

# Applicativo per la gestione di un'azienda specializzata nella vendita di mobili

Gian Maria Gennai & Christian Di Buò

Elaborato per il corso di Ingegneria del Software.



Ingegneria Informatica  
Università degli studi di Firenze

# Indice

<b>1 Progettazione</b>	<b>2</b>
1.1 Statement funzionale . . . . .	2
1.2 Processo per lo sviluppo del progetto . . . . .	2
1.3 Use case principali . . . . .	3
1.3.1 Use Case templates . . . . .	5
1.3.2 Mockups . . . . .	7
1.4 Class Diagram completo . . . . .	9
1.5 Elenco delle classi . . . . .	10
<b>2 Implementazione</b>	<b>11</b>
2.1 Generalità sull'applicazione . . . . .	11
2.2 Pattern utilizzati . . . . .	11
2.2.1 Singleton . . . . .	11
2.2.2 Observer . . . . .	12
2.2.3 Composite . . . . .	13
2.2.4 Builder . . . . .	14
2.3 CatalogPackage . . . . .	15
2.3.1 CatalogWorker . . . . .	15
2.3.2 Catalog . . . . .	16
2.3.3 AbstractProduct, CompoundProduct, ElementaryService	16
2.4 Invoice package . . . . .	17
2.4.1 Cart . . . . .	17
2.4.2 User . . . . .	18
2.4.3 Invoice InvoiceBuilder . . . . .	18
2.4.4 Builder . . . . .	18
2.5 WarehousePackage . . . . .	19
2.5.1 Warehouse . . . . .	19
2.5.2 WarehouseWorker . . . . .	19
2.5.3 Logistic . . . . .	20
2.5.4 ConcreteProduct, ConcreteCompoundProduct, ConcreteE- lementaryService . . . . .	20
2.5.5 Courier . . . . .	21
<b>3 Testing</b>	<b>22</b>
3.1 Test strutturali . . . . .	22
3.1.1 CartTest . . . . .	22
3.1.2 CatalogTest . . . . .	22
3.1.3 LogisticTest . . . . .	24
3.1.4 UserTest . . . . .	24
3.1.5 SystemTest . . . . .	24
<b>4 Sequence Diagrams</b>	<b>25</b>
<b>5 Statement funzionale completo</b>	<b>28</b>

# 1 Progettazione

## 1.1 Statement funzionale

Lo scopo di questo progetto è quello di realizzare un applicativo software per un'azienda specializzata nella vendita al dettaglio di mobili e altra oggettistica per la casa.

Il *WarehouseWorker* aggiunge nel magazzino i prodotti disponibili per la vendita.

Il *CatalogWorker* inserisce i prodotti, o una loro composizione, nel catalogo.

Nel *catalogo* troviamo tutti i prodotti che possono essere visti e acquistati dall'utente.

Un *utente*, cui è associato un ID temporaneo, effettua delle ricerche nel catalogo e può aggiungere/rimuovere i vari prodotti dal carrello.

L'utente può completare l'acquisto creando un ordine, con una fattura che funge da sommario. Dovrà inoltre inserire il suo indirizzo, scegliere un metodo di pagamento e se desidera che la consegna venga effettuata a domicilio oppure con ritiro in negozio.

Una volta che l'utente ha completato l'ordine *Logistic* riceve la relativa fattura, rimuove i prodotti acquistati dal magazzino e, successivamente, seleziona uno o più corrieri assegnandogli l'ordine che sarà poi consegnato all'utente.

Nell'idea iniziale il progetto era più articolato, con varie feature in più, pertanto per sviluppi futuri si rimanda allo statement completo nella sezione 5.

## 1.2 Processo per lo sviluppo del progetto

Per lo sviluppo di questo progetto siamo partiti da un brainstorming generale sul funzionamento di un ipotetica azienda che produce e vende prodotti. Per fare questo abbiamo chiesto informazioni a persone aventi un lavoro all'interno di questo settore e abbiamo controllato siti web di grandi aziende per renderci conto di come un utente può interfacciarsi.

Dopo aver raccolto le informazioni necessarie abbiamo iniziato a creare una bozza della struttura su carta portando avanti il *Class Diagram* e lo *Use Case Diagram* per poi passare a creare la struttura vera a propria grazie a **StarUML**(software per lo sviluppo uml sviluppato da *MKLabs Co. Ltd.*).

Dopo questa fase preliminare abbiamo iniziato ad implementare tutta la struttura in linguaggio *Java* utilizzando l'IDE **Eclipse**(sviluppato da *Apache Software Foundation* e *Oracle Corporation*), e qui i tempi si sono un po' dilungati dato che abbiamo trovato alcuni problemi lungo il percorso dei quali ci siamo accorti soltanto durante lo sviluppo e che ci hanno riportato anche a rivere i vari schemi fatti in precedenza.

Quando abbiamo chiuso tutta la struttura, assicurandoci che corrispondeva il codice con quello che dicevano i due schemi, allora abbiamo scelto le classi su cui andare ad effettuare i vari test.

### 1.3 Use case principali

Lo *User* 1 può:

1. Vedere i prodotti nel catalogo con le rispettive informazioni.
2. Aggiungere prodotti al carrello.
3. Eliminare prodotti dal carrello.
4. Avere un riassunto dell'ordine fatto.
5. Completare l'ordine.

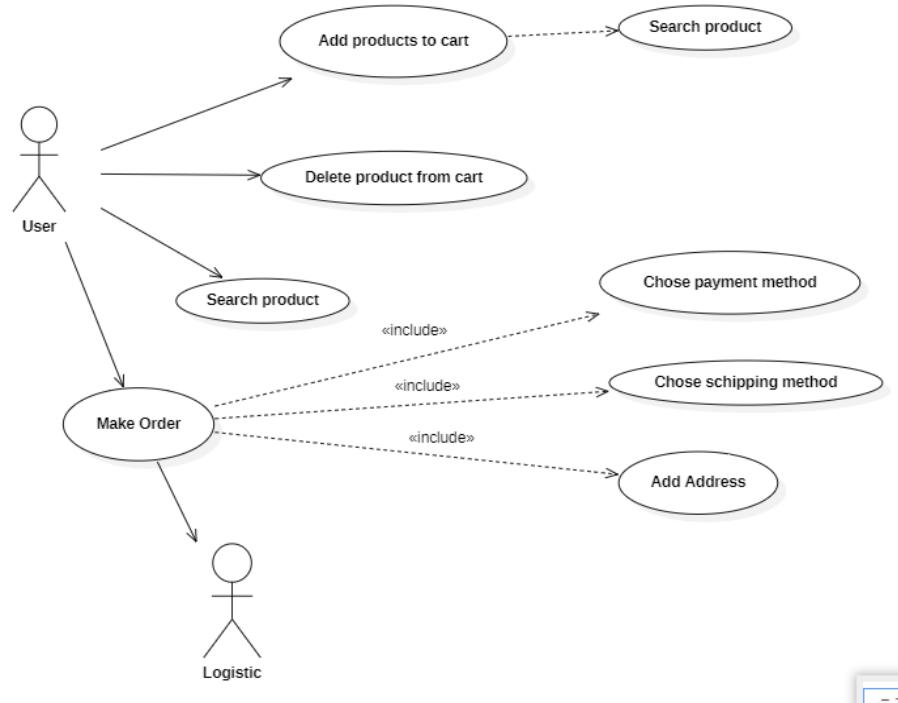


Figura 1: User Use Case Diagram

La classe *Logistic* rappresentato in Figura 2 può:

1. aggiungere fatture relative agli ordini degli utenti.
2. creare spedizioni ed assegnarle ad un corriere.

