. Il sistema mostra la finestra Order List (Figura 12). 3. Il fioraio può visionare la lista di tutti gli ordini, completati e non.

UC	Fill Order
Level	User Goal
Actor	Florist
Basic Course	<ol style="list-style-type: none"> 1. Il fioraio si trova nella schermata del MenuFlorist e clicca su "Fill order" (Figura 6). 2. Il sistema mostra la finestra Fill order, che mostra il primo ordine incompleto dalla lista di ordini (Figura 11). 3. Il fioraio visiona gli articoli presenti nell'ordine. 4. Mano a mano che il fioraio procede all'elaborazione degli articoli, accede allo storage per prelevare gli elementi necessari. 5. Una volta che tutti i box relativi agli articoli hanno il segno di spunta, il fioraio clicca su "Call shipping company" per fissare il ritiro dell'ordine (Figura 11). 6. La finestra si chiude. 7. Il sistema mostra la schermata del MenuFlorist (Figura 6).
Alternative Course	<ol style="list-style-type: none"> 5.a La quantità di almeno un elemento nello storage oltrepassa la soglia minima. <ol style="list-style-type: none"> 5.a.1 Il sistema contatta automaticamente il Supplier per rifornirsi dell'elemento mancante.

3.3 Mock Up

I Mock Up riportati di seguito descrivono l'interfaccia grafica destinata agli utenti.

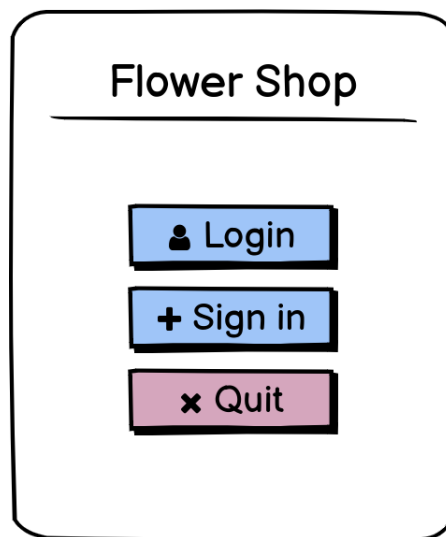
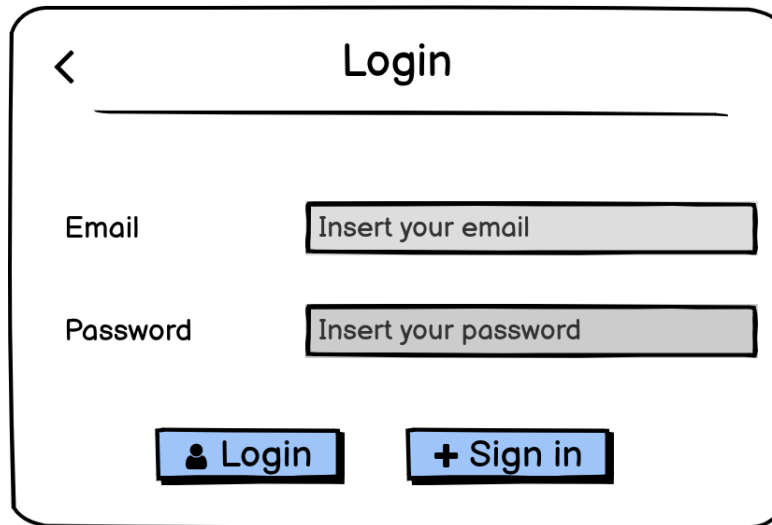
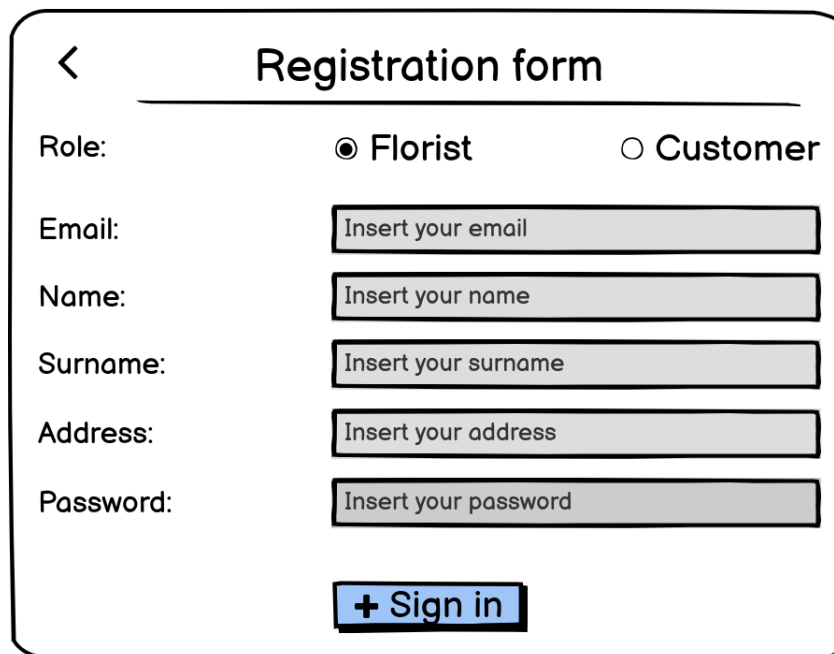


Figura 3: Finestra iniziale.



