Progetto Java Store

Maltempo Luca, Ruotolo Samuele, Siliani Pietro

a.a. 2021-2022

Indice

1 Introduzione

L'elaborato consiste nella realizzazione di un applicativo che permetta l'acquisto di prodotti da un negozio. Il cliente, registratosi al sistema, potrà visionare il catalogo, scegliere uno o più prodotti e completare l'acquisto. L'utente, inoltre, ha la possibilità di visionare lo stato delle spedizioni e richiedere diverse operazioni, come la modifica dell'indirizzo di destinazione, la cancellazione di un ordine o la sua restituzione.

1.1 Specifica di Progetto

Il Cliente attraverso il Negozio si registra al sistema, effettua l'operazione di connessione inserendo i propri dati utente (E-Mail, Password), e terminate le interazioni col sistema si disconnette da esso.
Il Cliente visualizza il catalogo dal Negozio, seleziona gli articoli desiderati e procede all'acquisto. La procedura d'acquisto richiede all'utente il Servizio di Spedizione desiderato e l'inserimento dei seguenti dati: nome, indirizzo di consegna. Se l'utente ha spedizioni attive può visualizzarne lo stato in tempo reale, richiedere la modifica dell'indirizzo di consegna e la cancellazione dell'ordine relativo a tale consegna. Se sono presenti spedizioni consegnate l'utente può richiedere il reso dell'ordine associato a tale spedizione.
Il Negozio è costituito da un Reparto Utente, un Reparto Acquisti ed un Reparto Spedizioni. Tramite il Reparto Utente effettua le operazioni necessarie all'identificazione del Cliente. Il Reparto Acquisti permette di visualizzare il catalogo e di accedere alla procedura d'acquisto. Infine, il Reparto Spedizioni consente di monitorare lo stato delle spedizioni attive e di richiedere:

- La modifica dell'indirizzo di consegna.
- La cancellazione di un ordine.
- Il reso di un ordine.

Il Reparto Utente permette ai diversi utenti di registrarsi al sistema, effettuare una procedura di identificazione (Login) e infine di disconnettersi (Logout).
Il Reparto Acquisti espone un catalogo dal quale selezionare ed acquistare prodotti di diverso tipo, definisce ed implementa la procedura d'acquisto. Una volta eseguito l'acquisto il Reparto Acquisti comunica al Reparto Spedizioni del Negozio la necessità di gestire la Spedizione stessa trasferendogli i dati dell'acquirente, i prodotti acquistati ed il tipo di servizio selezionato dall'utente. Il Reparto Spedizioni riceve la richiesta di creazione di una spedizione, dopodiché procede alla creazione della Spedizione a cui associa un codice univoco generato automaticamente. Dopo aver creato una Spedizione procede ad assegnarla ad un Corriere. In base alla tipologia di servizio selezionata dall'utente, il Reparto Spedizioni istanzia un Servizio di Spedizione e gli affida la spedizione appena creata.
Il Servizio di Spedizione si occupa di verificare la validità delle richieste di modifica indirizzo, cancellazione e reso relative alla spedizone associata.

Il Servizio di Spedizione implementa inoltre la procedura di modifica dell'indirizzo di consegna e dello stato della spedizione. Sono disponibili 3 Servizi di Spedizione:

- Un Servizio Standard che prevede tempi di consegna mediamente più lenti e permette la modifica dell'indirizzo di destinazione fino a quando la spedizione non è ancora partita.
- Un Servizio Premium che prevede tempi di consegna rapidi e permette la modifica dell'indirizzo di destinazione fino a quando la spedizione non è in consegna.
- Un Servizio di Reso creato dal Dipartimento Spedizioni per la gestione di un reso. Sul Servizio di Reso non è possibile effettuare richieste di cancellazione, modifica indirizzo e reso.

La Spedizione è univocamente definita da un codice di spedizione ed è descritta da: nominativo del mittente, nominativo del destinatario, indirizzo del mittente, indirizzo del destinatario, contenuto e stato.
Il Corriere, presa in carico una Spedizione, controlla la priorità della stessa, si occupa del processo di consegna e di aggiornare lo stato della Spedizione tramite il Servizio di Spedizione.

2 Diagrammi

2.1 Use Case Diagrams

2.1.1 Customer

Figure 1: Use Case Diagram del Customer.

2.1.2 Store

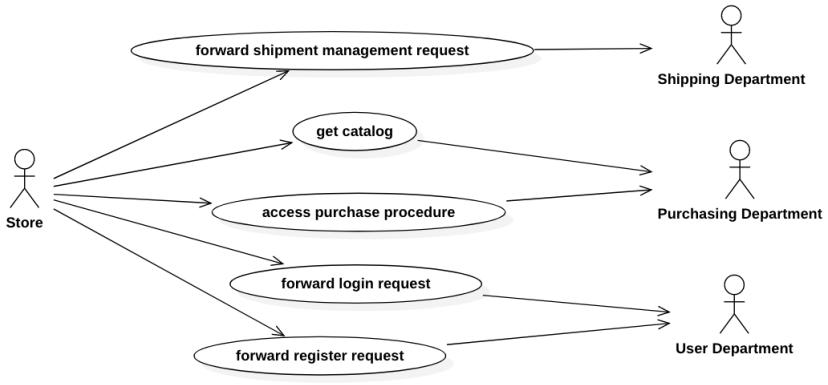


Figure 2: Use Case Diagram dello Store.

2.1.3 Purchasing Department

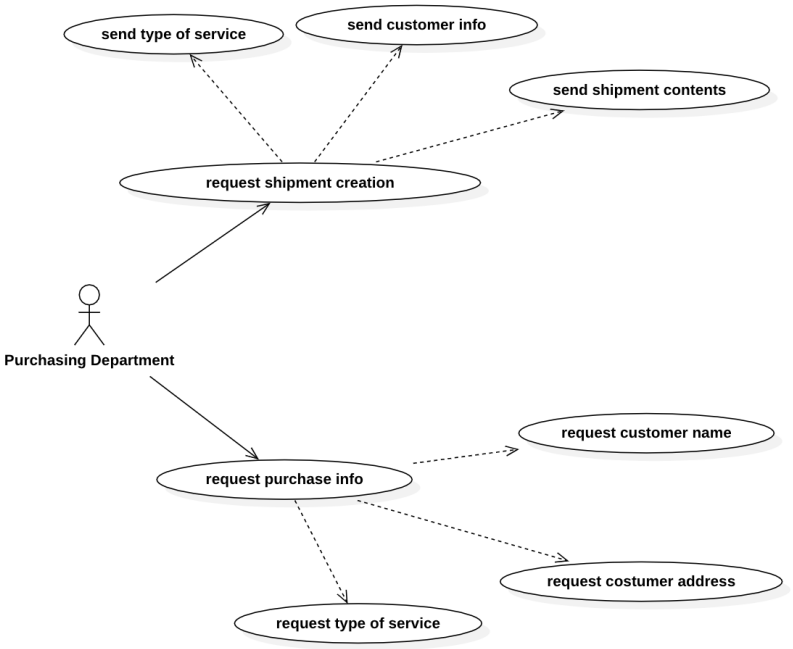


Figure 3: Use Case Diagram del Purchasing Department.

2.1.4 Shipping Department

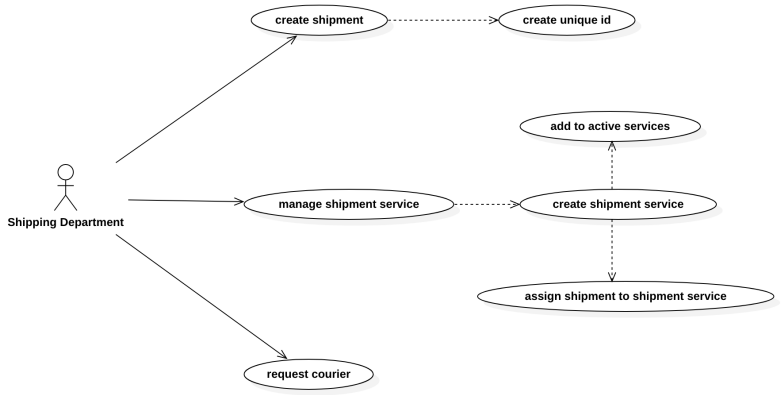


Figure 4: Use Case Diagram dello Shipping Department.

2.1.5 User Department

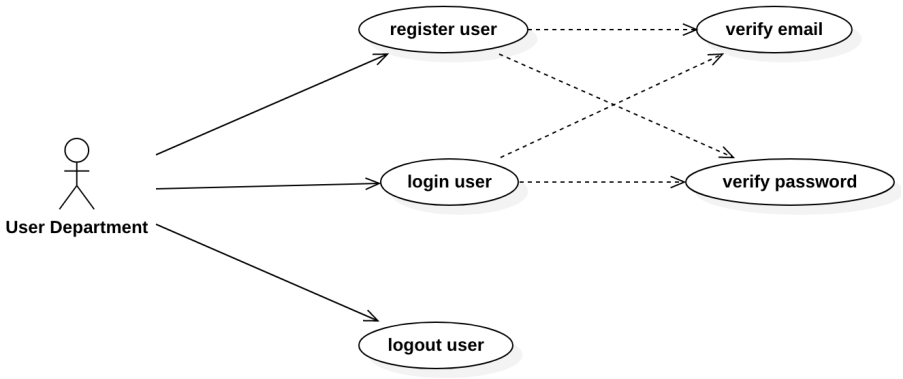


Figure 5: Use Case Diagram dello User Department.

2.1.6 Shipment Service

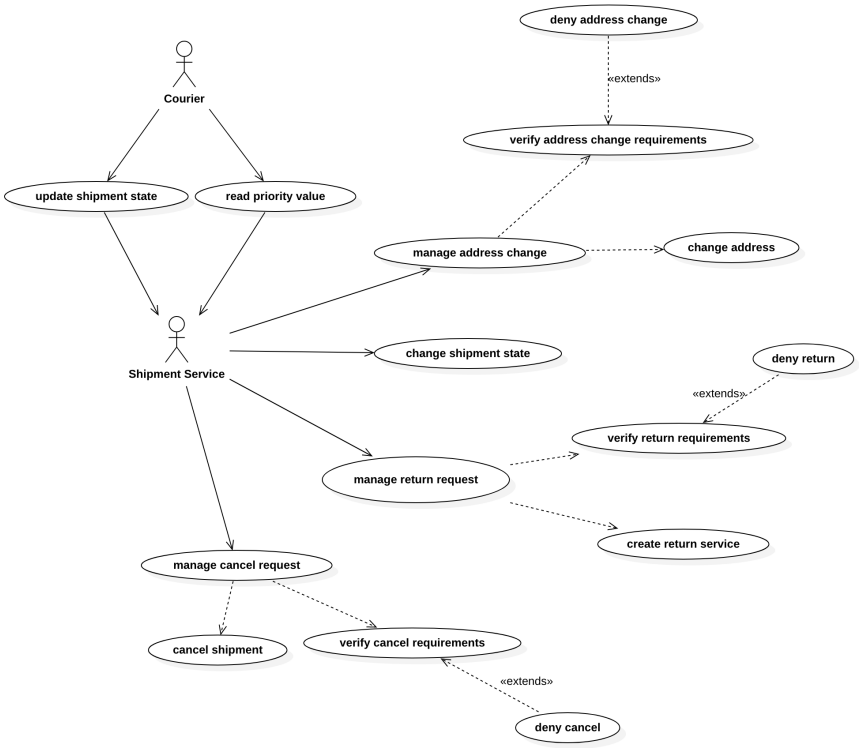


Figure 6: Use Case Diagram dello Shipment Service.