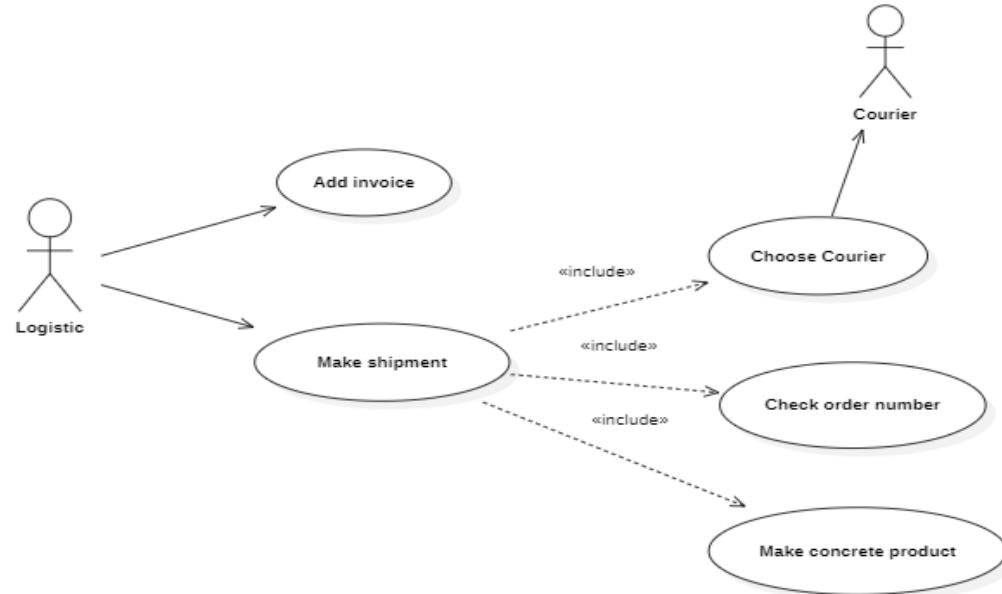


Figura 2: Logistic Use Case Diagram

Il *CatalogWorker* rappresentato in Figura 3 può:

1. Creare prodotti astratti(sia singoli che composti).
2. Aggiungere i prodotti creati al catalogo.
3. Eliminare i prodotti dal catalogo.

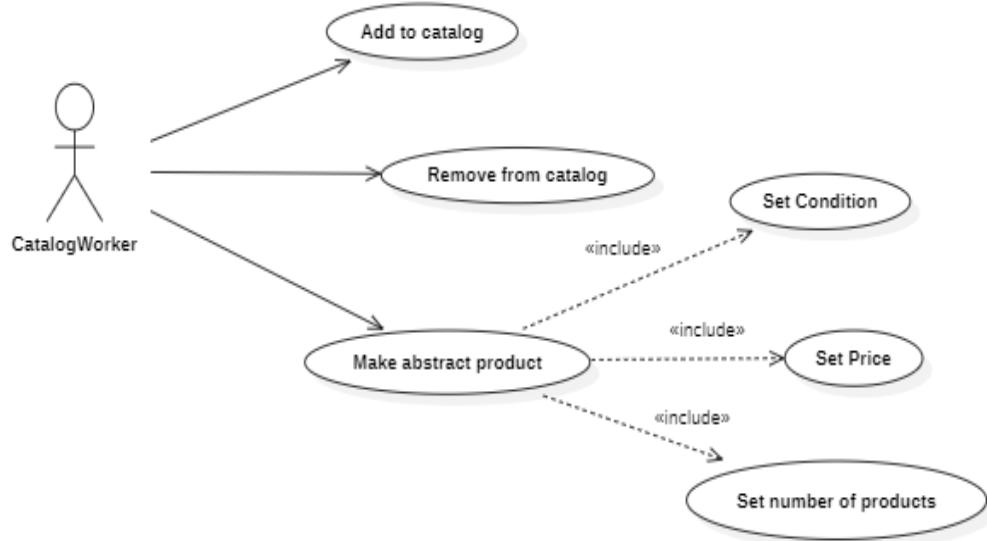


Figura 3: CatalogWorker Use Case Diagram

### 1.3.1 Use Case templates

UC#1	Ricerca Prodotto
Description	Il cliente ricerca un prodotto attraverso l'interfaccia
Level	User Goal
Actor	User (primary), System (secondary)
Basic Course	<ol style="list-style-type: none"> <li>1. il caso inizia quando l'utente ha fatto pressione nel campo digitazione del pannello di ricerca</li> <li>2. l'utente digita il nome da cercare</li> <li>3. l'utente preme il pulsante di avvio ricerca</li> <li>4. il sistema presenta i risultati a schermo</li> </ol>
Alternate Course	<ol style="list-style-type: none"> <li>2a. L'utente abbandona l'area di ricerca</li> <li>3a. Il sistema non trova corrispondenza con il nome inserito.</li> </ol>

<b>UC#2</b>	<b>Inserimento prodotto nel carrello</b>
Description	Il cliente sceglie ed inserisce un prodotto nel carrello
Level	User Goal
Actor	User (primary), System (secondary)
Basic Course	<ol style="list-style-type: none"> <li>1. Lo Use Case inizia quando un utente, dopo aver selezionato il numero di articoli, preme sull'apposito pulsante di aggiunta al carrello relativo ad un prodotto</li> <li>2. Il sistema notifica la corretta aggiunta del prodotto</li> </ol>
Alternate Course	1a. Il sistema non riesce ad aggiungere il prodotto al carrello.

<b>UC#3</b>	<b>Creazione di un ordine</b>
Description	Attraverso l'interfaccia l'utente crea un ordine
Level	User Goal
Actor	User (primary), System (secondary)
Pre-conditions	L'utente deve aver completato almeno una volta lo UC#2
Basic Course	<ol style="list-style-type: none"> <li>1. Il caso inizia quando l'utente preme sull'apposito pulsante "crea ordine"</li> <li>2. Il sistema presenta l'interfaccia di creazione dell'ordine.</li> <li>3. L'utente preme le icone relative alle opzioni per modalità di spedizione e modalità di pagamento.</li> <li>4. L'utente inserisce i campi richiesti.</li> <li>5. L'utente preme il tasto completa ordine.</li> </ol>
Alternate Course	<ol style="list-style-type: none"> <li>5a. Il carrello dell'utente è vuoto</li> <li>5b. L'utente preme il tasto completa ordine, avendo inserito informazioni logicamente errate.</li> </ol>

UC#4	Spedizione merci all'utente
Description	Attraverso l'interfaccia l'utente crea un ordine
Level	System Goal
Actor	Logistic
Pre-conditions	Un utente deve aver completato almeno una volta lo UC#3
Basic Course	<ol style="list-style-type: none"> <li>Il sistema prende in carico la fattura relativa all'ordine di un cliente.</li> <li>Vengono esaminate le merci acquistate.</li> <li>Il sistema affida le merci che devono essere consegnate a casa ai corrieri in base alla loro capacità massima di trasporto.</li> <li>I corrieri prendono in consegna la merce.</li> <li>I corrieri notificano che hanno completato la spedizione.</li> </ol>
Alternate Course	<ol style="list-style-type: none"> <li>L'utente ha selezionato, come modalità di spedizione, ritiro in negozio per tutta la merce.</li> <li>Tutti i corrieri sono occupati.</li> </ol>