The Login form is contained within a rounded rectangle. At the top left is a back arrow icon. The title 'Login' is centered at the top. Below the title is a horizontal line. The form contains two input fields: 'Email' with the placeholder text 'Insert your email' and 'Password' with the placeholder text 'Insert your password'. At the bottom, there are two buttons: a blue button with a person icon and the text 'Login', and a blue button with a plus icon and the text 'Sign in'.

Figura 4: Finestra di Login.



The Registration form is contained within a rounded rectangle. At the top left is a back arrow icon. The title 'Registration form' is centered at the top. Below the title is a horizontal line. The form starts with a 'Role:' label followed by two radio buttons: 'Florist' (which is selected) and 'Customer'. Below this are five input fields: 'Email' (placeholder: 'Insert your email'), 'Name' (placeholder: 'Insert your name'), 'Surname' (placeholder: 'Insert your surname'), 'Address' (placeholder: 'Insert your address'), and 'Password' (placeholder: 'Insert your password'). At the bottom is a blue button with a plus icon and the text 'Sign in'.

Figura 5: Finestra di registrazione.

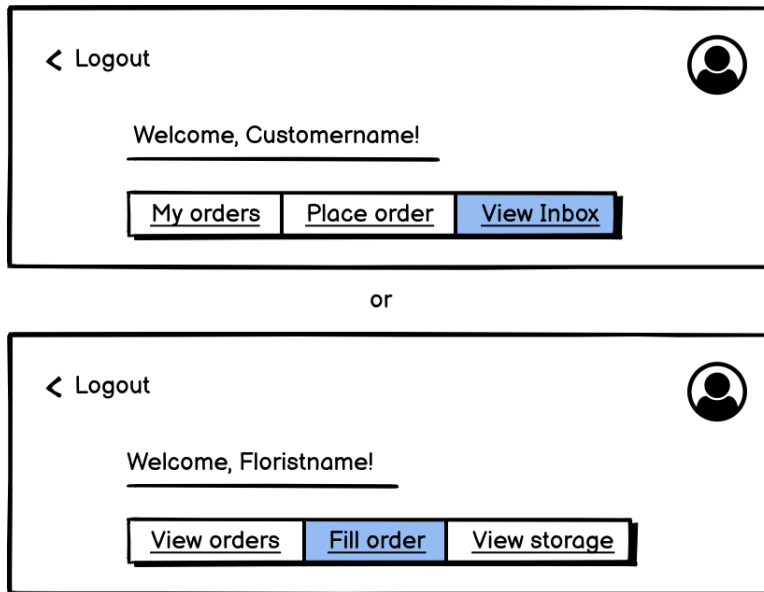


Figura 6: Finestra di menu del cliente e del fioraio.

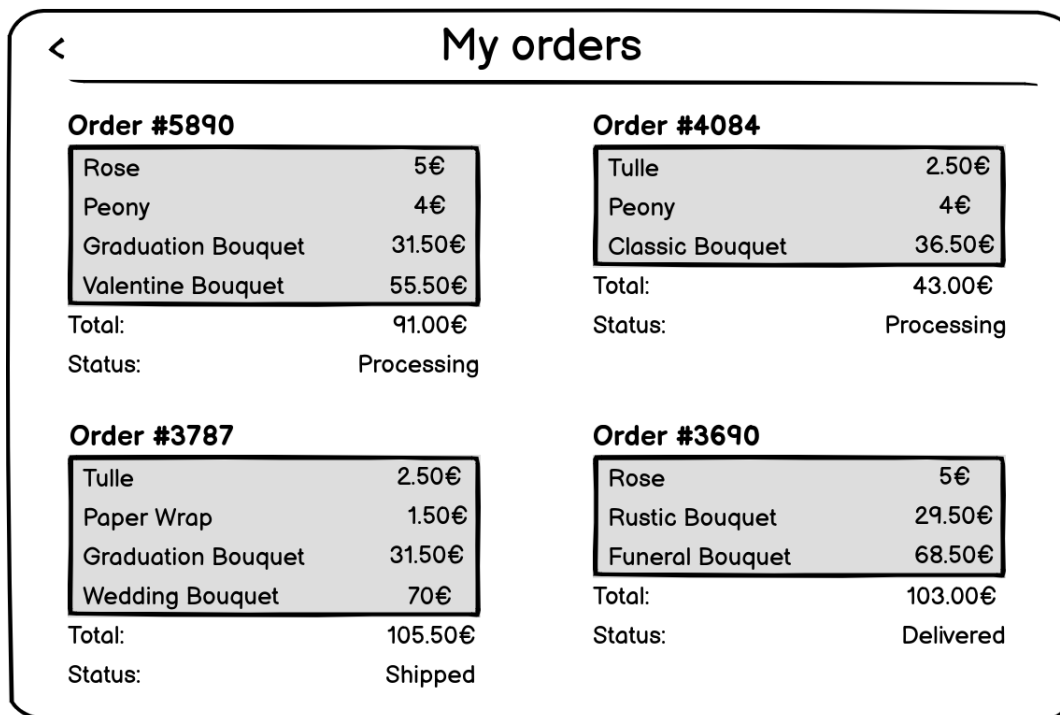


Figura 7: Finestra che mostra la lista di ordini effettuati da un cliente.