2.2 Use Case Templates

2.2.1 Acquisto

UC	**Buy Products #Purchase.1**
Level	User Goal
Description	Il cliente acquista almeno un prodotto tramite il negozio.
Primary Actors	Cliente
Secondary Actors	Negozio, Dipartimento Acquisti
Pre-conditions	• Il cliente ha già inserito nel carrello almeno un prodotto.
Post-conditions	• Viene creata una spedizione contenente i prodotti acquistati e il cliente può visualizzarla dalla home del negozio.
Basic Course	1. Il cliente seleziona “procedi al checkout” dall’interfaccia. 2. Il negozio richiede indirizzo di destinazione e nominativo del destinatario. 3. Il cliente inserisce i dati richiesti negli appositi campi. 4. Il negozio richiede il tipo di spedizione. 5. Il cliente seleziona il tipo di spedizione scegliendo tra “Standard” e “Premium”. 6. Il cliente conferma l’acquisto.
Alternative Flows	

Figure 7: Use Case Template dell’Acquisto.

2.2.2 Gestisci Spedizione

UC	**Manage Shipment #Shipment.1**
Level	User Goal
Description	L'utente visualizza lo storico spedizioni e seleziona una spedizione per visualizzarne i dati e/o richiedere un servizio ad essa associato.
Primary Actors	Cliente
Secondary Actors	Negozio
Pre-conditions	
Post-conditions	
Basic Course	1. Il cliente seleziona "I miei ordini" dalla home del negozio 2. Il sistema presenta al cliente lo storico delle spedizioni, per ogni spedizione riporta il numero di spedizione e lo stato. 3. Il cliente seleziona una spedizione specifica. 4. Il sistema mostra al cliente le informazioni complete della spedizione e opzioni di gestione quali "cambia indirizzo di consegna", "cancella ordine" o "richiedi reso".
Alternative Flows	5a. Il cliente seleziona "cambia indirizzo di consegna", si rimanda al use case: **#Shipment.4.** 5b. Il cliente seleziona "cancella ordine", si rimanda al use case: **#Shipment.3.** 5c. Il cliente seleziona "richiedi reso", si rimanda al use case: **#Shipment.2.**

Figure 8: Use Case Template della Gestione di una Spedizione.

2.2.3 Richiesta di Reso

UC	**Return Order #Shipment.2**
Level	User Goal
Description	Il cliente richiede il reso di una spedizione consegnata.
Primary Actors	Cliente
Secondary Actors	Negozio, Servizio di Spedizione
Pre-conditions	• Il cliente deve avere almeno una spedizione consegnata.
Post-conditions	• Lo storico spedizioni riporta “reso effettuato”.
Basic Course	1. Il cliente seleziona una spedizione consegnata e ne richiede il reso. 2. Il sistema comunica al cliente l’avvio della procedura di reso. 3. Il sistema aggiorna lo stato del reso in tempo reale. 4. Il sistema comunica al cliente la ricezione del reso.
Alternative Flows	
Issues	Il sistema richiede al cliente la motivazione del reso? Il sistema verifica la validità della richiesta di reso?

Figure 9: Use Case Template della Richiesta di Reso.

2.2.4 Cancellazione Ordine

UC	**Cancel Order #Shipment.3**
Level	User Goal
Description	Il cliente richiede la cancellazione di una spedizione in viaggio.
Primary Actors	Cliente
Secondary Actors	Negozio, Dipartimento Spedizioni, Servizio di Spedizione
Pre-conditions	• Il cliente deve avere almeno un ordine in viaggio.
Post-conditions	• Lo storico spedizioni riporta “ordine cancellato”.
Basic Course	1. Il cliente seleziona una spedizione in viaggio e ne richiede la cancellazione. 2. Il sistema comunica l’inoltro della richiesta al corriere. 3. Il sistema conferma al cliente l’approvazione della richiesta.
Alternative Flows	3a. Il sistema comunica al cliente il rifiuto della richiesta indicando il motivo.
Issues	Il sistema richiede al cliente la motivazione della cancellazione?

Figure 10: Use Case Template della Cancellazione Ordine.

2.2.5 Cambio Indirizzo

UC	**Change Address #Shipment.4**
Level	User Goal
Description	Il cliente richiede la modifica dell'indirizzo di consegna di una spedizione attiva.
Primary Actors	Cliente
Secondary Actors	Negozio, Servizio di Spedizione
Pre-conditions	• Il cliente deve avere almeno un ordine attivo.
Post-conditions	• Viene modificato l'indirizzo di consegna della spedizione. • Lo storico spedizioni riporta il nuovo indirizzo di consegna.
Basic Course	1. Il cliente seleziona una spedizione attiva e richiede di cambiarne l'indirizzo di consegna. 2. Il sistema chiede all'utente l'inserimento di un nuovo indirizzo di consegna. 3. Il cliente digita l'indirizzo di consegna desiderato nel campo apposito. 4. Il sistema comunica all'utente l'avvenuta modifica
Alternative Flows	4a. Il sistema rifiuta la richiesta indicando una motivazione.

Figure 11: Use Case Template del Cambio Indirizzo.

2.2.6 Registrazione Utente

UC	**Register #User.1**
Level	User Goal
Description	Il cliente si registra con una mail e una password.
Primary Actors	Cliente
Secondary Actors	Negozio, Dipartimento Utente
Pre-conditions	
Post-conditions	• Il sistema memorizza le credenziali dell'utente
Basic Course	1. Il cliente seleziona "registrati" dall'interfaccia. 2. Il cliente inserisce una email e una password. 3. Il sistema verifica la validità dell'email e della password. 4. Il sistema conferma al cliente l'avvenuta registrazione.
Alternative Flows	4a. Il sistema impedisce all'utente di registrarsi indicando una motivazione (e.g. email non valida o già inserita, password troppo breve, ecc...)

Figure 12: Use Case Template della Registrazione Utente.

2.2.7 Login Utente

UC	**Login #User.2**
Level	User Goal
Description	Il cliente esegue il login sulla piattaforma.
Primary Actors	Cliente
Secondary Actors	Negozio, Dipartimento Utente
Pre-conditions	• Il cliente deve essersi registrato.
Post-conditions	• Il cliente ottiene accesso al sistema.
Basic Course	1. Il cliente seleziona “login” dall’interfaccia. 2. Il cliente inserisce una email e una password. 3. Il sistema verifica la correttezza delle credenziali. 4. Il sistema comunica all’utente il successo dell’operazione.
Alternative Flows	4a. Il sistema nega l’accesso al cliente indicando la motivazione (e.g. mail assente dal sistema, password errata)

Figure 13: Use Case Template del Login Utente.

2.3 Class Diagrams

2.3.1 Store

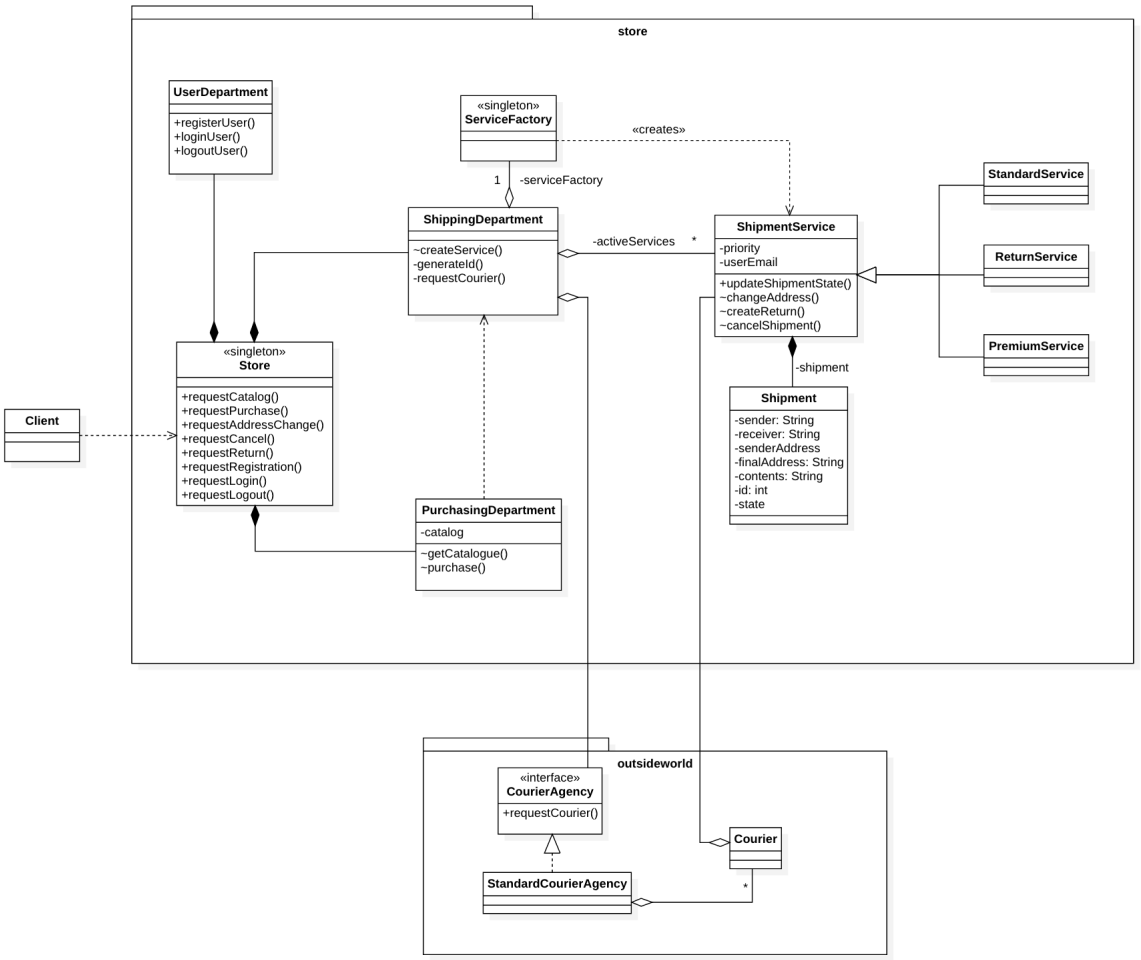


Figure 14: Class Diagram dello Store.