### 1.3.2 Mockups

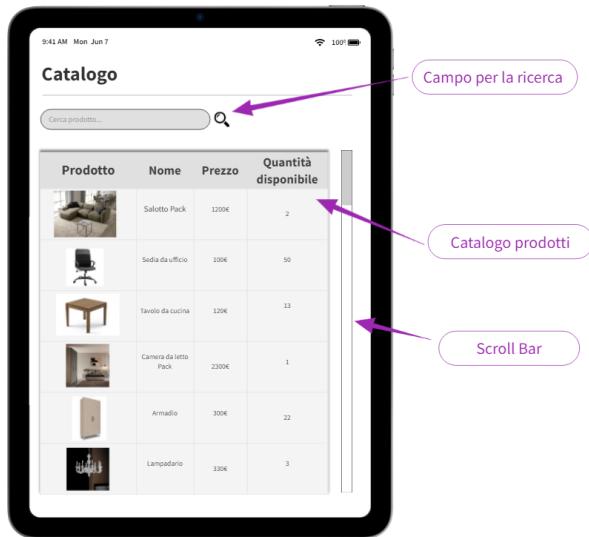


Figura 4: Mockups Catalogo

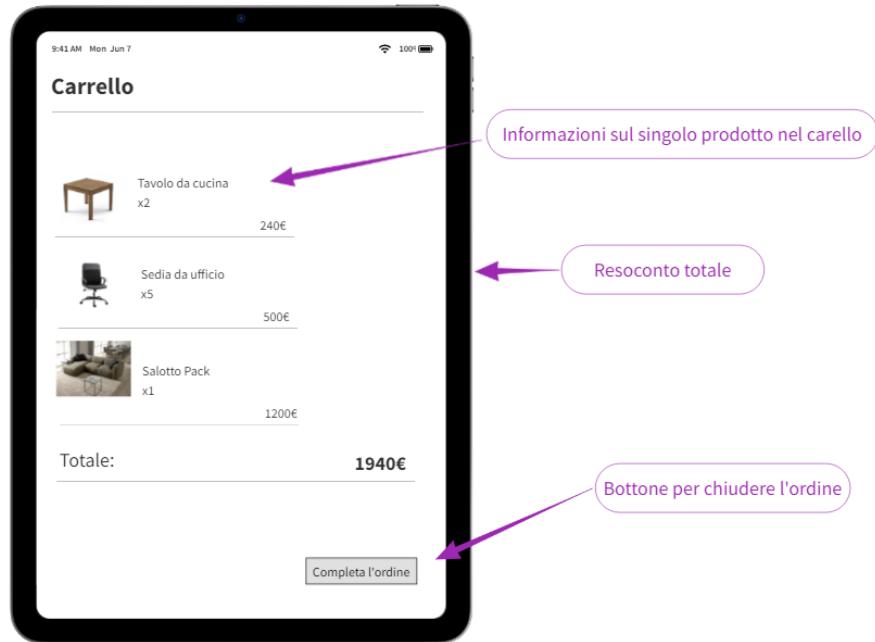


Figura 5: Mockups Carello



Figura 6: Mockups pagina degli Ordini

## 1.4 Class Diagram completo

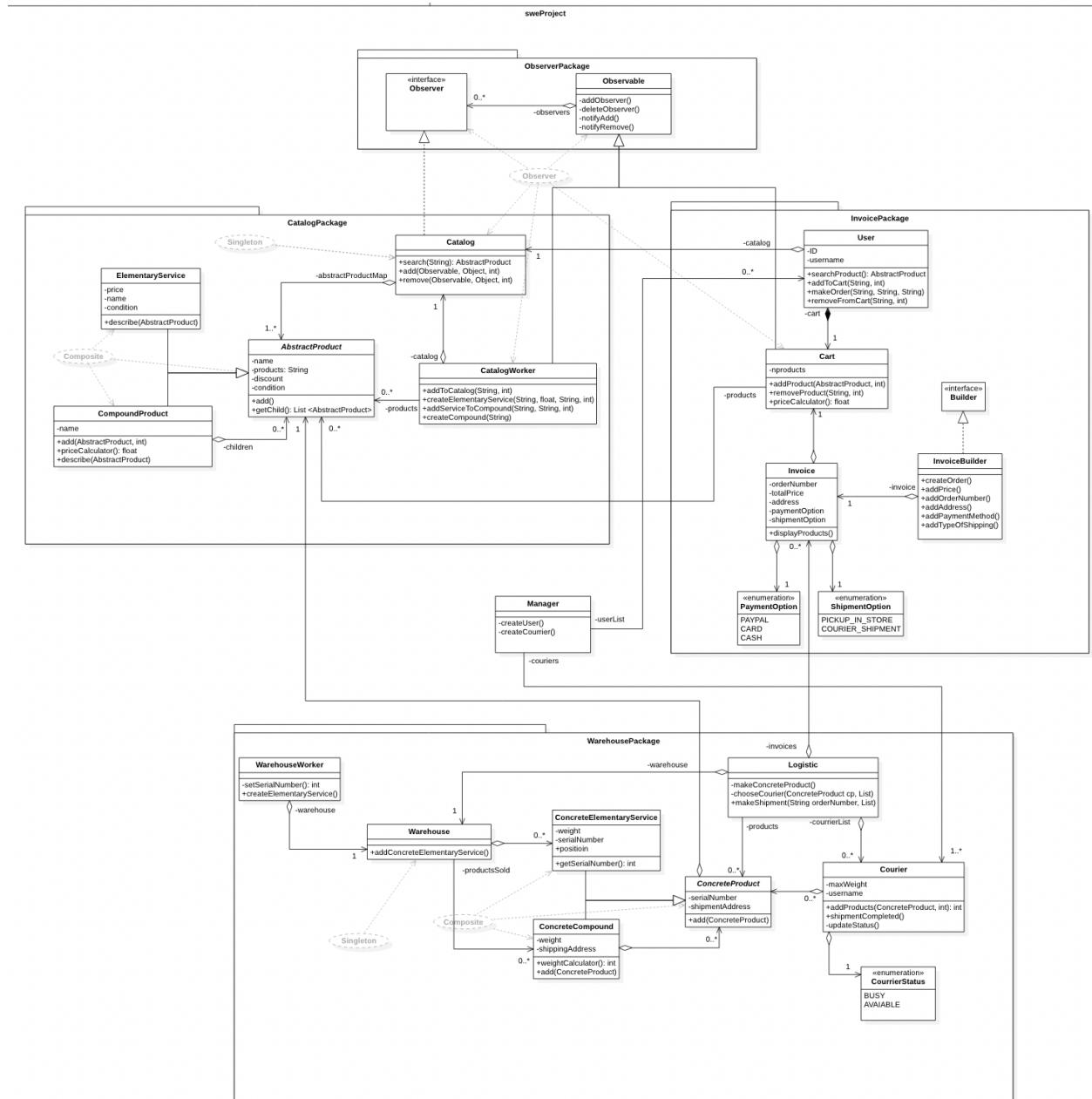


Figura 7: Class diagram completo

## 1.5 Elenco delle classi

In questa sezione spieghiamo in breve quelle che sono le classi più significative, che verranno poi descritte meglio più avanti.

- **Catalog** rappresenta il catalogo contenente tutti i prodotti che l'utente può acquistare.
- **Warehouse** rappresenta il magazzino contenente tutti gli oggetti che possono essere inseriti nel catalogo.
- **AbstractProduct** sono i prodotti astratti che verranno inseriti nel catalogo.
- **ConcreteProduct** sono i prodotti concreti che verranno spediti all'utente.
- **CatalogWorker** rappresenta colui che è in grado di poter aggiungere un prodotto, o una composizione di essi, nel catalogo.
- **WarehouseWorker** rappresenta colui che è in grado di poter istanziare oggetti concreti ed aggiungerli nel magazzino.
- **User** rappresenta un utente, chiunque inizi ad utilizzare l'applicazione rappresenta un utente.
- **Invoice** rappresenta la fattura relativa all'ordine del cliente che verrà inviata alla classe Logistic.
- **Logistic** è la classe che riceve l'ordine del cliente. Decrementa il numero dei prodotti in magazzino, genera l'ordine da consegnare al cliente e, se richiesto da quest'ultimo, seleziona il corriere per la spedizione a domicilio.
- **Courier** rappresenta un corriere, ogni corriere ha una capacità massima di trasporto, cioè un numero massimo di chili.
- **Manager** rappresenta una classe statica, che fa da burrattinaio, ci permette di tenere memoria gli utenti che hanno effettuato degli ordini e dei corrieri che sono assunti.

## 2 Implementazione

### 2.1 Generalità sull'applicazione

L'applicativo è stato scritto utilizzando il linguaggio Java nella versione Java SE 16 e conta di 4 packages contenenti 3 enumerazioni, 24 classi di cui 4 sotto test.