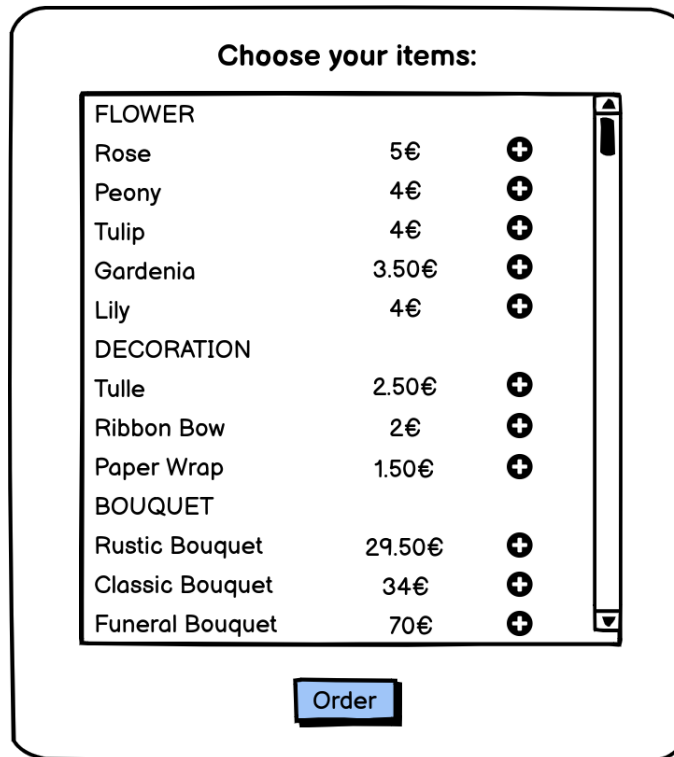


Figura 8: Finestra che mostra gli articoli del catalogo che si possono aggiungere all'ordine.

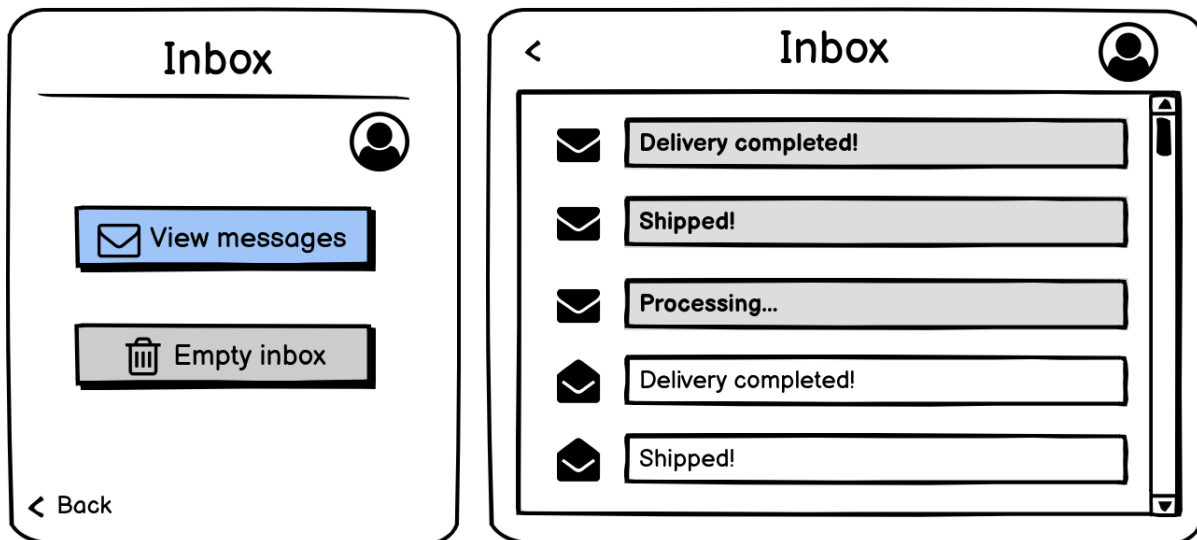
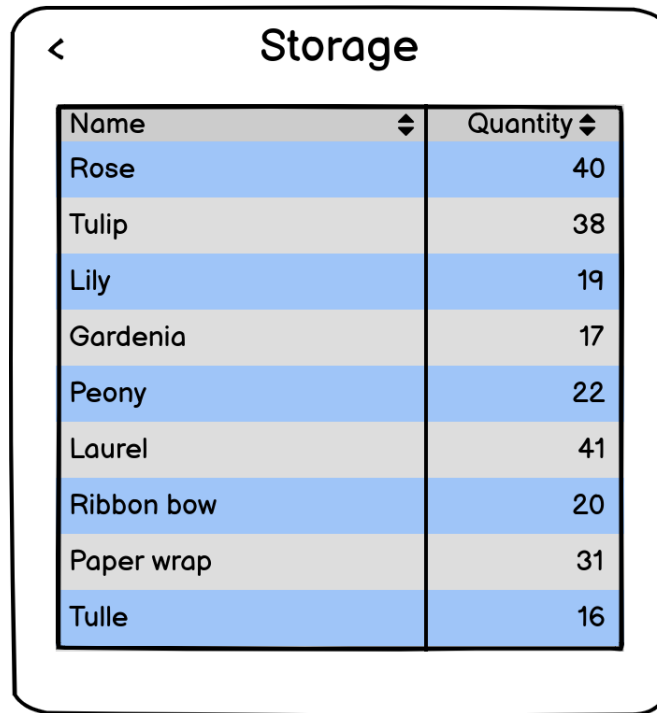


Figura 9: Finestra che mostra l'*InboxMenu* e la lista di messaggi in ingresso.