Lo Store è una **Facade**: aggrega i servizi offerti dai reparti sottostanti e permette al Client di accedervi. Lo Store è inoltre un singleton: questo perchè per tutto il funzionamento del programma non è necessario che dello Store sia attiva più di una sola istanza.
Lo Store è composto da tre reparti: PurchasingDepartment, ShippingDepartment e UserDepartment. La relazione tra Store e i tre reparti è descritta da una composizione: lo Store non avrebbe senso di esistere senza di essi, ciò in-

fatti causerebbe problemi nella gestione degli acquisti, delle spedizioni e nella registrazione e login degli utenti. Alcune delle richieste, quelle riguardanti acquisti, aggiunta al carrello di alcuni elementi e visualizzazione del catalogo, saranno gestite dallo Store inoltrandole al PurchasingDepartment, altre, come quelle riguardanti la gestione delle spedizioni, saranno invece inoltrate allo ShippingDepartment, infine, quelle riguardanti la gestione degli utenti, saranno inoltrate allo UserDepartment.
É prevista un'interazione tra PurchasingDepartment e ShippingDepartment, come esplicitato nel Class Diagram, ma non sono imposte limitazioni dal diagramma su come questa interazione debba avvenire.
Lo ShippingDepartment possiede una lista di ShipmentService attivi e tramite essa può procedere alla creazione delle Shipment e alla loro associazione ad uno ShipmentService. La creazione degli ShipmentService però non è fatta direttamente dallo ShippingDepartment che invece sfrutta, per tale funzione, una factory: la ShipmentFactory. Essa, conoscendo il tipo di servizio da istanziare, procede alla creazione di esso e ad associargli la particolare spedizione appena creata dallo ShippingDepartment. La scelta di utilizzare una factory semplifica tutte le operazioni di creazione: lo ShippingDepartment non avrà bisogno di avere metodi di creazione differenti a seconda del particolare tipo di ShipmentService.
É stato scelto, per lo ShipmentService, di usare una classe astratta, da cui derivano tutte le classi corrispondenti ad un'implementazione specifica di un Servizio di Spedizione, in modo da semplificare le operazioni con tale classe: in questo modo è infatti possibile astrarre completamente dai particolari implementativi delle classi che derivano dalla classe base e, soprattutto, ciò permette di specificare il particolare tipo di ShipmentService a runtime. É così possibile avere un codice generale, che valga per tutti i tipi di ShipmentService così da poter estendere il progetto aggiungendo nuovi tipi di ShipmentService senza intaccare il corretto funzionamento del programma.
La relazione che intercorre tra uno ShipmentService e una Shipment è quella della composizione. Lo ShipmentService è infatti un oggetto che viene utilizzato solo legato ad una Shipment per gestirne alcuni aspetti e di conseguenza non avrebbe senso di esistere se non fosse associato ad essa, d'altra parte se una Shipment non fosse associata ad uno ShipmentService non se ne potrebbero aggiornare i campi come lo stato o l'indirizzo di consegna, dunque una Shipment non ha senso di esistere se non è gestita da uno ShipmentService.
Ai fini di testare l'aggiornamento in tempo reale dello stato di Shipment, sono state sviluppate l'interfaccia CourierAgency e la classe Courier: la prima permette a ShippingDepartment di assegnare un'istanza di ShipmentService ad un'istanza di Courier, la seconda procede all'aggiornamento dello stato di Shipment tramite il metodo pubblico updateShipmentState() offerto da ShipmentService.

2.3.2 Strategy

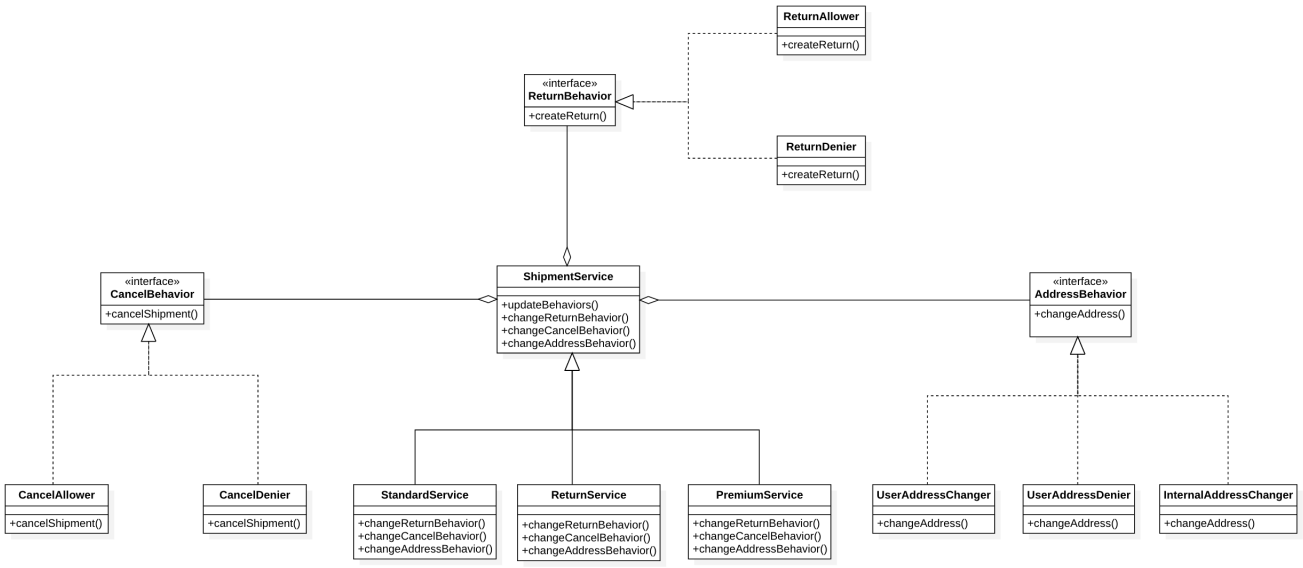


Figure 15: Class Diagram dello Strategy.

Le politiche di gestione delle modalità di cambio di indirizzo, del reso e della cancellazione dell'ordine, variano a seconda della particolare istanza della classe derivata di ShipmentService, inoltre una richiesta dell'utente relativa ad un ordine può essere accettata o respinta a seconda dello stato della spedizione che varia durante l'esecuzione dell'applicativo, per questo motivo è stato scelto di utilizzare il design pattern Strategy. Sono presenti una serie di interfacce, ognuna definisce un metodo per la gestione di una particolare richiesta (reso, cancellazione e modifica indirizzo), la classe ShipmentService ha un riferimento ad ognuna di queste interfacce e lo sfrutta per accedere alla specifica implementazione del metodo di gestione della richiesta. Questa soluzione permette quindi di poter modificare in modo dinamico il particolare comportamento adottato dai vari ShipmentService: ogni volta che cambia lo stato della Shipment si controlla se è opportuno modificare il comportamento degli ShipmentService ed eventualmente si procede all'aggiornamento.
Le interfacce utilizzate dallo Strategy sono:

- ReturnBehavior, utilizzata per implementare le politiche di reso:
  - ReturnAllower: permette di eseguire il reso di una spedizione;
  - ReturnDenier: non permette di eseguire il reso di una spedizione;
- CancelBehavior, utilizzata per implementare le politiche di cancellazione:

- CancelAllower: permette di eseguire la cancellazione di una spedizione;
- CancelDenier: non permette di eseguire la cancellazione di una spedizione;

- AddressBehavior, utilizzata per implementare le politiche di cambio indirizzo:
  - UserAddressChanger: permette all'utente di eseguire il cambio dell'indirizzo di spedizione;
  - UserAddressDenier: non permette all'utente di eseguire il cambio dell'indirizzo di spedizione;
  - InternalAddressChanger: è stata pensata la possibilità di modificare l'indirizzo di un reso da parte di un amministratore del sistema, mentre ciò non deve essere possibile per un cliente. Tuttavia nell'applicativo non sono stati assegnati ruoli di amministratore, di conseguenza InternalAddressChanger è analogo a UserAddressDenier anche se riporta un messaggio di errore differente.

2.3.3 MVC

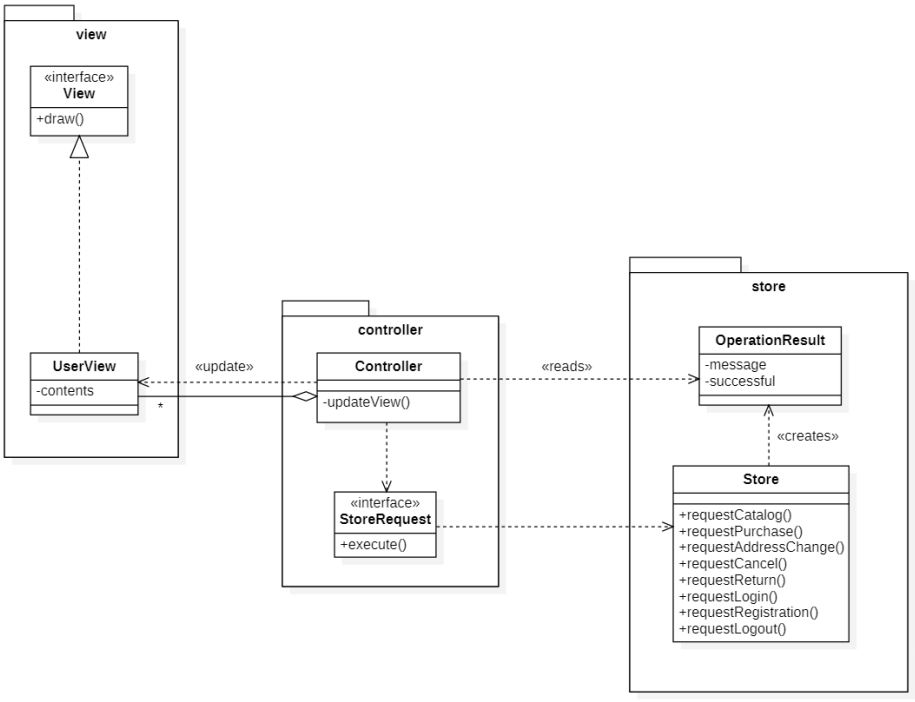


Figure 16: Class Diagram del pattern MVC

Per gestire l'aggiornamento in tempo reale delle spedizioni e per implementare un log che tenga traccia delle interazioni tra utente e sistema è stato utilizzato il pattern Model View Controller. Ogni UserView è associata ad un singolo utente e mostra:

- L'elenco delle spedizioni dell'utente, corredate di informazioni aggiuntive come indirizzo di destinazione e stato.
- Un log delle richieste rivolte al sistema e dei relativi risultati.

Il Model è costituito dai Reparti Utente, Acquisti e Spedizioni con i quali il Controller interagisce attraverso la facciata offerta da Store; l'interazione tra i due è mediata da due ulteriori oggetti: StoreRequest e OperationResult.
StoreRequest è un'interfaccia, ogni implementazione del metodo execute() farà riferimento ad un particolare metodo pubblico offerto da Store e fornirà gli input necessari alla chiamata. OperationResult costituisce una struct che racchiude l'esito dell'operazione (successo o insuccesso) ed un messaggio da comunicare all'utente. L'utilizzo di StoreRequest e OperationResult semplifica la comunicazione tra Model e Controller, il Controller non ha infatti bisogno di essere a conoscenza dei metodi pubblici esposti da Store, ignorando così anche gli input ad essi necessari. Inoltre, sebbene la comunicazione tra Store e Controller non si possa ridurre al solo utilizzo di OperationResult, questa classe standardizza la modalità con cui si comunica al Controller l'esito di una richiesta rivolta a Store.
Il Controller possiede inoltre una collezione di UserView, e si occupa di aggiornare ogni UserView tramite il metodo updateView().

2.3.4 Listener

A corredo del pattern MVC è stato utilizzato il pattern Listener con due implementazioni differenti, una permette di svillupare il sistema di comunicazione tra Client e Controller, l'altra fornisce un'ulteriore modalità di comunicazione tra il Model ed il Controller tramite la quale mantenere aggiornate le informazioni di una Spedizione mostrate a schermo con quelle effettivamente registrate nel Model.