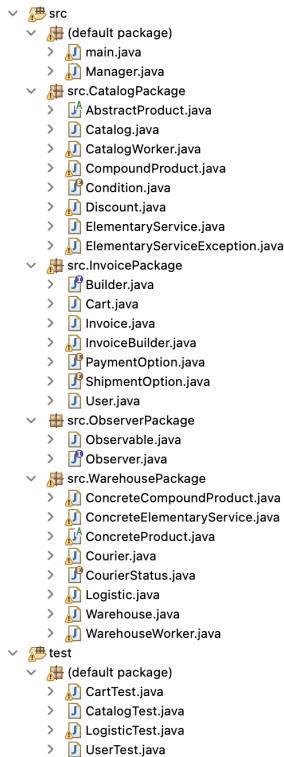


Figura 8: Elenco di tutte le classi

### 2.2 Pattern utilizzati

Prima di addentrarci nello specifico spieghiamo quali pattern abbiamo utilizzato.

#### 2.2.1 Singleton

Il *Singleton* è un idioma/pattern di buona programmazione, in particolare una classe Singleton deve avere una sola istanza e fornire un punto di accesso globale a tale istanza.

Il pattern si applica ponendo privato il costruttore della classe (così che nessuno possa istanziarla a piacere) e usufruendo di un metodo statico getInstance si

può accedere all'unica istanza di questa classe, tale metodo infatti si preoccupa di creare una istanza alla classe se usato per la prima volta, o altrimenti di rendere il riferimento alla stessa.

Nel nostro caso il Singleton è stato utilizzato per implementare le classi Warehouse e Catalog.

### 2.2.2 Observer

L'*Observer* è un pattern tipico comportamentale. Viene utilizzato quando vogliamo mantenere allineato lo stato di due classi.

Noi lo abbiamo usato per tenere sempre aggiornata la lista degli elementi astratti all'interno del catalogo.

La classe Observable e l'interfaccia Observer sono però deprecate in Java 9, uno dei motivi principali è la mancanza di una vasta gamma di servizi, ad esempio: supportano solo l'invio della notizia che qualcosa è cambiato, ma non danno alcuna informazione aggiuntiva. Nel nostro caso questo era un problema poichè dovevamo fornire informazioni su cosa era cambiato, pertanto abbiamo deciso di implementarlo seguendo quella che era il pattern deprecato, ma aggiungendo alcune funzionalità.

Un esempio di utilizzo è nella classe *Cart* dove era necessario incrementare e decrementare il numero di prodotti presenti nel catalogo nel momento in cui l'utente aggiungeva o eliminava prodotti dal suo carrello, altrimenti sarebbero sorte delle incongruenze.

Degli snippet del codice sono visibili nella figura 9.

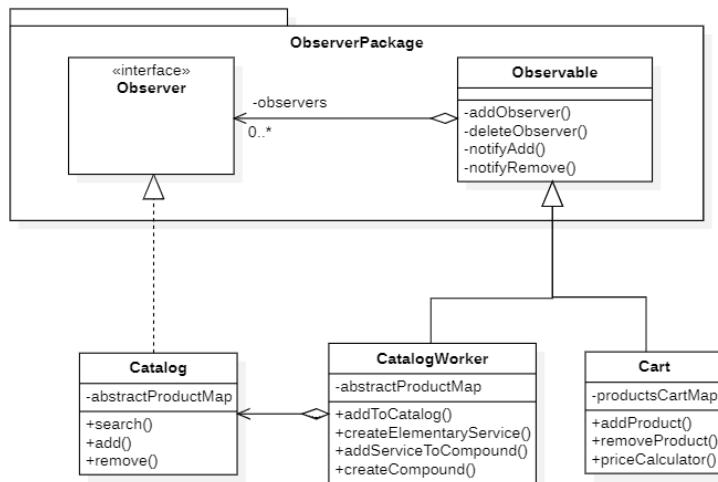


Figura 9: Pattern *Observer*

### 2.2.3 Composite

Il Composite è un pattern strutturale, ovvero un pattern che mi risolve il problema dell'andare ad assemblare un oggetto all'interno di una struttura più complessa, lasciando però tutta la struttura flessibile ed efficiente.

Abbiamo utilizzato questo pattern per creare i vari prodotti da mettere nel catalogo, dato che potevano essere sia singoli che composizioni di prodotti.

Il composite è stato riutilizzato anche nel package Warehouse per generare le merci da spedire all'utente.

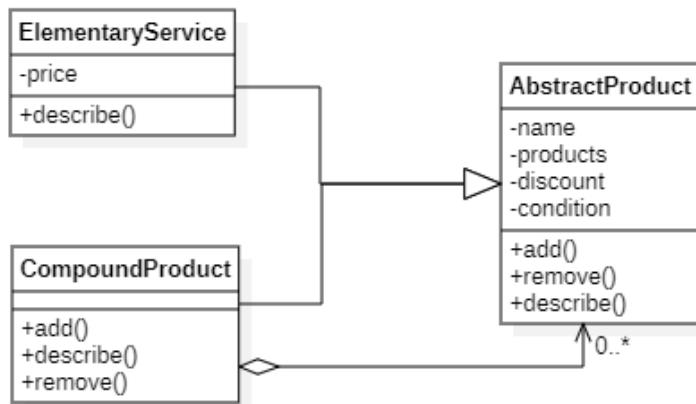


Figura 10: Pattern Composite

#### 2.2.4 Builder

Il Builder è un pattern creazionale, in questa implementazione il Builder è rappresentato in modo semplificato. Separa la costruzione di un oggetto complesso dalla sua rappresentazione, in modo che lo stesso processo di costruzione consenta la creazione di diverse rappresentazioni.

Nella nostra implementazione consiste in una classe, che ha per composizione l'oggetto che deve costruire. La classe presenta poi un insieme di metodi per settare vari parametri del product. Tale classe implementa un'interfaccia Builder che espone come metodo astratto il solo getProduct, imperativo per ogni builder concreto da implementare.