A screenshot of a mobile application window titled "Storage". It features a table with two columns: "Name" and "Quantity". The table lists ten items with their respective quantities. The rows alternate between light blue and light gray backgrounds.

Name	Quantity
Rose	40
Tulip	38
Lily	19
Gardenia	17
Peony	22
Laurel	41
Ribbon bow	20
Paper wrap	31
Tulle	16

Figura 10: Finestra che mostra gli articoli presenti in magazzino e la rispettiva quantità.



A screenshot of a mobile application window titled "Fill order:". Below the title, it says "Filling order #8128:". There are three items listed with checkboxes: "Rose" (checked), "Graduation Bouquet" (unchecked), and "Classic Bouquet" (unchecked). At the bottom, there is a blue button with the text "Call Shipping Company".

Figura 11: Finestra che mostra gli elementi dell'ordine da completare e il bottone per chiamare lo spedizioniere per fissare il ritiro.

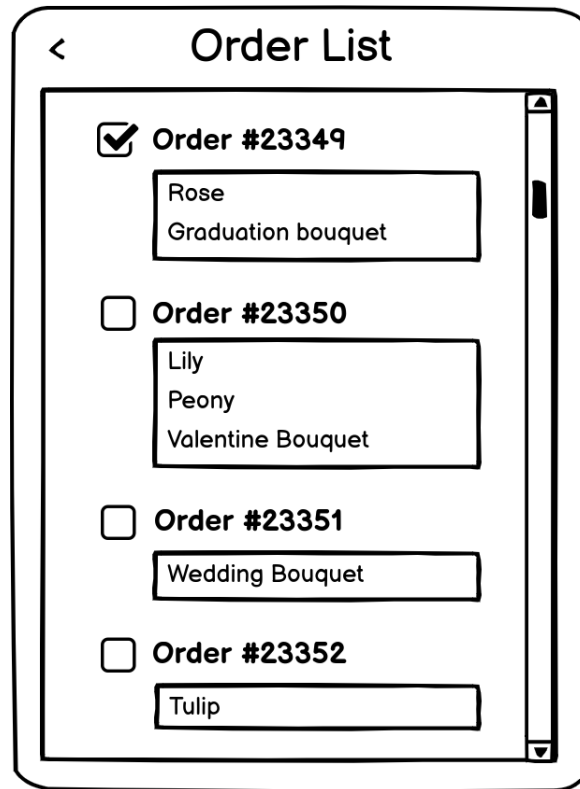


Figura 12: Finestra che mostra la lista di ordini.

4 Implementazione

4.1 Design Pattern

4.1.1 Observer

L'Observer è un pattern di tipo comportamentale e definisce una relazione tra oggetti, così che quando un oggetto (Subject) cambia stato, tutti i suoi osservatori (Observers) vengono notificati e automaticamente aggiornati.

Questo pattern ci è stato utile per poter mettere in atto due funzionalità diverse: l'invio di messaggi al cliente via via che lo stato dell'ordine si aggiorna e gestire il rifornimento dei magazzino tramite *Supplier* in base alle quantità di articoli presenti. Nel programma sono state implementate due interfacce: *Subject* e *Observer*

Listing 1: Interfaccia Observer

```
public interface Observer {  
    void update(Object obj);  
}
```

Listing 2: Interfaccia Subject

```
public interface Subject {  
    void notify(Object obj);  
    void subscribe(Observer o);  
    void unsubscribe(Observer o);  
}
```

In questo caso l'Observer è stato realizzato in modalità push, cioè ottenendo direttamente l'oggetto richiesto, infatti il metodo `update()` dell'interfaccia Observer ha come parametro Object che corrisponde al dato di interesse.

Dal momento che il pattern è stato utilizzato per due funzionalità diverse, nel programma si trovano due diverse implementazioni delle interfacce Subject e Observer.

Nel caso della gestione di messaggi al cliente il *ConcreteSubject* è **Order**, osservato dal *ConcreteObserver* **CustomerNotifier**. Sarà quest'ultimo infatti a essere notificato una volta che avviene un cambiamento nello stato dell'ordine. Quando si verifica ciò, il *CustomerNotifier* provvederà a inviare un messaggio nella inbox del cliente che ha effettuato l'ordine.

Nel caso invece della gestione del rifornimento del magazzino, il *ConcreteSubject* è la classe **Storage** mentre il *ConcreteObserver* è la classe **Supplier**. Lo scopo è assicurarci che le quantità di fiori e decorazioni presenti nel magazzino non oltrepassino una soglia minima. Nel caso in cui questo accada, sarà lo *Storage* a notificare il *Supplier* che necessita il rifornimento di un dato elemento tramite il metodo `notify()` che passa come parametro il nome dell'elemento in questione. Infine il *Supplier*, che è stato così allertato, eseguirà il

metodo `update()` in modalità `push`, che prevede l'invocazione del metodo `sendItem()` che provvederà a inviare un quantitativo fissato degli elementi richiesti.