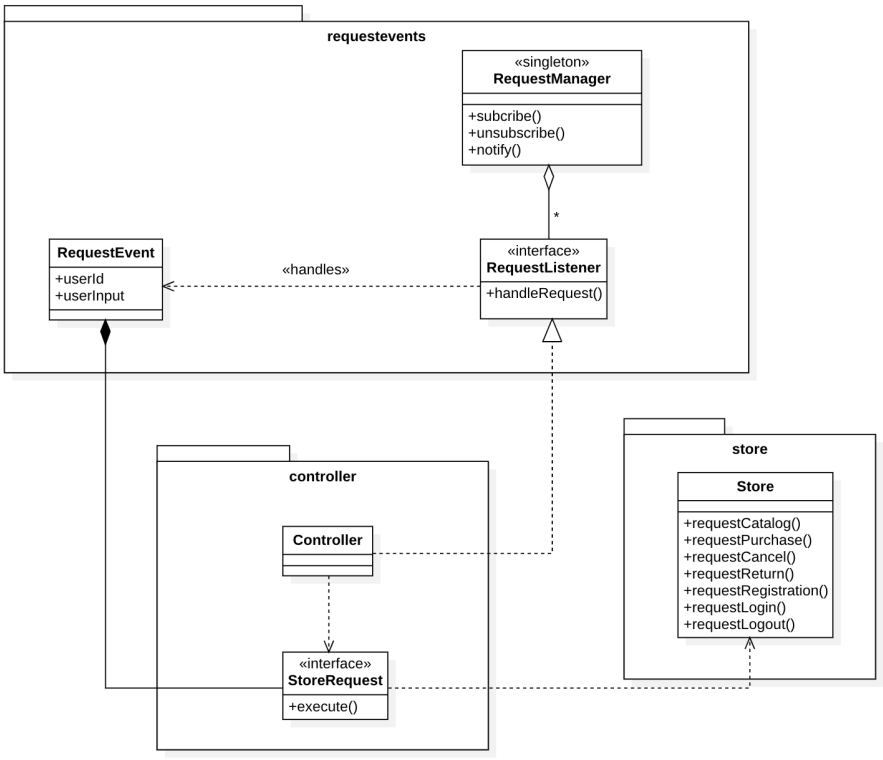


Figure 17: Class Diagram del pattern Listener per la gestione delle Richieste

Richieste Per poter accettare richieste dal Client il Controller implementa l'interfaccia RequestListener, ogni richiesta effettuata dall'utente è incapsulata in un evento di tipo RequestEvent il quale contiene anche un identificatore dell'utente che ha eseguito la richiesta e gli input che l'utente ha fornito.
Il singleton RequestManager si occupa dell'iscrizione, da parte di uno o più RequestListener, alle richieste di loro interesse, l'identificativo di una richiesta è associato a una particolare implementazione dell'interfaccia StoreRequest; nell'applicativo, al fine di associare ogni implementazione ad un particolare identificativo, è stato utilizzato il pattern Strategy in combinazione con una enumaration.
Ogni RequestEvent deve sempre contenere una StoreRequest, è compito della classe che crea un'istanza di RequestEvent scegliere l'implementazione di StoreRequest da incapsulare all'interno di essa, il metodo notify() di RequestManager permette di propagare un RequestEvent verso tutti i RequestListener che si sono iscritti all'implementazione di StoreRequest traspotata da esso.
Nel caso specifico dell'applicativo realizzato l'unico RequestListener è il Controller, l'utilizzo del pattern Listener per l'interazione con il Controller simula

la presenza di bottoni, ai quali è affidato il compito di creare eventi e che costituiscono l'interfaccia a disposizione dell'utente.

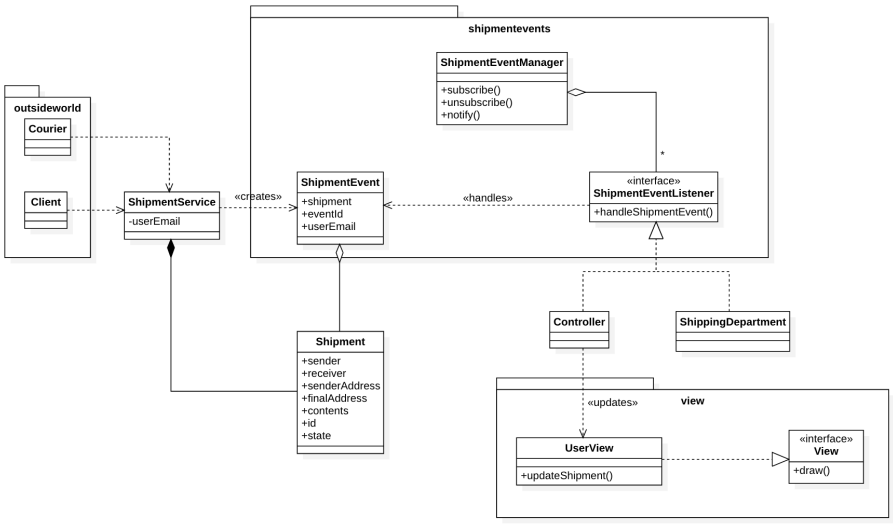


Figure 18: Class Diagram del pattern Listener per l'aggiornamento delle Spedizioni

Spedizioni Un'ulteriore forma di comunicazione tra Controller e Model è realizzata mediante l'impiego di ShipmentEvent, un'istanza di questa classe ha il ruolo di propagare informazioni riguardanti una spedizione a dei listener opportunamente registrati ad uno o più eventi, ogni qual volta un'istanza di ShipmentService modifica l'istanza di Shipment associata ne incapsula una copia (in particolare un riferimento ad una deep copy) all'interno di un'istanza di ShipmentEvent, sarà poi ShipmentEventManager, tramite l'interfaccia notify(), a permettere la propagazione dell'evento indirizzandolo verso i listener interessati; la notifica avviene quindi in modalità pull, in quanto ShipmentEvent trasporta un riferimento a Shipment, da utilizzare a discrezione dei listener per recuparare le informazioni di loro interesse come ad esempio stato della spedizione, indirizzo di consegna e contenuto.
Nell'applicativo l'interfaccia ShipmentEventListener è implementata sia da Controller che da ShippingDepartment, il Controller sfrutta la spedizione per aggiornare il contenuto di UserView, aggiungendo una voce per una nuova spedizione o modificandone una già esistente; ShippingDepartment invece si registra ad eventi di reso, e viene così notificato ogni qual volta sia necessario creare una nuova spedizione corrispondente ad un reso richiesto dall'utente.

2.4 Sequence Diagram

2.4.1 Richiesta di Reso

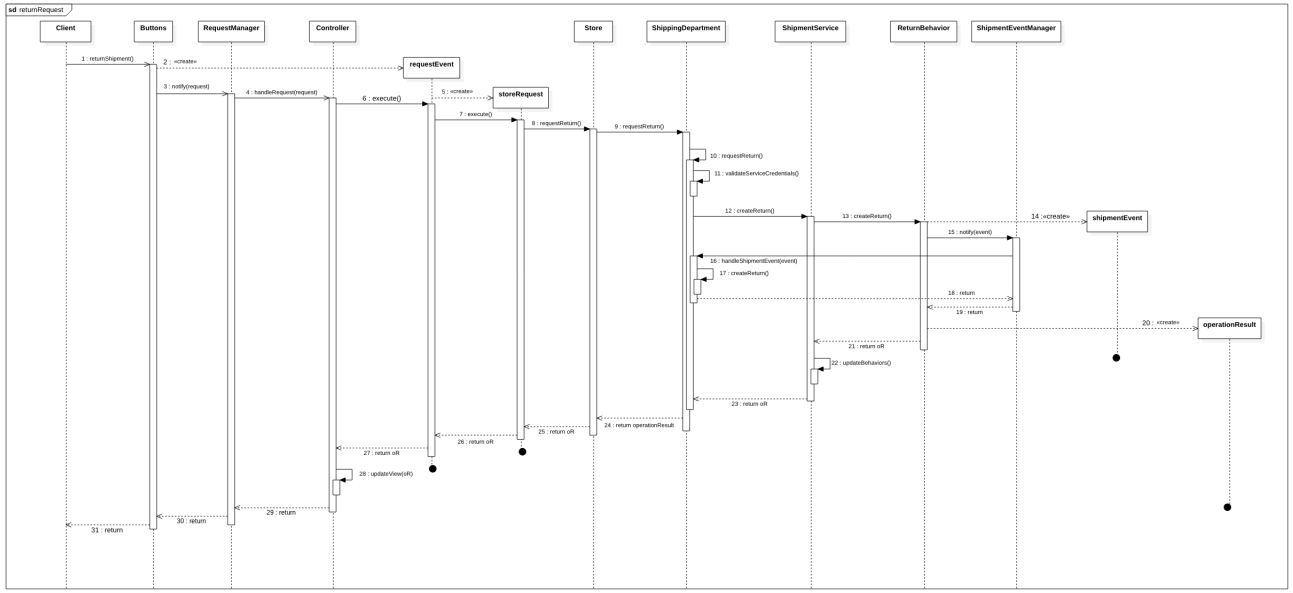


Figure 19: Sequence Diagram della richiesta di reso.

Nota: *I nomi di classi che cominciano per una lettera minuscola indicano istanze il cui ciclo di vita è limitato al periodo di gestione della richiesta, e che verranno quindi cancellate prima del termine dell'esecuzione del programma.*

Il Client attraverso Button richiede un reso, questo causa la generazione di un evento di tipo richiesta di cui il Controller è un Listener. La creazione di un requestEvent provoca a sua volta la creazione di uno storeRequest, quest'ultimo è una enumerazione di possibili tipi di richiesta, ogni tipo realizza l'implementazione concreta del metodo execute(). A questo punto il Controller prende in carico la gestione della richiesta ed esegue il metodo execute() di requestEvent che a sua volta inoltra la chiamata al vero execute() contenuto nello storeRequest appena creato. Il metodo execute() dello storeRequest provvede a chiamare, tramite la facade Store, il metodo per effettuare il reso dello ShippingDepartment. Quest'ultimo seleziona l'istanza di ShipmentService preposta a gestire la spedizione richiesta dal Client. Lo ShipmentService utilizza il proprio riferimento all'interfaccia ReturnBehavior per chiamare l'implementazione di createReturn() corretta in base allo stato della spedizione e, assumendo che l'implementazione utilizzata sia quella di ReturnAllower, tale metodo genera un evento per notificare a ShippingDepartment la necessità di creare una nuova spedizione di reso, una volta fatto questo ritorna il risultato tramite un oggetto di tipo operationResult.
Le richieste di cambio indirizzo di spedizione o cancellazione di un ordine seguono lo stesso flusso di esecuzione.
Creato un evento di tipo richiesta tramite lo specifico bottone dell'interfaccia, il Controller, che è in ascolto delle richieste, gestisce questo tipo di eventi eseguendo il metodo execute() che a sua volta è implementato diversamente in base al tipo di richiesta tramite la enum storeRequest. Infine, sarà lo ShippingDepartment ad inviare la richiesta ad un'istanza specifica di ShipmentService che si occuperà della sua gestione e di creare un operationResult. Il Controller, ricevuto indietro l'operationResult può procedere all'aggiornamento del Log del Client aggiungendo l'esito della richiesta.

3 Unit Testing

Al fine di verificare il corretto comportamento dell'applicativo sono stati sviluppati degli Unit Test tramite l'impiego del framework JUnit. I test riguardano sia le singole classi che gli Use Case individuati nella sezione Use Case Templates. A scopo riassuntivo di seguito è riportato solo il codice corrispondente ai test riguardanti gli Use Case, si noti che tali test mirano a verificare la corretta interazione tra le classi dell'applicativo, test più approfonditi sul comportamento delle singole classi (ad esempio verifica della validità di una richiesta per ogni possibile stato di una spedizione) sono stati sviluppati separatamente in altre classi di Test.