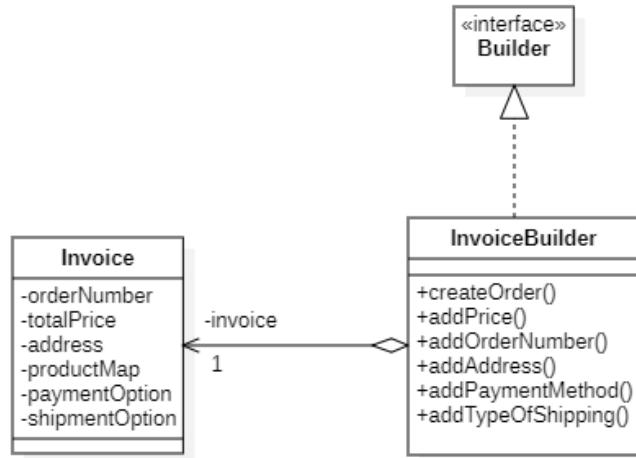


Figura 11: Pattern *Builder*

## 2.3 CatalogPackage

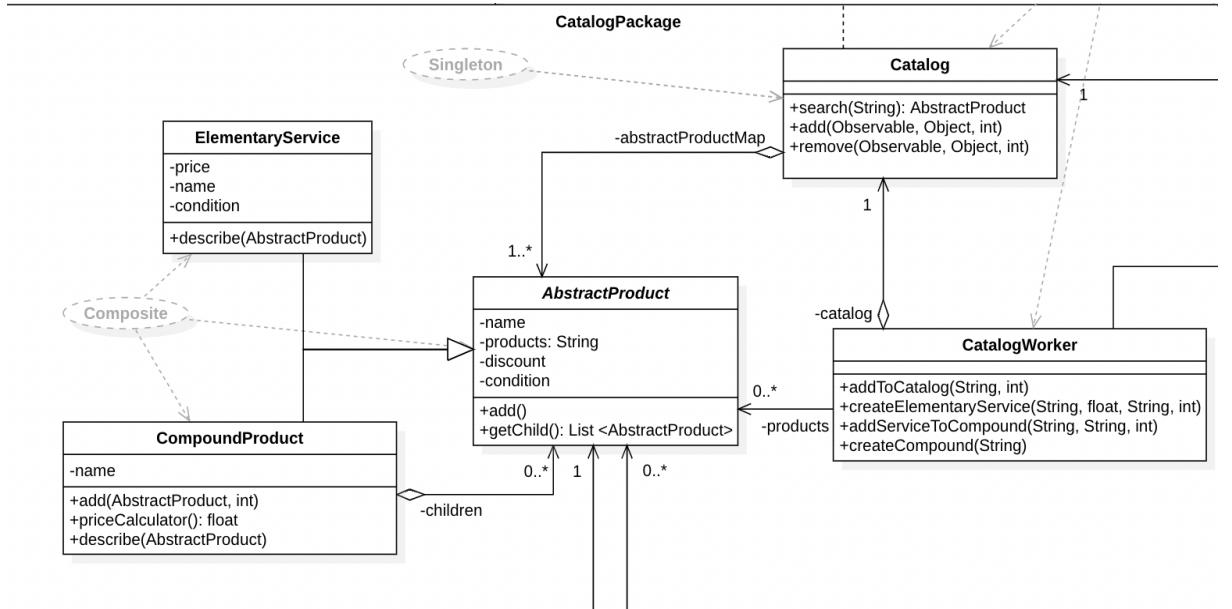


Figura 12: Package del catalogo

### 2.3.1 CatalogWorker

Come accennato rappresenta colui che ha il compito di inserire, ed eventualmente rimuovere, i prodotti astratti nel catalogo.

Nel costruttore possiamo osservare come venga instaurata la connessione con *Observer* (Catalog).

I metodi di rilievo sono:

- *createElementService()*, *createCompoundProduct()*, *addServiceToCompound()*. Questi tre metodi permettono di creare, attraverso l'utilizzo del *Composite*, un prodotto astratto, singolo o composto, da inserire nel catalogo. Il metodo *createElementService()* permette di generare i prodotti singoli(*Leaf*), che tramite il metodo *addServiceToCompound()*, potranno essere aggiunti in un contenitore di componenti creato con il metodo *createCompoundProduct()*.
- *addToCatalog()*: avvia la procedura di allineamento con l'osservatore, l'observer è stato realizzato in modalità push.

```

public AbstractProduct createElementaryService(String nameOfProduct, float price, String condition, int weight) {
    AbstractProduct p = new ElementaryService(nameOfProduct, price, condition, weight);
    abstractProductMap.put(nameOfProduct, p);
    return p;
}
public AbstractProduct createCompoundProduct(String nameOfCompound) {
    abstractProductMap.put(nameOfCompound, new CompoundProduct(nameOfCompound));
    return abstractProductMap.get(nameOfCompound);
}
public void addServiceToCompound(String compoundProduct, String elementaryService, int numberOfElementaryService) throws ElementaryServiceException {
    abstractProductMap.get(compoundProduct).add(abstractProductMap.get(elementaryService), numberOfElementaryService);
}

```

Figura 13: snippet catalog worker

### 2.3.2 Catalog

```

public static Catalog getInstance(){
    if(instance == null)
        instance = new Catalog();
    return instance;
}
public AbstractProduct search(String nameOfProduct) throws NoSuchElementException {
    AbstractProduct tmp = null;
    Object []l = abstractProductMap.keySet().toArray();
    for (int i = 0; i < abstractProductMap.size(); i++) {
        if (Objects.equals((AbstractProductMap)[i].getName(), nameOfProduct)){
            if ((AbstractProductMap)[i].getPrice() == 0) {
                throw new NoSuchElementException("Product not available!");
            }else {
                tmp = ((AbstractProduct)[i]);
            }
        }
        if (tmp == null)
            throw new NoSuchElementException("Don't exist any product with that name!");
    }
    return tmp;
}

```

il metodo *search* permette di cercare un prodotto sulla base del nome, se non ce ne sono viene sollevata un'eccezione.

### 2.3.3 AbstractProduct, CompoundProduct, ElementaryService

```

public float priceCalculator(){
    float totalPrice=0;
    for(int i=0; i<children.size(); i++){
        AbstractProduct p = children.get(i);
        totalPrice += p.getPrice();
    }
    return totalPrice;
}

```

Figura 14: snippet priceCalulator()

AbstractProduct è una classe *astratta* che rappresenta i prodotti che verranno aggiunti nel catalogo. Implementa, con le classi **ElementaryService** e **CompoundProduct**, l'idioma *Composite*. Ogni ElementaryService ha un prezzo e una condizione(enumarazione), nuovo o usato, a cui corrisponde uno sconto, calcolato tramite la classe *Discount*. La classe CompoundProduct, qualora il prodotto sia composto, effettua il calcolo del prezzo complessivo con il metodo *priceCalculator()*, come si vede in figura 14.

Classe realizzata tramite l'idioma *Singleton* essendo unico per tutti gli utenti. I metodi di rilievo sono *add* e *remove*, chiamati dall'Observable(CatalogWorker). *Add* effettua un controllo evitando di aggiungere nel catalogo prodotti omonimi. *Remove* evita che vengano rimossi più prodotti di quanti ce ne sono sollevando eventualmente un'eccezione. Inoltre,

AbstractProduct è una classe *astratta* che rappresenta i prodotti che verranno aggiunti nel catalogo. Implementa, con le classi **ElementaryService** e **CompoundProduct**, l'idioma *Composite*.

Ogni ElementaryService ha un prezzo e una condizione(enumarazione), nuovo o usato, a cui corrisponde uno sconto, calcolato tramite la classe *Discount*.

## 2.4 Invoice package

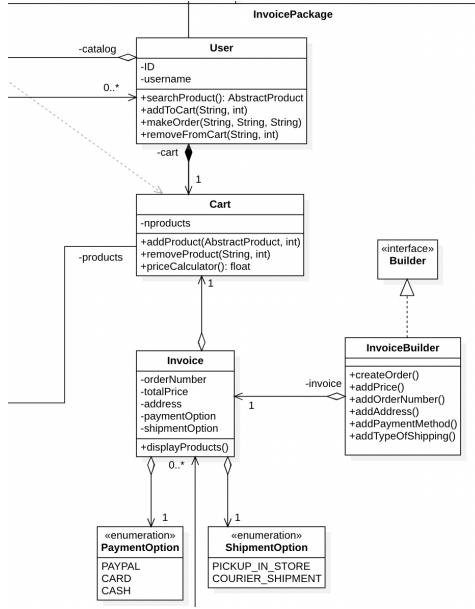


Figura 15: Invoice package

Il seguente package invece viene utilizzato nel momento in cui l'utente inizia ad interagire con il sistema. Qui il cliente decide di controllare quello che è disponibile all'interno del catalogo per poi salvarlo all'interno del carrello che è una sorta di contenitore per i prodotti che si andranno ad acquistare.

### 2.4.1 Cart

```

private Map<AbstractProduct, Integer> abstractProductsMap;
public Cart() {
    Catalog catalog = Catalog.getInstance();
    this.addObserver(catalog);
    abstractProductsMap = new HashMap<>();
}
public Cart(Cart c){
    abstractProductsMap = new HashMap<>();
    for(Map.Entry<AbstractProduct, Integer> entry: c.abstractProductsMap.entrySet()) {
        abstractProductsMap.put(entry.getKey(), entry.getValue());
    }
}
    
```

e *removeProduct*, oltre alla loro funzione classica, rispettivamente rimuovono e aggiungono quel prodotto dal catalogo mantenendolo aggiornato.

La classe *Cart* rappresenta un carrello che colleziona i prodotti attraverso una mappa che ha come chiave gli *AbstractProduct* e come valore il numero dei prodotti nel carrello.