4.1.2 State

L'utilizzo dello `State Pattern` ci ha permesso di implementare dei menù, che rendono più fluido l'accesso alle funzionalità del Programma. Nella classe *Program* è presente un'istanza di *Menu*, la quale viene riassegnata ogni qual volta sia necessario passare da un menù ad un altro.

Listing 3: Metodo `show()` dell'interfaccia *Menu*

```
public interface Menu {  
    void show();  
}
```

I principali **concreteState** che abbiamo creato sono *CustomerMenu* e *FloristMenu*. Essi ci permettono di creare due "interfacce" diverse a seconda che l'utente sia un cliente o un fioraio. Inoltre il menù spesso cambia a seguito di una decisione dell'utente, quindi a seguito di un input da tastiera, rendendo l'esecuzione del programma dinamica.

Nel *CustomerMenu* possiamo piazzare un ordine, passare all'*InboxMenu* e visualizzare gli ordini già fatti.

Listing 4: Metodo `show` della classe *CustomerMenu*

```
public class CustomerMenu implements Menu{  
  
    @Override  
    public void show() {  
        int menuItem;  
        boolean exit = false;  
        Customer currentCustomer = null;  
  
        if(Program.getInstance().getCurrentUser() instanceof Customer){  
            currentCustomer = (Customer)  
Program.getInstance().getCurrentUser();  
        }  
        else {  
            Program.getInstance().getCurrentUser().setLogged(false);  
            Program.getInstance().setMenu(new LoginMenu());  
        }  
        do {  
            System.out.println("Choose an option below:");  
            System.out.println("1: View my orders");  
            System.out.println("2: Place an order");  
            System.out.println("3: Inbox");  
        }  
    }  
}
```

```

        System.out.println("0: Logout");
        Scanner input = new Scanner(System.in);
        menuItem = input.nextInt();

        switch (menuItem) {
            case 1:
                System.out.println("Here's your order list: \n");
                OrderList.getInstance().printCustomerOrders(currentCustomer);
                break;

            case 2:
                currentCustomer.placeOrder();
                System.out.println("Order placed successfully.");
                break;

            case 3:
                System.out.println("Welcome to your Inbox.");
                Program.getInstance().setMenu(new InboxMenu(currentCustomer));
                exit = true;
                break;

            case 0:
                currentCustomer.setLogged(false);
                exit = true; Program.getInstance().logout();
                Program.getInstance().setMenu(new LoginMenu());
                System.out.println("Logged out successfully. Bye
                bye! \n");
                break;

            default:
                System.err.println("Invalid input.");
        }
    } while (!exit);
}

```

Nel *FloristMenu* invece possiamo completare un ordine, visualizzare la lista degli ordini, e visualizzare le disponibilità del magazzino.

Listing 5: Metodo show() di FloristMenu

```

public class FloristMenu implements Menu{

    @Override
    public void show() {

```



```

    int menuItem;
    boolean logout = false;
    Florist currentFlorist = null;

    if(Program.getInstance().getCurrentUser() instanceof Florist){
        currentFlorist = (Florist)
Program.getInstance().getCurrentUser();
    }
    else {
        Program.getInstance().getCurrentUser().setLogged(false);
        Program.getInstance().setMenu(new LoginMenu());
    }
    do{
        System.out.println("Choose an option below:");
        System.out.println("1. Fill order");
        System.out.println("2: View order list");
        System.out.println("3: View storage");
        System.out.println("0: Logout");
        Scanner input = new Scanner(System.in);
        menuItem = input.nextInt();

        switch(menuItem){
            case 0:
                currentFlorist.setLogged(false);
                logout = true;
                Program.getInstance().logout();
                Program.getInstance().setMenu(new LoginMenu());
                System.out.println("Logged out successfully. Bye
bye! \n");
                break;

            case 1:
                currentFlorist.fillOrder();
                break;

            case 2:
                OrderList.getInstance().displayOrders();
                break;

            case 3:
                currentFlorist.checkStorage();
                break;

            default:
                System.err.println("Invalid input.");

```

```

        }
    }
    while(!logout);
}
}

```

4.1.3 Singleton

Il pattern Singleton è un pattern di tipo creazionale che garantisce che una classe sia dotata di un punto di accesso globale e abbia una sola istanza, ciò è possibile tramite un riferimento statico. A livello implementativo si è sfruttato questo pattern per tre classi: *Program*, *OrderList* e *Catalog*. In tutti e tre i casi il Singleton ci ha permesso di assicurarsi che esistesse esattamente solo una istanza della classe e che fosse facilmente accessibile.

4.1.4 Composite

Il Composite pattern si è rivelato utile per la struttura dell'ordine: era necessario poter trattare i Bouquet (composizioni di fiori e decorazioni) allo stesso modo dei fiori e delle decorazioni. La chiave di questo pattern è una classe astratta **Product** che rappresenta sia i tipi "leaf" sia le loro composizioni. Ovvero, *Product* ha lo scopo di fornire un'interfaccia comune per articoli singoli cioè *Flower* e *Decoration* e per articoli composti come *Bouquet*. Il metodo che permette di ottenere i singoli elementi della composizione è `getItem()`: tramite un indice scorre la lista di Product da ritornare.