3.1 Classi Helper

Di seguito sono riportate le Classi Helper sviluppate a corredo delle classi di Test per semplificarne l'implementazione:

3.1.1 UseCaseUtility

Lo scopo della classe UseCaseUtility è quello di fornire metodi pubblici che semplifichino l'inizializzazione ed il cleanup eseguiti prima e dopo ogni test.

```
1  public final class UseCaseUtility {
2
3      private UseCaseUtility() {}
4
5      public static void init(UserDepartment userDepartment, ShippingDepartment
       ↪  shippingDepartment,
6                              PurchasingDepartment purchasingDepartment) throws
                               ↪  StoreInitializationException {
7          //set store departments
8          //then create the singleton instance of Store and register new users
9          initStore(userDepartment, shippingDepartment, purchasingDepartment);
10         registerUsers();
11     }
12
13     private static void initStore(UserDepartment userDepartment,
       ↪  ShippingDepartment shippingDepartment, PurchasingDepartment
       ↪  purchasingDepartment) {
14         Store.setUserDepartment(userDepartment);
15         Store.setShippingDepartment(shippingDepartment);
16         Store.setPurchasingDepartment(purchasingDepartment);
17     }
18
19     private static void registerUsers() throws StoreInitializationException {
20         //register and login new users, then initialize their carts and service
           ↪  lists
21         Store.getInstance().requestRegistration(UseCaseConstants.USER_EMAIL,
           ↪  UseCaseConstants.USER_PASSWORD);
22         Store.getInstance().requestLogin(UseCaseConstants.USER_EMAIL,
           ↪  UseCaseConstants.USER_PASSWORD);
23         Store.getInstance().addUserCart(UseCaseConstants.USER_EMAIL);
24         Store.getInstance().addUserServices(UseCaseConstants.USER_EMAIL);
25         Store.getInstance().requestRegistration(
           ↪  UseCaseConstants.ANOTHER_USER_EMAIL,
           ↪  UseCaseConstants.ANOTHER_USER_PASSWORD);
26         Store.getInstance().requestLogin(UseCaseConstants.ANOTHER_USER_EMAIL,
           ↪  UseCaseConstants.ANOTHER_USER_PASSWORD);
27         Store.getInstance().addUserCart(UseCaseConstants.ANOTHER_USER_EMAIL);
28         Store.getInstance().addUserServices(
           ↪  UseCaseConstants.ANOTHER_USER_EMAIL);
29     }
```

Il metodo init è utilizzato per inizializzare il negozio e gli utenti prima di ogni test tramite l'annotazione @Before, si appoggia sui metodi privati initStore(), che assegna le istanze dei dipartimenti a Store, e registerUsers(), che registra due

utenti ed inizializza il carrello e la lista di Servizi di Spedizione corrispondenti. Da notare che:

- La creazione delle istanze dei dipartimenti è delegata alle singole classi di test.
- L'istanza di Store che sarà utilizzata all'interno del test è creata in seguito alla prima chiamata del metodo getInstance() all'interno di registerUsers().

```
1    public static void cleanup() {
2        //logout registered users and delete Singleton instances
3        Buttons.getInstance().logoutUser(UseCaseConstants.ANOTHER_USER_EMAIL);
4        Buttons.getInstance().logoutUser(UseCaseConstants.USER_EMAIL);
5        Store.clearInstance();
6        RequestManager.clearInstance();
7        ShipmentEventManager.clearInstance();
8        ViewEventManager.clearInstance();
9    }
```

Il metodo cleanup, dopo aver eseguito il logout dei due utenti registrati, cancella le istanze delle classi Singleton utilizzate, è chiamato dopo ogni test sfruttando l'annotazione @After.

3.1.2 UseCaseConstants

Tale classe costituisce un insieme di costanti riutilizzate nei diversi test:

```
public final class UseCaseConstants {

    private UseCaseConstants () {}

    public static final String UNREG_USER_EMAIL = "unregistered@user.com";
    public static final String USER_NAME = "TEST USER";
    public static final String USER_EMAIL = "user@email.com";
    public static final String USER_PASSWORD = "userPassword1";
    public static final String ANOTHER_USER_EMAIL = "anotheruser@email.com";
    public static final String ANOTHER_USER_PASSWORD = "anotherUserPassword2";
    public static final String DESTINATION_ADDRESS = "destination address";
    public static final String LAMP_ID = "01998";
    public static final String SHIPMENT_ID = "#000001";
}
```

3.1.3 Package testagencies

Le classi del package testagencies sono utilizzate sia per i test relativi alle singole classi che per quelli associati ai Casi d'Uso, il loro scopo è quello di semplificare l'aggiornamento dello stato di una spedizione; in particolare:

- **InstantDeliveryAgency** implementa il metodo requestCourier() dell'interfaccia CourierAgency in modo tale da portare lo stato della spedizione assegnata a "Consegnato" o, nel caso di un reso, a "Reso consegnato":

```
public final class InstantDeliveryAgency implements CourierAgency {

    @Override
    public void requestCourier(ShipmentService shipmentService) {
        while (shipmentService.getShipment().getState().getNextState() !=
        ↪  null) {
            shipmentService.updateShipmentState();
        }
    }
}
```

- **ManualCourierAgency** implementa il metodo requestCourier() dell'interfaccia CourierAgency in modo da assegnare la spedizione ad un **ManualCourier**. **ManualCourier** offre un metodo pubblico updateShipment() che aggiorna lo stato della spedizione e può essere utilizzato per portare la spedizone nello stato desiderato:

```
public final class ManualCourierAgency implements CourierAgency {

```

```
    public ManualCourierAgency(ManualCourier courier) {
        this.courier = courier;
    }

    @Override
    public void requestCourier(ShipmentService shipmentService) {
        courier.setService(shipmentService);
    }

    ManualCourier courier;
}

public final class ManualCourier {

    public void updateShipment() {
        service.updateShipmentState();
    }

    public void setService(ShipmentService service) {
        this.service = service;
    }

    private ShipmentService service;
}
```

3.2 Acquisto

I test eseguiti per il Caso d'Uso relativo ad un acquisto sono i seguenti:

- Procedura d'acquisto eseguita correttamente sia nel caso di un servizio Standard che di un servizio Premium:

```
    @Test
    public void successfulPurchaseTest() throws
    ↪  StoreInitializationException {
        int standardQuantity = 1;
        int premiumQuantity = 3;
        successfulPurchase(standardQuantity, Constants.STANDARD);
        successfulPurchase(premiumQuantity, Constants.PREMIUM);
    }

    public static void successfulPurchase(int quantity, String
    ↪  typeOfService) throws StoreInitializationException {
        OperationResult addProductResult =
        Store.getInstance().addToCartRequest(UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.LAMP_ID, quantity);
        OperationResult purchaseResult =
        ↪  Store.getInstance().requestPurchase(typeOfService,
        ↪  UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.DESTINATION_ADDRESS,
        ↪  UseCaseConstants.USER_NAME);
        Assert.assertTrue(addProductResult.isSuccessful());
        Assert.assertTrue(purchaseResult.isSuccessful());
    }
```

- Acquisto da parte di un utente non registrato:

```
1       @Test
2       public void unregisteredUserTest() throws
        ↪  StoreInitializationException {
3           int quantity = 1;
4           String typeOfService = Constants.STANDARD;
5           OperationResult addProductResult =
            ↪  Store.getInstance().addToCartRequest(
            ↪  UseCaseConstants.UNREG_USER_EMAIL, UseCaseConstants.LAMP_ID,
            ↪  quantity);
6           OperationResult purchaseResult =
            ↪  Store.getInstance().requestPurchase(typeOfService,
7           UseCaseConstants.UNREG_USER_EMAIL,
            ↪  UseCaseConstants.DESTINATION_ADDRESS,
            ↪  UseCaseConstants.USER_NAME);
8
9           Assert.assertFalse(addProductResult.isSuccessful());
10          Assert.assertFalse(purchaseResult.isSuccessful());
11      }
```

- Acquisto da parte di un utente che non ha eseguito il login:

```
1       @Test
2       public void unloggedUserTest() throws StoreInitializationException {
3           int quantity = 2;
4           String typeOfService= Constants.STANDARD;
5           Store.getInstance().requestLogout(UseCaseConstants.USER_EMAIL);
6           OperationResult addProductResult =
            ↪  Store.getInstance().addToCartRequest(
            ↪  UseCaseConstants.USER_EMAIL, UseCaseConstants.LAMP_ID,
            ↪  quantity);
7           OperationResult purchaseResult =
            ↪  Store.getInstance().requestPurchase(typeOfService,
            ↪  UseCaseConstants.USER_EMAIL,
            ↪  UseCaseConstants.DESTINATION_ADDRESS,
            ↪  UseCaseConstants.USER_NAME);
8
9           Assert.assertFalse(addProductResult.isSuccessful());
10          Assert.assertFalse(purchaseResult.isSuccessful());
11      }
```

3.3 Cancellazione Ordine

Per lo Use Case relativo alla cancellazione di un ordine sono stati sviluppati i seguenti test:

- Cancellazione eseguita correttamente:

```java
    @Test
    public void successfulCancel() throws StoreInitializationException {
        int quantity = 1;
        String shipmentId = "#000001";
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result =
        ↪  Store.getInstance().requestCancel(UseCaseConstants.USER_EMAIL,
        ↪  shipmentId);
        Assert.assertTrue(result.isSuccessful());
    }
```

- Tentativo di cancellazione quando i requisiti non sono soddisfatti:

```java
    @Test
    public void cancelFailTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(instantAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result =
        ↪  Store.getInstance().requestCancel(UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di cancellazione di una spedizione da parte di un utente diverso dal proprietario della spedizione:

```java
    @Test
    public void cancelOtherTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result = Store.getInstance().requestCancel(
        ↪  UseCaseConstants.ANOTHER_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di cancellazione di una spedizione non esistente:

```java
    @Test
    public void missingShipmentTest() throws StoreInitializationException
    ↪  {
```

```
        OperationResult result =
        ↪  Store.getInstance().requestCancel(UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);
        Assert.assertFalse(result.isSuccessful());
    }
```

- Cancellazione di una spedizione da parte di un utente non registrato:

```
    @Test
    public void missingUserTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result = Store.getInstance().requestCancel(
        ↪  UseCaseConstants.UNREG_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Cancellazione di una spedizione da parte dell'utente proprietario della spedizione ma che non ha eseguito il login al sistema:

```
     @Test
    public void unloggedUserTest() throws StoreInitializationException {
        int quantity = 1;
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        Store.getInstance().requestLogout(UseCaseConstants.USER_EMAIL);
        OperationResult result =
        ↪  Store.getInstance().requestCancel(UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);

        Assert.assertFalse(result.isSuccessful());
    }
```

3.4 Cambio Indirizzo

Per il Caso D'Uso associato alla modifica di un indirizzo di consegna sono stati scritti i test seguenti:

- Modifica dell'indirizzo eseguita correttamente:

```
    @Test
    public void changeAddressSuccessTest() throws
    ↪  StoreInitializationException {
```

```
        shippingDepartment.setCourierAgency(manualAgency);
        BuyProductsTest.successfulPurchase(1, Constants.STANDARD);
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.USER_EMAIL, UseCaseConstants.SHIPMENT_ID,
        ↪  newAddress);

        Assert.assertTrue(result.isSuccessful());
    }
```

- Tentativo di modifica dell'indirizzo di consegna quando i requisiti per la modifica non sono soddisfatti:

```
    @Test
    public void addressChangeFailTest() throws
    ↪  StoreInitializationException {
        shippingDepartment.setCourierAgency(instantAgency);
        BuyProductsTest.successfulPurchase(1, Constants.STANDARD);
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.USER_EMAIL, UseCaseConstants.SHIPMENT_ID,
        ↪  newAddress);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di modifica dell'indirizzo di consegna da parte di un utente che non è il proprietario della spedizione:

```
    @Test
    public void changeOtherTest() throws StoreInitializationException {
        shippingDepartment.setCourierAgency(manualAgency);
        BuyProductsTest.successfulPurchase(1, Constants.STANDARD);
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.ANOTHER_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID, newAddress);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di modifica dell'indirizzo di consegna di una spedizone non esistente:

```
    @Test
    public void missingShipmentTest() throws StoreInitializationException
    ↪  {
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.USER_EMAIL, UseCaseConstants.SHIPMENT_ID,
        ↪  newAddress);
```