I metodi *addProduct*

```

public void removeProduct(String productName, int productNumber) throws IllegalArgumentException {
    AbstractProduct tmp = null;
    Object[] l = abstractProductsMap.keySet().toArray();
    for (int i = 0; i < abstractProductsMap.size(); i++) {
        if (Objects.equals(((AbstractProduct) l[i]).getName(), productName)) {
            tmp = ((AbstractProduct) l[i]);
            if(abstractProductsMap.remove(tmp, productNumber)) {
                this.notifyAdd(tmp, productNumber);
            }else if (abstractProductsMap.get(tmp) > productNumber) {
                abstractProductsMap.replace(tmp, abstractProductsMap.get(tmp), abstractProductsMap.get(tmp) - productNumber);
                this.notifyAdd(tmp, productNumber);
            } else {
                throw new IllegalArgumentException("You are trying to remove more products at how many are contained in the cart!");
            }
        }
    }
    if (tmp == null)
        throw new IllegalArgumentException("Don't exist any product with that name!");
}

```

Figura 16: snippet classe cart

#### 2.4.2 User

Rappresenta un utente, ogni utente ha un ID univoco. Il metodo makeOrder è abbozzato e incarica l'OrderBuilder di costruire un ordine, associandogli di default il proprio carrello e ID.

Il metodo getter per il carrello effettua la copia difensiva dello stesso.

```

public AbstractProduct createElementaryService(String nameOfProduct, float price, String condition, int weight) {
    AbstractProduct p = new ElementaryService(nameOfProduct, price, condition, weight);
    abstractProductMap.put(nameOfProduct, p);
    return p;
}
public AbstractProduct createCompoundProduct(String nameOfCompound) {
    abstractProductMap.put(nameOfCompound, new CompoundProduct(nameOfCompound));
    return abstractProductMap.get(nameOfCompound);
}
public void addServiceToCompound(String compoundProduct, String elementaryService,int numberOfElementaryService) throws ElementaryServiceException {
    abstractProductMap.get(compoundProduct).add(abstractProductMap.get(elementaryService), numberOfElementaryService);
}

```

Figura 17: snippet classe User

#### 2.4.3 Invoice InvoiceBuilder

La classe *InvoiceBuilder* estende l'interfaccia Builder e permette di creare un oggetto complesso, l'*Invoice* appunto, step-by-step.

La classe *Invoice* aggrega un insieme di attributi che caratterizzano l'ordine del cliente, non ha metodi di particolare rilievo.

#### 2.4.4 Builder

Fa da *Interfaccia* per la classe Builder concreta, espone un solo metodo *getProduct* che deve essere offerto da chi implementa tale classe.

## 2.5 WarehousePackage

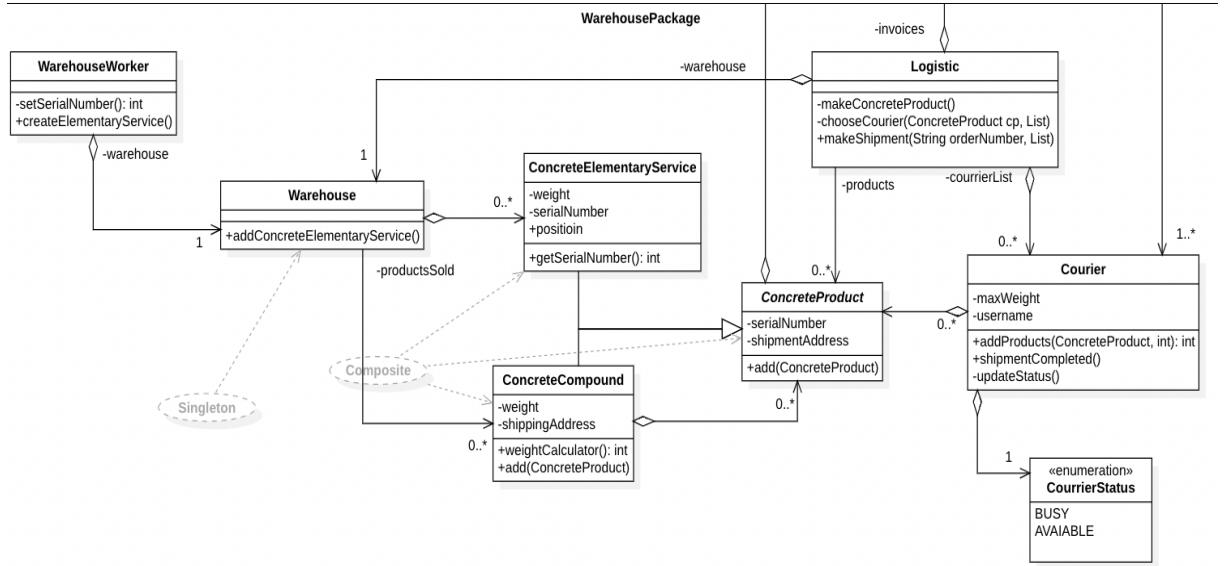


Figura 18: Warehouse package

### 2.5.1 Warehouse

La classe *Warehouse* è realizzata mediante l'idioma Singleton, ha al suo interno una lista di *ConcreteElementaryService* che verranno usati per comporre l'ordine del cliente, tiene memoria anche della merce che i clienti decidono di ritirare in negozio.

```

public AbstractProduct createElementaryService(String nameOfProduct, float price, String condition, int weight) {
    AbstractProduct p = new ElementaryService(nameOfProduct, price, condition, weight);
    abstractProductMap.put(nameOfProduct, p);
    return p;
}
public AbstractProduct createCompoundProduct(String nameOfCompound) {
    abstractProductMap.put(nameOfCompound, new CompoundProduct(nameOfCompound));
    return abstractProductMap.get(nameOfCompound);
}
public void addServiceToCompound(String compoundProduct, String elementaryService, int numberofElementaryService) throws ElementaryServiceException {
    abstractProductMap.get(compoundProduct).add(abstractProductMap.get(elementaryService), numberofElementaryService);
}
    
```

Figura 19: snippet classe warehouse

### 2.5.2 WarehouseWorker

Questa classe rappresenta colui che ha il compito di generare i *ConcreteElementaryService* ed inserirli nel magazzino.

### 2.5.3 Logistic

```

public boolean chooseCourier(ConcreteProduct cp, List<Courier> courierList) {
    int i = 0;
    int weight = cp.getWeight();
    while(i < courierList.size()) {
        if(courierList.get(i).getStatus() == CourierStatus.AVAILABLE) {
            weight = courierList.get(i).addProducts(cp, weight);
            if(weight <= 0)
                return true;
        }
        i++;
    }
    return false;
}

```

Figura 20: snippet classe logistic

È una delle classi di maggior rilievo nel progetto, riceve l'ordine effettuato dal cliente e tramite il metodo *makeShipment* provvederà ad incaricare i corrieri di effettuare la spedizione. Al suo interno vengono utilizzati altri metodi come:

- *chooseCourier* con il quale sceglie il corriere a cui assegnare la spedizione.
- *makeConcreteProduct* permette di creare quelli che sono i prodotti concreti, che verranno dati in consegna ai corrieri.
- *checkOrderNumber* permette di controllare se il numero d'ordine è corretto.

```

public void makeShipment(String orderNumber, List<Courier> courierList) throws NoSuchElementException,
    Invoice invoice = checkOrderNumber(orderNumber);
    String shippingAddress = invoice.getAddress();

    for (Map.Entry<AbstractProduct, Integer> set : invoice.getProductMap().entrySet()) {
        for(int j = 0; j < set.getValue(); j++) {
            ConcreteProduct cp = makeConcreteProduct(set.getKey());
            cp.setAddress(shippingAddress);
            if(invoice.getShipmentOption() == ShipmentOption.COURIER_SHIPMENT) {
                if(!chooseCourier(cp, courierList))
                    throw new RuntimeException("All Courier are busy");
            } else {
                warehouse.addproductsSold(cp);
            }
        }
    }
}

```

Figura 21: secondo snippet classe logistic

### 2.5.4 ConcreteProduct, ConcreteCompoundProduct, ConcreteElementaryService

*ConcreteProduct* è un'interfaccia che, a differenza della classe *AbstractProduct*, rappresenta il prodotto concreto che dovrà essere spedito all'utente. Abbiamo scelto l'interfaccia poiché tutti i metodi sarebbero stati astratti quindi non aveva

senso usare la classe astratta. *ConcreteCompoundProduct* ha un attributo con l'indirizzo di spedizione, mentre *ConcreteElementaryService* ha due importanti attributi il numero di serie, la posizione che occupa nel magazzino e il peso in chilogrammi.