Listing 6: Metodo getItem()

```

public class Bouquet extends Product{
[...]
    public Product getItem(int i) {
        return childProduct.get(i);
    }
}

```

4.2 Gestione dei CSV

Nell'elaborato i CSV si sono rivelati utili per memorizzare dati che era necessario recuperare al successivo avvio del programma. Tra questi si trovano quelli relativi agli utenti (email, nome, cognome, password, indirizzo...), necessari per poter permettere un login e uno storico degli ordini associato all'utente, ma non solo. Abbiamo infatti creato più CSV per poter salvare anche le quantità di ogni prodotto immagazzinate nello storage, gli ordini, così da poter avere uno storico, il catalogo, così da tenere traccia non solo del prezzo di ogni prodotto ma anche della loro "ricetta", ed infine i messaggi, così da avere anche uno storico delle notifiche. Di seguito mostriamo come abbiamo implementato i metodi di scrittura su/lettura da CSV prendendo in esame quelli della classe *Program*.

Listing 7: Metodo initUsers()

```
public void initUsers() {
    String pathToCSV = "users.csv";
    try {
        CSVReader reader = new CSVReader(new
FileReader(pathToCSV));
        List<String[]> csvBody = reader.readAll();
        for (String[] strings : csvBody) {
            if (strings[0].equals("florist")) {
                Florist f = new Florist(strings[1], strings[2],
strings[3], strings[4], strings[5], false );
                users.add(f);
            }
            if (strings[0].equals("customer")) {
                Customer c = new Customer(strings[1], strings[2],
strings[3], strings[4], strings[5], false );
                users.add(c);
            }
        }
        reader.close();
    } catch (Exception e) {
        System.err.println("Error: init on Program while reading
csv.");
    }
}
```

Listing 8: Metodo writeUserOnCSV()

```
public void writeUserOnCSV(String category, User currentUser) {
    String pathToCSV = "users.csv";
    try {
        CSVReader reader = new CSVReader(new
FileReader(pathToCSV));
        List<String[]> csvBody = reader.readAll();
        String[] newuser = {category, currentUser.getEmail(),
currentUser.getName(), currentUser.getSurname(),
currentUser.getAddress(),
currentUser.getHashPass(), String.valueOf(currentUser.getId())};
        csvBody.add(newuser);
        reader.close();

        CSVWriter writer = new CSVWriter(new
FileWriter(pathToCSV));
        writer.writeAll(csvBody);
        writer.flush();
    }
}
```

```

        writer.close();
    } catch (Exception e) {
        System.err.println("Error: Csv Exception.");
    }
}

```

4.3 Metodi Principali

4.3.1 Fill Order

Il metodo `fillOrder()` appartiene alla classe ***Florist***, ed è uno dei metodi principali in quanto al suo interno si procede alla creazione effettiva della scatola e al suo riempimento. All'interno di questo metodo infatti si guarda il contenuto dell'ordine al fine di ricreare i prodotti specificati; per fare ciò sfruttiamo il metodo `getComponents()` della classe *Order*. Qualora fosse presente un bouquet si procede alla sua scomposizione tramite il metodo `getItem()` della classe *Bouquet*.

A seguire si procede prendendo gli articoli necessari dal magazzino con il metodo `getItem()` della classe *Storage* e al loro inserimento nella scatola con il metodo `pack()` della classe *Box*. Per quanto riguarda i Bouquet, prima bisogna ricreare la composizione richiesta e poi si può procedere con l'inserimento.

Una volta aggiunti tutti i prodotti richiesti, la scatola viene chiusa con il metodo `close()` e poi spedita con `sendOrder()`.

Listing 9: Metodo `fillOrder()`

```

public Box fillOrder() {
    pickOrder();
    System.out.println("The order is being processed.");
    if (currentorder != null) {
        Box box = new Box(currentorder);
        for (Product i : currentorder.getComponents()) {
            if (i instanceof Flower || i instanceof Decoration) {
                box.pack(s.getItem(i.getName()));
            } else {
                Bouquet b = new Bouquet(i.getName());
                for (int itr = 0; itr < ((Bouquet) i).getSize();
itr++) {
                    Product tmp = ((Bouquet) i).getItem(itr);
                    b.addItem(s.getItem(tmp.getName()));
                }
                box.pack(b);
            }
        }
        box.close();
        currentorder.setComplete(true);
    }
}

```

```

        System.out.println("The order #" + currentorder.getId() +
" is completed.");
        sendOrder(box);
        OrderList.getInstance().refreshCSV(currentorder);
        return box;
    } else {
        System.out.println("There are no orders to process.");
    }
    return null;
}

```

4.3.2 Place Order

Il metodo `placeOrder()`, appartenente alla classe *Customer*, permette al cliente di scegliere gli articoli dal catalogo e di creare un ordine con gli articoli selezionati. Dopo aver visualizzato il catalogo, il metodo `chooseProduct()` legge gli input inseriti da tastiera che identificano i prodotti scelti. Dopodiché chiamando il metodo `createOrder()`, sempre della classe *Customer*, viene creato l'ordine con i prodotti corrispondenti agli input inseriti precedentemente. L'ordine viene poi inserito nella lista degli ordini.