```
        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di modifica dell'indirizzo di consegna da parte di un utente non registrato:

```
    @Test
    public void missingUserTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.UNREG_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID, newAddress);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di modifica dell'indirizzo di consegna da parte dell'utente proprietario della spedizione ma che non ha eseguito il login al sistema:

```
    @Test
    public void unloggedUserTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(manualAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        Store.getInstance().requestLogout(UseCaseConstants.USER_EMAIL);
        OperationResult result =
        ↪  Store.getInstance().requestAddressChange(
        ↪  UseCaseConstants.USER_EMAIL, UseCaseConstants.SHIPMENT_ID,
        ↪  newAddress);
    }
```

3.5 Richiesta di Reso

Per lo Use Case relativo al reso di un ordine sono stati sviluppati i seguenti test:

- Richiesta di reso eseguita correttamente:

```
    @Test
    public void returnSuccessTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(instantAgency);
        BuyProductsTest.successfulPurchase(quantity, Constants.PREMIUM);
        OperationResult result =
        ↪  Store.getInstance().requestReturn(UseCaseConstants.USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);
```

```
7
8           Assert.assertTrue(result.isSuccessful());
9       }
```

- Tentativo di reso quando i requisiti per il reso non sono soddisfatti:

```
1       @Test
2       public void returnFailTest() throws StoreInitializationException {
3           int quantity = 1;
4           shippingDepartment.setCourierAgency(manualAgency);
5           BuyProductsTest.successfulPurchase(quantity, Constants.PREMIUM);
6           OperationResult result =
            ↪  Store.getInstance().requestReturn(UseCaseConstants.USER_EMAIL,
            ↪  UseCaseConstants.SHIPMENT_ID);
7
8           Assert.assertFalse(result.isSuccessful());
9       }
```

- Tentativo di reso da parte di un utente che non è il proprietario di una spedizione:

```
1       @Test
2       public void returnOtherTest() throws StoreInitializationException {
3           int quantity = 1;
4           shippingDepartment.setCourierAgency(instantAgency);
5           BuyProductsTest.successfulPurchase(quantity, Constants.STANDARD);
6           OperationResult result = Store.getInstance().requestReturn(
            ↪  UseCaseConstants.ANOTHER_USER_EMAIL,
            ↪  UseCaseConstants.SHIPMENT_ID);
7
8           Assert.assertFalse(result.isSuccessful());
9       }
```

- Tentativo di reso di una spedizione non esistente:

```
1       @Test
2       public void missingShipmentTest() throws StoreInitializationException
        ↪  {
3           OperationResult result =
            ↪  Store.getInstance().requestReturn(UseCaseConstants.USER_EMAIL,
            ↪  UseCaseConstants.SHIPMENT_ID);
4           Assert.assertFalse(result.isSuccessful());
5       }
```

- Tentativo di reso da parte di un utente che non è registrato:

```
    @Test
    public void missingUserTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(instantAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        OperationResult result = Store.getInstance().requestReturn(
        ↪  UseCaseConstants.UNREG_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);

        Assert.assertFalse(result.isSuccessful());
    }
```

- Tentativo di reso da parte dell'utente proprietario della spedizione ma che non ha eseguito il login al sistema:

```
    @Test
    public void unloggedUserTest() throws StoreInitializationException {
        int quantity = 2;
        shippingDepartment.setCourierAgency(instantAgency);
        String typeOfService = Constants.STANDARD;
        BuyProductsTest.successfulPurchase(quantity, typeOfService);
        Store.getInstance().requestLogout(UseCaseConstants.USER_EMAIL);
        OperationResult result = Store.getInstance().requestReturn(
        ↪  UseCaseConstants.UNREG_USER_EMAIL,
        ↪  UseCaseConstants.SHIPMENT_ID);
    }
```

3.6 Registrazione e Login

I test sviluppati per gli Use Case di registrazione e login sono piuttosto semplici: vengono testate una condizione di successo e una di fallimento per entrambe le operazioni, test più approfonditi su queste operazioni (come tentativo di registrarsi con una mail già associata ad un altro utente o tentativo di accesso con una password errata) sono stati scritti in classi di Test separate, di seguito sono riportati i test per i suddetti Use Case:

```
@Test
    public void registrationSuccessTest() throws StoreInitializationException {
        OperationResult result =
        ↪  Store.getInstance().requestRegistration(testUserEmail,
        ↪  testUserPassword);
        Assert.assertTrue(result.isSuccessful());
    }

    @Test
    public void registrationFailTest() throws StoreInitializationException {
        OperationResult result =
        ↪  Store.getInstance().requestRegistration(invalidUserEmail,
        ↪  testUserPassword);
```

```
10          Assert.assertFalse(result.isSuccessful());
11      }
12
13      @Test
14      public void loginSuccessTest() throws StoreInitializationException {
15          registrationSuccessTest();
16          OperationResult result =
            ↪  Store.getInstance().requestLogin(testUserEmail, testUserPassword);
17          Assert.assertTrue(result.isSuccessful());
18      }
19
20      @Test
21      public void loginFailTest() throws StoreInitializationException {
22          registrationSuccessTest();
23          OperationResult result =
            ↪  Store.getInstance().requestLogin(invalidUserEmail,
            ↪  testUserPassword);
24          Assert.assertFalse(result.isSuccessful());
25      }
```

4 Implementazione

4.1 Struttura

Di seguito è presentata l'organizzazione dei packages del codice sorgente dell'applicativo.

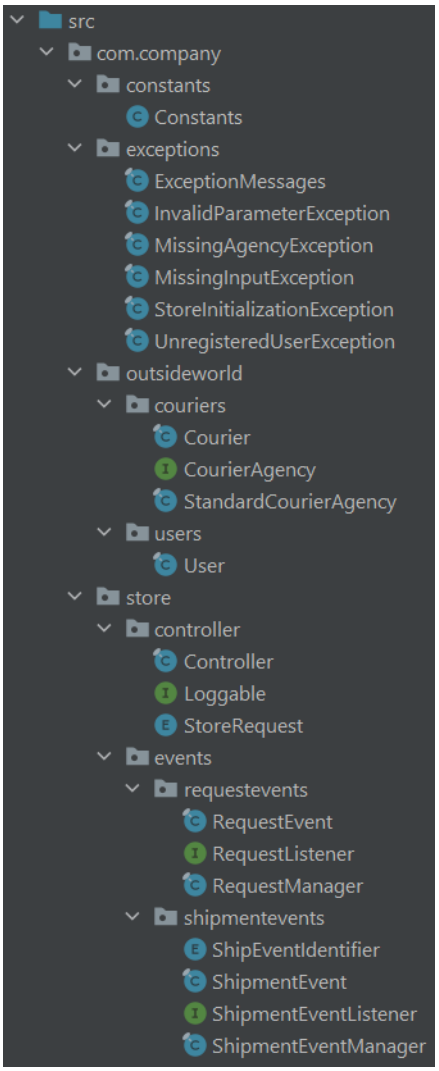


Figure 20: Organizzazione packages codice sorgente.

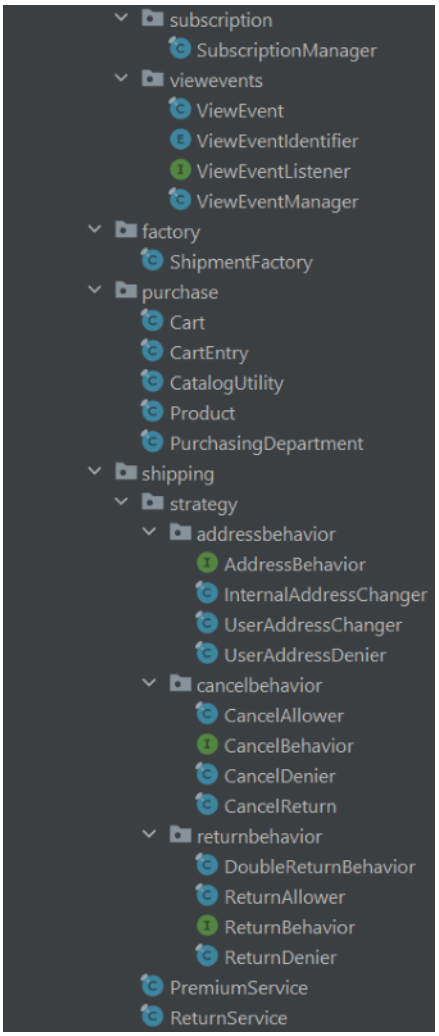


Figure 21: Organizzazione packages codice sorgente.

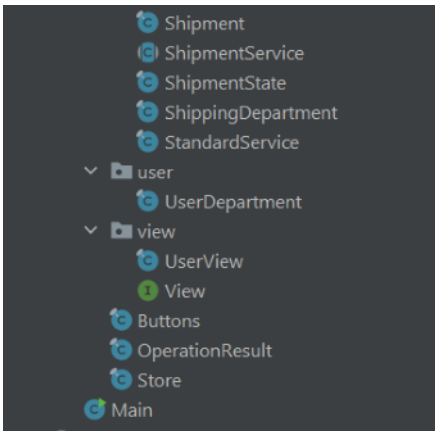


Figure 22: Organizzazione packages codice sorgente.

4.1.1 constants

Questo package contiene tutte le costanti che vengono utilizzate nel programma.

4.1.2 exceptions

Questo package è stato utilizzato per la creazione di alcune eccezioni necessarie, non già presenti nel linguaggio Java, con il corrispondene messaggio presente all'interno di *ExceptionMessages*.

4.1.3 outsideworld

Il package contiene tutto ciò che è esterno allo *store*.

- All'interno di *couriers* sono presenti tutte le classi con la funzione di gestione dei corrieri:
  - La classe *Courier* simula un corriere che si occupa della consegna ed aggiorna periodicamente lo stato della spedizione.
  - L'interfaccia *CourierAgency* fornisce la rappresentazione di una agenzia di corrieri che si occupa di assegnare un'istanza di ShipmentService ad un'istanza di Courier.
  - La classe *StandardCourierAgency* è l'agenzia di corrieri come concretamente implementata nell'applicativo.
- All'interno di *users* è presente una classe che rappresenta un utente.

4.1.4 store

Il package contiene tutto ciò che è interno allo *store*.

- Nel package *controller* è presente la politica di gestione del controller nell'ambito del design pattern Model View Controller.
- Nel package *events* è presente la politica di gestione del design pattern Listener.
- Nel package *factory* è presente la politica di gestione del design pattern Factory
- Nel package *purchase* è presente la parte del negozio relativa agli acquisti corrispondente ovvero a *PurchasingDepartment* e le classi *Cart*, *CartEntry*, *CatalogUtility*, *Product* che sono utilizzate per la creazione degli acquisti.
- Nel package *shipping* è presente la parte del negozio relativa alla gestione delle spedizioni: *ShippingDepartment* accoglie le richieste relative a una spedizione e ne delega la gestione a *ReturnService*, *PremiumService* e *StandardService*, il package *strategy* interno a *shipping* contiene le interfacce e le relative implementazioni usate per realizzare il pattern Strategy.
- Nel package *user* è presente la parte del negozio relativa alla gestione delle operazioni degli utenti all'interno del sistema: *UserDepartment* si occupa delle operazioni di registrazione, login e logout.
- Nel package *view* è presente la politica di getione della vista dell'utente nell'ambito del design pattern Model View Controller.
- La classe *Buttons* è utilizzata per la simulazione della navigazione dell'utente sull'interfaccia grafica dello Store.
- La classe *OperationResult* è usata per riportare al controller l'esito di una richiesta e un messaggio da comunicare all'utente.
- La classe *Store* fornisce l'effettiva implementazione dello Store come Facade.

4.1.5 Main

Nella classe *Main* sono istanziati gli oggetti necessari al corretto funzionamento dell'applicativo come il *Controller* e i dipartimenti dello *Store*, e si utilizzano i metodi della classe *Buttons* per simulare l'interazione degli utenti con il sistema.

4.2 Codice

Di seguito sono riportati frammenti di codice relativi ad alcune delle classi più importanti utilizzate nell'applicativo.

4.2.1 UserDepartment

Attributi UserDepartment presenta un unico attributo:

```
private static class UserData {
        private UserData(String password) {
            this.password = password;
            this.userIsLogged = false;
        }

        private boolean userIsLogged;
        private final String password;
    }


    private final HashMap<String, UserData> usrLoginInfo = new HashMap<>();
```

L'attributo *usrLoginInfo* è una mappa che associa ad ogni mail utente una struct UserData contenente la password dell'utente e un booleano che indica se l'utente ha eseguito il login o meno.
UserData è definita come una inner class statica di UserDepartment; è stato scelto di realizzarla come inner class poiché il suo unico scopo è quello di mantenere i campi *userIsLogged* e *password* ed è utilizzata unicamente all'interno della mappa *usrLoginInfo*.

Metodi Alcuni dei metodi implementati da UserDepartment sono:

```
    public OperationResult registerUser(String email, String password) {
        //validate email and psw, if valid check if email doesn't already
        ↪  exist
        //if it doesn't create instance of UserData to store password and
        ↪  login state
        //insert new UserData instance in map of users

        String lowerCaseEmail = email.toLowerCase();
        String emailFormatMsg = checkEmailValidity(lowerCaseEmail);
        String message;
        boolean successful = false;
        if (emailFormatMsg.equals(Constants.SUCCESS)) {
            String pswFormatMsg = checkPasswordValidity(password);
            if (pswFormatMsg.equals(Constants.SUCCESS)) {
                if (usrLoginInfo.containsKey(lowerCaseEmail)) {
                    message = Constants.EMAIL_ALREADY_USED;
                } else {
                    usrLoginInfo.put(lowerCaseEmail, new
                    ↪  UserData(password));
                    successful = true;
                    message = Constants.REGISTRATION_SUCCESS;
                }
            } else {
                message = pswFormatMsg;
```

```
            }
        } else {
            message = emailFormatMsg;
        }

        return new OperationResult(message, successful);
    }
```

Questo metodo implementa la procedura di Registrazione di un nuovo utente, come prima cosa si verifica che la struttura della mail sia valida tramite il metodo checkEmailValidity() (per semplicità si richiede semplicemente che compaia il carattere "@"), il risultato di tale controllo è una stringa, se tale stringa corrisponde ad un messaggio di successo si continua con la procedura, altrimenti la si utilizza per generare il messaggio di errore da comunicare all'utente. Se la struttura della mail è valida si procede, in modo analogo, con la verifica della struttura della password tramite il metodo checkPasswordValidity(); se tale controllo ha esito positivo si controlla che la mail utilizzata per la registrazione non sia già in uso da parte di un altro utente, se ciò non avviene si memorizzano le credenziali dell'utente nella mappa *usrLoginInfo* e si ritorna un messaggio di successo.

```
    public OperationResult loginUser(String email, String password) {
        //check user exists then recover associated UserData instance and
        ↪  set userIsLogged to true

        String lowerCaseEmail = email.toLowerCase();
        String message;
        boolean successful = false;
        if (usrLoginInfo.containsKey(lowerCaseEmail)) {
            if
            ↪  (password.equals(usrLoginInfo.get(lowerCaseEmail).password))
            ↪  {
                usrLoginInfo.get(lowerCaseEmail).userIsLogged = true;
                successful = true;
                message = Constants.LOGIN_SUCCESS;
            } else {
                message = Constants.WRONG_PSW;
            }
        } else {
            message = Constants.WRONG_EMAIL;
        }
        return new OperationResult(message, successful);
    }
```

Questo metodo implementa la procedura di Login, si verifica che la mail utilizzata per il login sia memorizzata nella mappa *usrLoginInfo*, se ciò avviene si controlla se la password inserita corrisponde a quella memorizzata, in caso di corrispondenza si imposta a *true* il valore di *userIsLogged*

e si ritorna un messaggio di successo, altrimenti si ritorna un messaggio di errore.

```
    public OperationResult logOut(String email) {
        //check if user exists and is not logged out, then log out the
        ↪  user

        String lowerCaseEmail = email.toLowerCase();
        String message;
        boolean successful = false;
        if (usrLoginInfo.containsKey(lowerCaseEmail)) {
            if (usrLoginInfo.get(lowerCaseEmail).userIsLogged) {
                usrLoginInfo.get(lowerCaseEmail).userIsLogged = false;
                successful = true;
                message = Constants.LOGOUT_SUCCESS;
            } else {
                message = Constants.ALREADY_LOGGED_OUT;
            }
        } else {
            message = Constants.WRONG_EMAIL;
        }

        return new OperationResult(message, successful);
    }
```

Tale metodo implementa la procedura di Log Out, si occupa semplicemente di impostare il valore di *userIsLogged* a *false*. Prima di eseguire questa operazione si verifica che la mail specificata corrispinda ad un utente e che tale utente abbia effettivamente eseguito il Login in precedenza.

4.2.2 ShippingDepartment

Attributi Gli attributi usati da ShippingDepartment sono:

```
        private CourierAgency courierAgency = null;
```

Un riferimento ad un'interfaccia CourierAgency, utilizzata per richiedere un corriere a cui assegnare una spedizione appena creata, il valore dell'attributo è impostato esternamente tramite un apposito metodo setter, se quando si tenta di creare una spedizione il valore dell'attributo è **null** verrà lanciata eccezione e non sarà creato alcun riferimento all'istanza del servizio di spedizione, il servizio di spedizione e la spedizione associata risulteranno quindi irraggiungibili e saranno cancellati dal Garbage Collector.

```
1 ●     private final Map<String, Map<String, ShipmentService>>
        ↪  activeServices = new HashMap<>();
```

È una mappa di mappe: ad ogni utente, identificato da una stringa contenente la relativa mail, è associata una mappa di servizi di spedizione, questi ultimi rappresentano i servizi che gestiscono le singole spedizioni appartenenti all'utente, ogni servizio è identificato da una stringa contenente l'id univoco della spedizione.
Ogni qual volta un utente si registra con successo al sistema la StoreRequest relativa alla registrazione provvede, attraverso l'interfaccia offerta da Store, a creare una nuova mappa di servizi per il nuovo utente.

```
1 ●     private int currentId = 1;
```

Un contatore utilizzato per tenere traccia del numero di spedizioni create e generare automaticamente un id per ciscuna di esse, viene incrementato ogni qual volta si riesce ad assegnare una spedizione ad un servizio senza lanciare eccezione.

Metodi Alcuni dei metodi implementati da ShippingDepartment sono:

```
1 ●     public void handlePurchase(String userEmail, String service, String
        ↪  destination, String receiver, String contents, String id) {
2           Shipment shipment = createShipment(Constants.STORE_NAME,
            ↪  receiver, Constants.STORE_ADDRESS, destination, contents,
            ↪  id);
3           try {
4               createService(service, shipment, userEmail);
5           } catch (Exception e) {
6               manageServiceCreationException(e);
7           }
8       }
9
10      public void handlePurchase(String userEmail, String service, String
        ↪  destination, String receiver, String contents) {
11          handlePurchase(userEmail, service, destination, receiver,
            ↪  contents, generateId());
12      }
```

Il metodo permette la creazione di una spedizione relativa ad un acquisto e di un servizio di spedizione per gestirla, è invocato da PurchasingDepartment una volta completato un acquisto da parte dell'utente. A scopo di testing è stato fatto overload di questo metodo per poter passare esternamente l'id da assegnare alla spedizione, fuori dai test la versione utilizzata è la seconda: si delega la prima implementazione del metodo a gestire

un acquisto utilizzando un id generato automaticamente da ShippingDepartment tramite il metodo generateId(). Il metodo createShipment() istanzia una Shipment chiamandone il costruttore e la ritorna. Il metodo createService() è descritto di seguito. Tutte le possibili eccezioni lanciate da createService(), meglio descritte di seguito, sono gestite nello stesso modo: si stampa su console un messaggio che informa dell'avvenimento, inoltre, per come è strutturato il metodo createService(), il lancio di un eccezione impedirà di memorizzare un riferimento al servizio nella mappa di servizi associata all'utente, spedizione e servizio saranno quindi eliminati dal Garbage Collector.

```
1  ●    private void createService(String typeOfService, Shipment shipment,
       ↪  String userEmail) throws InvalidParameterException,
       ↪  UnregisteredUserException, MissingAgencyException {
2           //create new ShipmentService and assign it to given Shipment
3           //if creation is successful increase currentId and fire an event
            ↪  signaling shipment creation
4           //if requestCourier() doesn't throw exception a courier will
            ↪  maintain a reference to the new instance of ShipmentService
5           //call requestCourier() as last possible method that can throw
            ↪  exception
6           //only add new ShipmentService to activeServices once sure that no
            ↪  exception will be thrown
7
8           ShipmentService newService =
            ↪  ShipmentFactory.getInstance().createService(shipment,
            ↪  typeOfService, userEmail);
9           if (!activeServices.containsKey(userEmail)) {
10              throw new UnregisteredUserException(userEmail);
11          }
12          requestCourier(newService);
13
14          activeServices.get(userEmail).put(shipment.getId(), newService);
15          currentId++;
16          ShipmentEvent event = new
            ↪  ShipmentEvent(ShipEventIdentifier.CREATED, shipment,
            ↪  userEmail);
17          ShipmentEventManager.getInstance().notify(event);
18      }
```

Come prima cosa viene creata un'istanza del Servizio di Spedizione richiesto tramite l'apposita factory, dopodichè si verifica se esiste una mappa di Servizi di Spedizione associata all'utente proprietario della spedizione, se esiste vi si inserisce la nuova spedizione creata e si procede a richiedere un corriere a cui affidare la consegna.
Le possibili eccezioni che si possono verificare sono:

- **InvalidParameterException** lanciata dal metodo createService() della factory nel caso in cui alcuni parametri utilizzati per la creazione di uno ShipmentService non siano validi (ad esempio eventuali parametri con valore *null*).

– **UnregisteredUserException** lanciata nel caso in cui non esista una lista di servizi associata all'utente che ha eseguito l'acquisto.

– **MissingAgencyException** lanciata dal metodo requestCourier() nel caso in cui il riferimento a CourierAgency posseduto da ShippingDepartment sia *null*.

Tutte le eccezioni sono gestite come specificato al punto relativo al metodo handlePurchase()

```
1 ●     public OperationResult changeAddress(String email, String shipmentID,
        ↪  String newAddress) {
2           OperationResult result;
3           result = validateServiceCredentials(email, shipmentID);
4
5           if (result.isSuccessful()) {
6               result =
                ↪  activeServices.get(email).get(shipmentID).changeAddress(
                ↪  newAddress);
7           }
8
9           return result;
10      }
```

Questo è il metodo utilizzato per delegare una richiesta relativa alla modifica di un indirizzo di consegna al servizio di spedizione associato alla spedizione desiderata, la variabile *result* è un riferimento a OperationResult, il metodo validateServiceCredentials() verifica che esista una mappa di servizi associata alla mail dell'utente e che tale mappa contenga una spedizione con l'id specificato, se ciò non avviene si procederà semplicemente a ritornare *result* che conterrà un messaggio di errore, altrimenti il valore di *result* è sostituito con quello ritornato dal metodo changeAddress() di ShipmentService. Le richieste di reso e cancellazione di una spedizione sono gestite allo stesso modo, l'unica differenza è che, in caso di credenziali corrette, si delega la gestione della richiesta rispettivamente ai metodi createReturn() e cancelService() di ShipmentService.

4.2.3 PurchasingDepartment

Attributi Gli attributi di PurchasingDepartment sono i seguenti:

```
1 ●     private final ShippingDepartment shippingDepartment;
```

Un riferimento a ShippingDepartment, il suo valore è specificato nel costruttore, è utilizzato per chiamare il metodo handlePurchase() di ShippingDepartment al termine della procedura d'acquisto.