### 2.5.5 Courier

Rappresenta un corriere, ogni corriere ha un peso massimo(in kg) che può trasportare e uno stato (AVAILABLE o BUSY); quando la capacità di trasporto massima è raggiunta lo stato diviene BUSY, quando il corriere consegna la merce diviene AVAILABLE. Il metodo principale è *addProducts* che permette di prendere in consegna la merce e, se necessario, aggiorna lo stato qualora avesse raggiunto il carico massimo.

```
public void addProducts(ConcreteProduct cp) {  
    if (status == CourierStatus.AVAILABLE) {  
        products.add(cp);  
        increaseWeight(cp.getWeight());  
    } else if (status == CourierStatus.BUSY)  
        throw new RuntimeException("Courier is busy");  
    }  
  
    private void increaseWeight(int w) {  
        weight += w;  
        updateStatus();  
    }  
  
    private void updateStatus() {  
        if (weight >= maxWeight)  
            status = CourierStatus.BUSY;  
        else status = CourierStatus.AVAILABLE;  
    }  
}
```

Figura 22: snippet classe courier

## 3 Testing

In questa sezione andiamo a descrivere i casi di test usati per garantire il corretto funzionamento di alcune parti dell'applicazione.

I test sono stati realizzati con JUnit e sono test di tipo funzionale e strutturale.

### 3.1 Test strutturali

#### 3.1.1 CartTest

```
@Test
public void calculatePriceTest() {
    assertEquals(totPriceExpected, cart.priceCalculator(), 0);
}

@Test
public void updateProductTest() {
    int oldNProducts = cart.getProductList().size();
    double oldTotPrice = cart.priceCalculator();
    double newProductPrice = 20;

    cm.createElementaryService("tavolo", 20, "NEW", 33);
    cm.addToCatalog("tavolo", 1);
    AbstractProduct ap = cm.getAbstractProduct("tavolo");
    cart.addProduct(ap, 1);
    assertEquals(oldNProducts + 1, (cart.getProductList()).size());
    assertEquals(oldTotPrice + newProductPrice, cart.priceCalculator(), 0);

    cart.removeProduct("tavolo", 1);
    assertEquals(oldNProducts, (cart.getProductList()).size());
}

@Test
public void clearTest() {
    cart.clear();
    assertEquals(0, cart.getProductList().size());
}
```

Figura 23: Snippet Cart Test

Analizziamo i test effettuati sul carrello, Nello snippet 24 possiamo vedere i tre metodi di test.

Il metodo più interessante è *updateProductTest()*, dove controlliamo che l'aggiunta e la rimozione di elementi dal carrello venga effettuata correttamente.

#### 3.1.2 CatalogTest

Qui andiamo ad analizzare i test che vengono effettuati sul catalogo, come nel carrello anche qui abbiamo una prima fase di *setUp* dove andiamo ad inizializzare tutti gli elementi necessari per il funzionamento del test.

I metodi *search*, *add* e *remove* vengono testati insieme, abbiamo testato le stesse funzioni sia per il *CatalogWorker* che per il *Cart*.

```

@Test
public void updateCatalogWorkerTest() {

    cw.addToCatalog("tavolo_cucina", 5);
    catalog.search("tavolo_cucina");

    assertThrows(NoSuchElementException.class, () -> {
        catalog.search("cucina");
    });
    cw.removeFromCatalog("sedia", 3);

    catalog.search("sedia");

    assertThrows(CatalogException.class, () -> {
        cw.removeFromCatalog("sedia", 100);
    });
}

@Test
public void updateCartTest() {
    cw.createElementaryService("tavolo", 20, "NEW", 33);
    cw.addToCatalog("tavolo", 10);

    u1.searchProduct("tavolo");
    u1.addToCart("tavolo", 10);
    assertThrows(NoSuchElementException.class, () -> {
        catalog.search("tavolo");
    });

    u1.removeFromCart("tavolo", 3);
    assertEquals(3, u1.searchNumberProduct("tavolo"));
}

```

Figura 24: Snippet Catalog Test

### 3.1.3 LogisticTest

```
public class LogisticTest {
    Logistic l;
    List<Courier> courierList;

    @Before
    public void setup() {
        l = new Logistic();
        courierList = new ArrayList<>();
        Courier c1 = new Courier("c1", 80);
        Courier c2 = new Courier("c2", 80);
        Courier c3 = new Courier("c3", 80);
        courierList.add(c1);
        courierList.add(c2);
        courierList.add(c3);
    }

    @Test
    public void chooseCouriertest(){
        ConcreteElementaryService cp1 = new ConcreteElementaryService("cp1", 240, "scaffale1", "NEW", 001);
        l.chooseCourier(cp1, courierList);
        for(int i = 0; i < courierList.size(); i++) {
            assertEquals(CourierStatus.BUSY, courierList.get(i).getStatus());
        }
        courierList.get(0).shipmentCompleted();
        assertEquals(CourierStatus.AVAILABLE, courierList.get(0).getStatus());
    }
}
```

Figura 25: Snippet Logistic Test

Un'altra classe sotto test è **Logistic**, abbiamo testato solo il metodo *chooseCourier*, gli altri metodi sono testati nel SystemTest; come mostrato nello snippet 25, verifichiamo che l'assegnazione della spedizione ai corrieri avvenga correttamente controllando che, una volta raggiunta la capacità massima, il loro stato divenga BUSY.

### 3.1.4 UserTest

Per quanto riguarda la classe User vengono testati due metodi: il primo che aggiunge un prodotto al carrello, il secondo che ricerca un prodotto all'interno del catalogo. Non si ritiene che l'implementazione sia degna di approfondimenti e può essere visionata direttamente sul codice.

### 3.1.5 SystemTest

Abbiamo infine effettuato un test E2E, nel quale abbiamo esaminato il corretto funzionamento dell'intero progetto. In questo test è stato effettuato un ordine da parte dell'utente e abbiamo verificato che un corriere lo prendesse in consegna correttamente.

## 4 Sequence Diagrams

Di seguito sono riportati 3 sequence diagram relativi a:

1. inserimento di un prodotto nel carrello da parte dell'utente.