Listing 10: Metodo `placeOrder()`

```

public void placeOrder() {
    Catalog.getInstance().displayFloristCatalog();
    boolean done = false;
    ArrayList<Integer> productlist = new ArrayList<>();
    while(!done) {
        productlist.add(chooseProduct());
        System.out.println("Product added to the order. Would you
like to add any other item? (y/n)");
        Scanner input = new Scanner(System.in);
        char c = input.findInLine(".").charAt(0);
        if (c == 'n') {
            done = true;
        }
    }
    createOrder(productlist);
}

```

4.3.3 Login e Sign Up

Nell'elaborato abbiamo inserito un login, questo ci ha permesso di associare all'utente i propri ordini e messaggi. Per implementarlo è stato necessario introdurre un tipo di crittografia (nel nostro caso molto semplice) che ci facesse ottenere un encrypt della password

inserita, da confrontare poi con l'encrypt delle password inserite al momento dell'accesso. Per fare ciò abbiamo importato la classe *MessageDigest* [2] che incorpora vari algoritmi di crittografia, tra cui SHA-256, che è quello che abbiamo deciso di utilizzare. Per poter eseguire l'accesso è però necessario essere registrati, infatti se si inserisce un email non memorizzata il programma reindirizza alla registrazione. Dopo aver inserito le informazioni necessarie si accede senza dover rieseguire il login. Di seguito troverete i metodi utilizzati nel metodo `show()` della classe *LoginMenu*, ovvero `login()`, `signUp()` e `encode()`.

Listing 11: Metodi `login()` e `signUp()` della classe *Program*

```
public class Program{
[...]
```

```
    public boolean login(String email, String encoded) {
        boolean b = false;
        for (User u : users) {
            if (u.getEmail().equals(email)) {
                if (Objects.equals(u.getHashPass(), encoded)) {
                    System.out.println("Successfully logged in.");
                    u.setLogged(true);
                    currentUser = getUser(email);
                    System.out.println("Welcome to the Flower Shop, "
+ currentUser.getName() + "!");
                    b = true;
                } else {
                    System.out.println("Wrong password!");
                    currentUser = null;
                    b = false;
                }
            }
        }
        return b;
    }
}
```

```
    public void signUp(String category, String email, String name,
String surname, String address, String encoded) {
        if (category.equals("customer")) {
            currentUser = new Customer(email, name, surname, address,
encoded, true);
            users.add(currentUser);
        }
        if (category.equals("florist")) {
            currentUser = new Florist(email, name, surname, address,
encoded, true);
            users.add(currentUser);
        }
    }
}
```

```
    }  
    writeUserOnCSV(category, currentUser);  
}
```

Listing 12: Metodo encode() della classe LoginMenu

```
public class LoginMenu implements Menu{  
[...]  
    public String encode(String password){  
        md.update(password.getBytes());  
        byte[] messageDigestSHA256 = md.digest();  
        StringBuilder stringBuffer = new StringBuilder();  
        for (byte bytes : messageDigestSHA256) {  
            stringBuffer.append(String.format("%02x", bytes & 0xff));  
        }  
        return stringBuffer.toString();  
    }  
}
```

5 Unit Test

Per quanto riguarda lo Unit Testing abbiamo sfruttato JUnit. Abbiamo testato vari metodi di più classi, con lo scopo di verificare il corretto funzionamento delle principali operazioni del programma.

Tutte le classi test hanno una funzione chiamata `setUp()` che legge i CSV rilevanti per la classe test in esame.

5.1 Customer Test

Si è scelto di testare alcune funzionalità della classe *Customer* ovvero creare un ordine e cancellare i messaggi dalla inbox di un cliente.

In `testCreateOrder()` viene creato un ordine con 5 prodotti casuali e inserito in *OrderList*. Si controlla poi che le componenti dell'ordine, compresa la composizione degli eventuali bouquet, sia corretta.

In `testClearInbox()` si vuole controllare che l'inbox venga effettivamente ripulito, quindi si verifica che la dimensione iniziale dell'inbox sia diversa da 0 e che, successivamente alla chiamata del metodo `clearInbox()`, la lista dei messaggi sia vuota.

Listing 13: Customer tests

```
@Test
@DisplayName("Test that order has the right size and that it has
the same products of the last order in orderlist")
void testCreateOrder() {
    ArrayList<Integer> testnum = new ArrayList<>();
    int num;
    for(int i = 0; i < 5; i++) {
        num = (int) (Math.random() * 15);
        testnum.add(num);
    }
    c.createOrder(testnum);
    Order o = OrderList.getInstance().getLastOrder();
    assertNotNull(o);
    assertEquals(c, o.getCustomer());
    assertEquals("Processing", o.getStatus());
    for(int i = 0; i < o.getComponents().size(); i++)
        assertEquals(Catalog.getInstance().getFloristProduct(
            testnum.get(i)).getName(),
            o.getComponents().get(i).getName());
}

@Test
void testClearInbox() {
    assertNotEquals(0, c.getInboxSize());
}
```



```

        c.clearInbox();
        assertEquals(0, c.getInboxSize());
    }

```

5.2 Florist Test

Questa classe test ci permette di testare alcuni tra i metodi principali della classe *Florist*, quali la creazione di un ordine e il suo invio al cliente.

`testFillOrder()` si occupa di completare un ordine e di impacchettarlo per poi controllare che i prodotti contenuti nel pacco corrispondano effettivamente a quelli richiesti dal cliente nell'ordine.