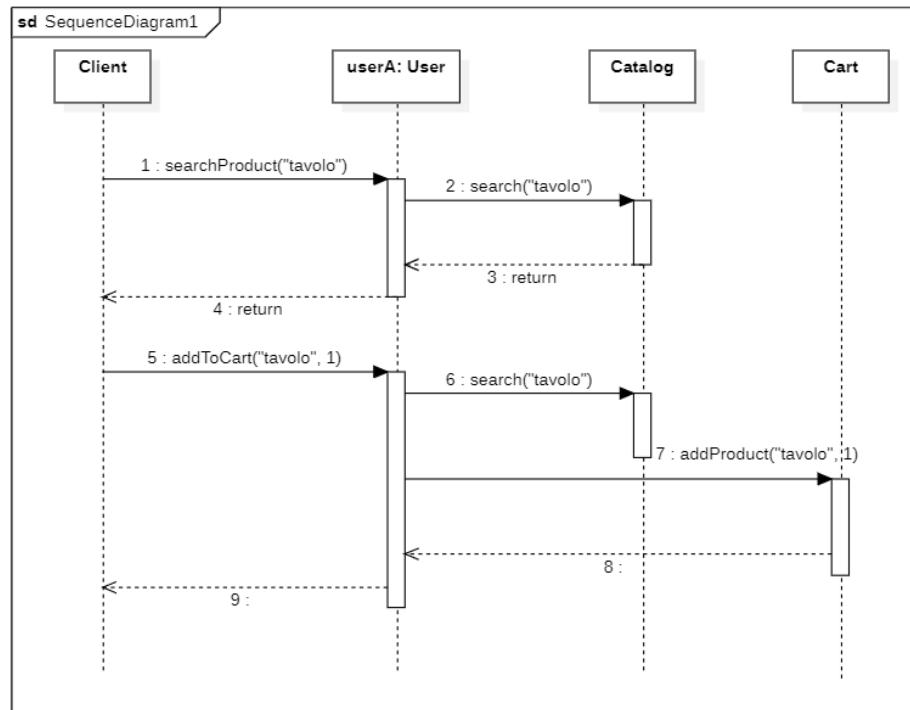


Figura 26: primo sequence Diagram

2. creazione ordine dopo che l'utente ha completato l'ordine.

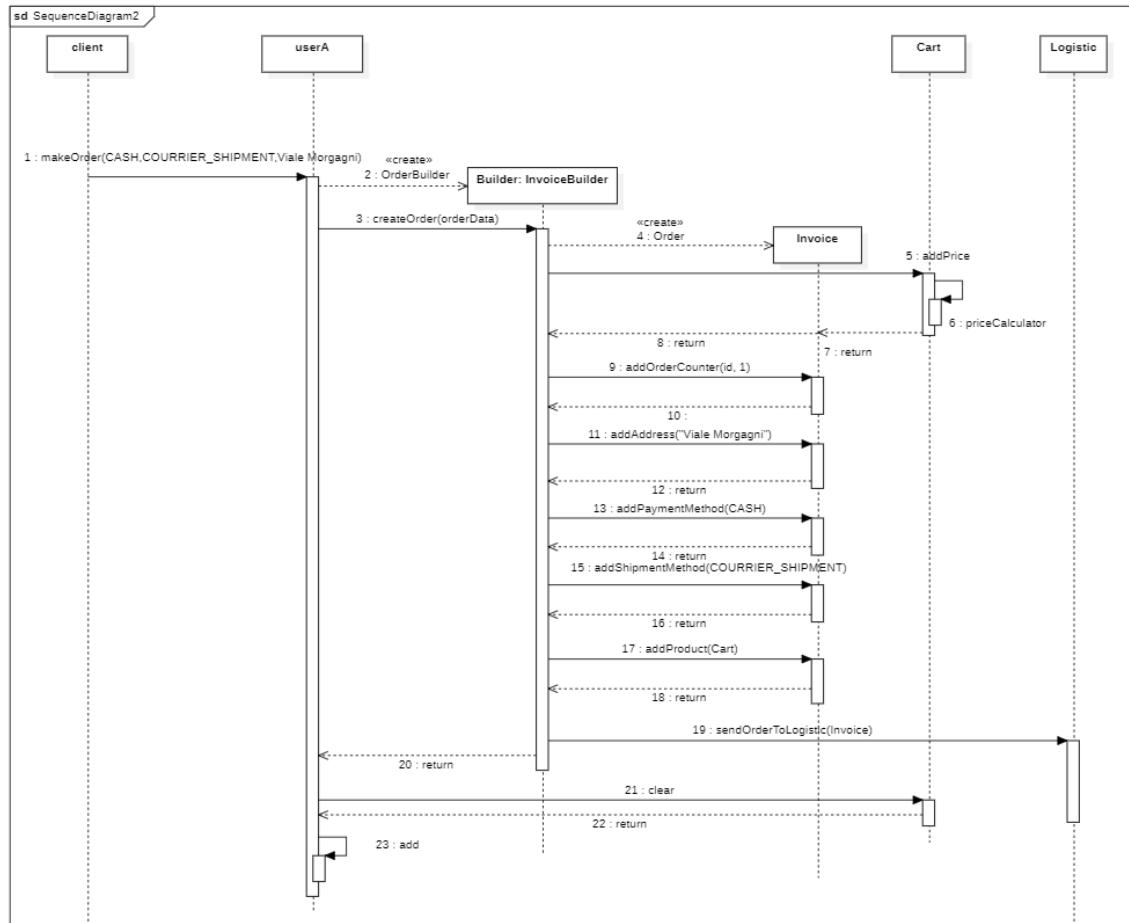


Figura 27: secondo sequence Diagram

3. creazione della spedizione da parte della classe logistic scelta dei corrieri.

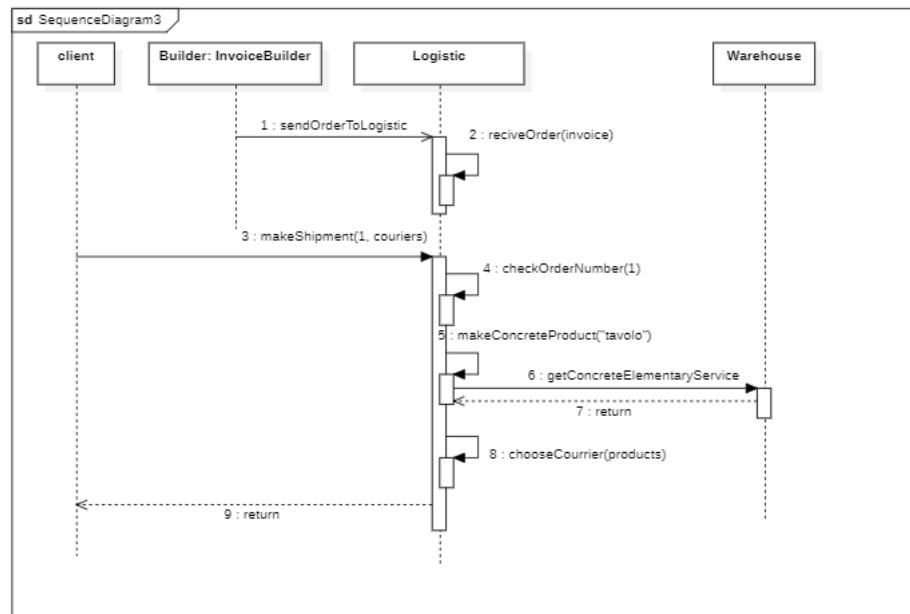


Figura 28: terzo sequence Diagram

## 5 Statement funzionale completo

Di seguito viene riportato lo statement funzionale completo per sviluppi futuri dell'applicazione.

Una delle possibili estensioni è sicuramente l'inserimento di un'interfaccia grafica.

Nel *catalogo* oltre a tutti i prodotti che possono essere visti e acquistati dall'utente si potrebbe:

- consentire all'utente di ordinare prodotti non disponibili al momento, ciò richiede che venga effettuato un ordine ai fornitori dell'azienda in questioni per richiedere la merce.
- permettere all'utente di modificare interattivamente le composizioni presenti nel catalogo, scegliendo ad esempio fra diverse colorazioni.

L'utente può completare l'acquisto creando un ordine in aggiunta a quanto disponibile potrebbe essere possibile scegliere un pagamento dilazionato, e scegliere un giorno preciso con la consegna.

Una volta che l'utente ha completato l'ordine *Logistic* riceve la relativa fattura, rimuove i prodotti acquistati dal magazzino e, successivamente, seleziona uno o più corrieri assegnandogli l'ordine che dovrà essere consegnato all'utente.

Ai corrieri assunti nell'azienda potrebbero essere affiancata l'opzione dei contoterzisti a cui far spedire la merce per fornire supporto. A questi ultimi il cliente potrebbe assegnare un voto(rating), fornendo così all'azienda un feedback sull'affidabilità di questi.