`testSendOrder()` controlla lo stato dell'ordine dopo la chiamata a determinati metodi. Infatti dopo aver creato e chiuso la scatola, oltre ad assicurarsi che questa non sia nulla, chiama il metodo `sendOrder()` e controlla che lo stato dell'ordine sia aggiornato su "Delivered". In aggiunta si controlla anche che la scatola sia chiusa, quindi complessivamente il test in questione verifica anche che il setter funzioni correttamente.

Listing 14: Florist tests

```

@Test
@DisplayName("Test that checks that products of the order are correct")
void testFillOrder() {
    Box b = f.fillOrder();
    assertNotNull(b);
    ArrayList<Product> products = b.getItems();
    Order o = b.getOrder();
    ArrayList<Product> components = o.getComponents();
    int i = 0;
    for(Product p : products){
        assertEquals(components.get(i).getName(), p.getName());
    }
}

@Test
@DisplayName("Test that checks order status")
void testSendOrder() {
    Box b = new Box(o);
    b.close();
    assertNotNull(b);
    f.sendOrder(b);
    assertTrue(b.isClosed());
    assertEquals("Delivered", o.getStatus());
}

```

5.3 OrderList Test

I test di questa classe ci permettono di verificare se *OrderList* aggiunge correttamente i nuovi ordini creati alla lista e se effettivamente la funzione `getOrder()` ritorna il primo ordine non ancora completato, quindi si verifica che lo status e il completamento dell'ordine siano rispettivamente "Processing" e false .

Listing 15: OrderList tests

```
@Test
@DisplayName("Test that checks if order has been added to
orderlist")
void testPutOrder() {
    Order o = new Order(c);
    int test = OrderList.getInstance().getSize();
    OrderList.getInstance().putOrder(o);
    assertEquals(o, OrderList.getInstance().getLastOrder());
    assertEquals(test+1, OrderList.getInstance().getSize());
}

@Test
@DisplayName("Test that checks if getOrder actually returns the
last")
void testGetOrder() {
    Order o = OrderList.getInstance().getOrder();
    assertNotNull(o);
    assertFalse(o.isComplete());
    assertEquals("Processing", o.getStatus());
}
```

5.4 Program Test

I test principali della classe *Program* verificano che i metodi riguardanti il login, la registrazione e il logout funzionino correttamente. `testLoginCorrectPassword()` verifica che con la password corretta si effettui il login, mentre invece `testLoginWrongPass()` controlla che, con l'inserimento di una password sbagliata, il login non venga effettuato. `testSignUp` supervisiona la registrazione assicurandosi che, dopo la registrazione di un utente fittizio, il login eseguito successivamente avvenga in maniera corretta. `testLogout()` esamina l'attributo `currentUser` dopo aver chiamato il metodo `logout()` e verifica che esso sia nullo.

Listing 16: Program tests

```

@Test
@DisplayName("Test that checks log in when a correct password is
entered")
void testLoginCorrectPassword() {
    boolean b =
Program.getInstance().login("pippobaudo@gmail.com",
lm.encode("pippo"));
    assertTrue(b, "La password inserita in realta' e' sbagliata.");
}

@Test
@DisplayName("Test that checks log in when an invalid password is
entered")
void testLoginWrongPass() {
    boolean b =
Program.getInstance().login("pippobaudo@gmail.com",
lm.encode("rail"));
    assertFalse(b, "La password inserita in realta' e' giusta");
}

@Test
@DisplayName("Test that registers and logs in a new user")
void signUp() {
    Program.getInstance().signIn("customer",
"mikebuongiorno@rail.it", "mike", "buongiorno",
"via vai di mike buongiorno", lm.encode("allegria"));

    boolean b =
Program.getInstance().login("mikebuongiorno@rail.it",
lm.encode("allegria"));
    assertTrue(b, "La password inserita in realta' e' sbagliata.");
}

@Test
@DisplayName("Test that checks if the current user is null after
logging out")
void testLogout() {
    Program.getInstance().logout();
    assertNull(Program.getInstance().getCurrentUser());
}

```

5.5 Storage Test

Infine, in questa classe test sono stati verificati due metodi della classe *Storage*: `getItem()` e `refresh()`. Nel primo test (`testGetItem()`) si controlla che il prodotto restituito sia effettivamente quello che avevamo richiesto. Per verificare invece il metodo `refresh()`, che compare in `getItem()`, ci affidiamo a `testRefresh()` il quale verifica che una volta prelevato un articolo dal magazzino, venga aggiornata la quantità disponibile di tale articolo.

Listing 17: Storage tests

```
@Test
@DisplayName("Test that checks if the returned value is a product
name")
void testGetItem() {
    Product p = storage.getItem("lily");
    assertNotNull(p);
    assertEquals(p.getName(), "lily");
}

@Test
@DisplayName("A storage test that checks if the quantity is
updated")
void testRefresh() {
    int initial_q = storage.getQuantity("lily");
    storage.getItem("lily");
    assertEquals(storage.getQuantity("lily"), initial_q-1);
}
```

Fonti aggiuntive

- [1] Opencsv Users Guide
<http://opencsv.sourceforge.net/>
- [2] MessageDigest
<https://docs.oracle.com/javase/7/docs/api/java/security/MessageDigest.html>
- [3] JUnit 5
<https://junit.org/junit5/docs/current/user-guide/>