```java
    private final Map<String, Product> catalog;
```

Una mappa che associa ad ogni codice prodotto, codificato in una stringa, un'istanza della classe Prodotto.

```java
    private final Map<String, Cart> carts = new HashMap<>();
```

Una mappa che associa ad ogni utente, identificato tramite la relativa mail, la propria istanza di Cart.

Metodi Alcuni dei metodi implementati da PurchasingDepartment sono:

```java
    public OperationResult addToCart(String productId, int quantity,
    ↪  String userEmail) {
        //if user cart and selected product both exist add or increase
        ↪  selected product by specified quantity

        String userEmailLowerCase = userEmail.toLowerCase();
        boolean successful = false;
        String operationMessage;
        String addToCartText = " added to cart";
        String noProductText = "Product does not exist";
        String noUserText = "no such user found";

        Cart userCart = carts.get(userEmailLowerCase);

        if (userCart != null) {
            Product product = catalog.get(productId);
            if (product != null) {
                userCart.increaseProduct(product, quantity);
                operationMessage = product.getName() + " x" + quantity +
                ↪  addToCartText;
                successful = true;
            } else {
                operationMessage = noProductText;
            }
        } else {
            operationMessage = userEmailLowerCase + noUserText;
        }

        return new OperationResult(operationMessage, successful);
    }
```

Metodo utilizzato per aggiungere un prodotto al carrello dell'utente o per incrementarne la quantità nel caso in cui il prodotto sia già presente nel carrello. Se l'id specificato dall'utente corrisponde ad un prodotto ed esiste un carrello associato all'utente si delega l'incremento o l'aggiunta del prodotto al metodo increaseProduct() di Cart.

```java
    public OperationResult purchase(String typeOfService, String
    ↪  userEmail, String destinationAddress, String receiver) {
        //if user cart exists and is not empty:
        //1. notifies ShippingDepartment of purchase
        //2. clears user cart
        //3. generates a result message with purchase info

        String userEmailLowerCase = userEmail.toLowerCase();
        boolean successful = false;
        String operationMessage;
        String userText = "User ";
        String purchasedText = " has purchased: ";
        String serviceText = "with service: ";
        String failedPurchaseText = "Purchase failed, cart is empty";
        String noUserText = "no such user found";
        String priceText = "total price: ";

        Cart userCart = carts.get(userEmailLowerCase);

        if (userCart != null) {
            if (!userCart.isEmpty()) {
                double total = userCart.getTotal();
                String cartContentsString = userCart.toString();
                shippingDepartment.handlePurchase(userEmailLowerCase,
                ↪  typeOfService, destinationAddress,
                        receiver, cartContentsString);
                    userCart.clear();
                    operationMessage = userText + userEmailLowerCase +
                    ↪  purchasedText + cartContentsString +
                            serviceText + typeOfService + ", " +
                            ↪  priceText + ": " + total + "€";
                    successful = true;

            } else {
                operationMessage = failedPurchaseText;
            }

        } else {
            operationMessage = userEmailLowerCase + noUserText;
        }

        return new OperationResult(operationMessage, successful);
    }
```

Il metodo è utilizzato per completare un acquisto, come prima cosa si verifica che esista un carrello associato all'utente e che non sia vuoto, se il controllo ha esito positivo si chiama il metodo handlePurchase() di ShippingDepartment per creare una spedizione associata all'acquisto, si svuota il carrello dell'utente e si genera un messaggio di riepilogo dell'acquisto, questo messaggio sarà utilizzato, insieme all'esito della procedura, per creare un'istanza di OperatonResult che sarà ritornata come risultato.

4.2.4 ShipmentService e classi derivate

Attributi Gli attributi della classe ShipmentService sono:

```
1 ●     private final int priority;
```

Un intero che indica la priorità della spedizione, utilizzato dal corriere per determinare la velocità con cui eseguire la consegna; tale attributo è inizializzato ad un valore di default dal costruttore delle classi derivate (HIGH PRIORITY = 1 per PremiumService e LOW PRIORITY = 2 per StandardService e ReturnService).

```
1 ●     private final Shipment shipment;
```

La spedizione associata a ShipmentService.

```
1 ●     private final String userEmail;
```

La mail dell'utente proprietario della spedizione, è utilizzata per specificare a quale utente comunicare un eventuale aggiornamento dello stato della spedizione.

```
1 ●     private AddressBehavior addressBehavior =
        ↪  UserAddressChanger.getInstance();
2       private CancelBehavior cancelBehavior = CancelAllower.getInstance();
3       private ReturnBehavior returnBehavior = ReturnDenier.getInstance();
```

I riferimenti alle implementazioni delle strategie di gestione delle richieste, inizialmente la cancellazione e la modifica dell'indirizzo di consegna sono sempre consentite, mentre si nega la possibilità di eseguire un reso.

Metodi Alcuni dei metodi implementati della classe ShipmentService sono:

```
1 ●     public final synchronized void updateShipmentState() {
2           //if current shipment state has a successor change state to
            ↪  successor
3           //fire an event with UPDATED id and a deep copy of shipment
4           //update service Strategies
5
6           if (shipment.getState().getNextState() != null) {
7               shipment.setState(shipment.getState().getNextState());
8               ShipmentEvent shipmentEvent = new
                ↪  ShipmentEvent(ShipEventIdentifier.UPDATED, new
                ↪  Shipment(shipment), userEmail);
```

```
            ShipmentEventManager.getInstance().notify(shipmentEvent);
            updateBehaviors();
        }

    }
```

Il metodo chiamato dal corriere che si occupa della consegna per aggiornarne lo stato, prima di tutto verifica che lo stato attuale abbia un successore, in caso affermativo aggiorna lo stato al suo successore, crea un evento inserendovi una copia della spedizione, usa il metodo notify() di ShipmentEventManager per informare il Controller dell'aggiornamento ed infine chiama il metodo updateBehaviors() per aggiornare i riferimenti agli Strategy.

```
    synchronized final void updateBehaviors() {
        changeAddressBehavior();
        changeCancelBehavior();
        changeReturnBehavior();
    }
```

Il metodo updateBehaviors() delega altri tre metodi per l'aggiornamento degli Strategy.

```
    abstract void changeAddressBehavior();

    final void changeAddressBehaviorDefault() {
        //destination address cannot be changed if another address change
        ↪  request is yet to be notified to the courier
        //once the courier has been notified of the request a new request
        ↪  may be submitted
        //destination address cannot be changed if shipment is cancelled

        if (getShipment().getState().getCurrentState().equals(
        ↪  Constants.REQUEST_RECEIVED)
                || getShipment().getState().equals(Constants.CANCELLED))
            setAddressBehavior(UserAddressDenier.getInstance());
        else if (getShipment().getState().getCurrentState().equals(
        ↪  Constants.ADDRESS_CHANGED)) {
            setAddressBehavior(UserAddressChanger.getInstance());
        }
    }
```

A scopo riassuntivo è riportato solo il metodo changeAddressBehavior(), il comportamento di changeCancelBehavior() e changeReturnBehavior() è analogo.
Il metodo controlla se lo stato della spedizione corrisponde ad uno stato di transizione a partire dal quale bisogna variare la modalità di gestione del cambio di indirizzo, in tal caso aggiorna il riferimento allo Strategy

associato a tale operazione. La classe base ShipmentService fornisce inoltre il metodo changeAddressBehaviorDefault() che esegue dei controlli su degli stati di transizione validi sia per StandardService che per PremiumService ed è chiamato all'interno degli override di changeAddressBehavior() presenti nelle classi derivate. Gli stati di transizione considerati in changeAddressBehaviorDefault() sono:

- **REQUEST RECEIVED**: lo stato indica che è stata ricevuta una richiesta di modifica dell'indirizzo di consegna e si è già provveduto a modificarlo, tuttavia la modifica deve ancora essere visualizzata dal corriere. In tale stato non è possibile eseguire un'ulteriore richiesta per modificare l'indirizzo di destinazione.
- **CANCELLED**: lo stato indica che la spedizione è stata cancellata, pertanto qualsiasi richiesta di modifica verrà negata.
- **ADDRESS CHANGED**: segue lo stato **REQUEST RECEIVED**, indica che il corriere ha visualizzato la modifica dell'indirizzo di consegna e la procedura è quindi completata, a questo punto è nuovamente possibile modificare l'indirizzo di consegna.

```
1  @Override
2  void changeAddressBehavior() {
3      if (getShipment().getState().equals(Constants.SENT))
4          setAddressBehavior(UserAddressDenier.getInstance());
5      super.changeAddressBehaviorDefault();
6  }
```

L'implementazione riportata sopra è invece l'Override del metodo eseguito da StandardService, poichè con un servizio Standard la modifica dell'indirizzo di consegna non è più ammessa a partire dallo stato **SENT** tale stato diventa un ulteriore stato di transizione in seguito al quale bisogna impedire la modifica dell'indirizzo. Dopo il controllo sullo stato **SENT** si eseguono quelli già implementati nella classe base delegando changeAddressBehaviorDefault().

```
1  synchronized final OperationResult changeAddress(String newAddress) {
2      OperationResult operationResult =
       ↪  addressBehavior.changeAddress(shipment, userEmail,
       ↪  newAddress);
3      if (operationResult.isSuccessful())
4          updateBehaviors();
5      return operationResult;
6  }
```

Il metodo implementa la procedura di gestione della richiesta di modifica dell'indirizzo, in particolare delega il metodo changeAddress() del relativo Strategy, se l'operazione è permessa, e di conseguenza viene eseguita

con esito positivo, si aggiornano gli Strategy tramite il metodo updateBehaviors(), questo perchè la modifica dell'indirizzo di consegna porta ad una transizione di stato, in particolare si passa in uno stato di "Richiesta Ricevuta" in cui si attende che il corriere visualizzi la modifica, tale stato è seguito da quello di "Indirizzo Cambiato", che costituisce uno stato di attesa prima di riprendere la consegna, infine la procedura di consegna riparte dallo stato in cui era la spedizione quando è stata ricevuta la richiesta di modifica.

4.3 Output

Nel Main è stata simulata l'interazione di tre utenti con l'applicativo, ciascuno dei quali esegue Use Case differenti.
In questa sezione sono riportate delle stampe realizzate dal programma: sono utilizzate per tenere traccia delle operazioni svolte e per verificare il corretto funzionamento della procedura di aggiornamento dello stato di una spedizione, in particolare di seguito sono riportate delle stampe relative alla sessione di uno dei tre utenti simulati.

```
LUCA@MAIL.COM
[15:14:56.261] unlogged user has requested REGISTER_REQUEST
with input [USER_EMAIL: luca@mail.com][USER_PSW: lucaPassword1]

[15:14:56.288] Operation successful with message:
"Signin done."
```

Figure 23: La prima operazione eseguita dall'utente è la richiesta di registrazione tramite un bottone dell'interfaccia, inserisce i dati negli appositi form e procede alla conferma della richiesta.
Il Sistema quindi riporta a schermo l'esito della richiesta e conferma al Client la sua registrazione.

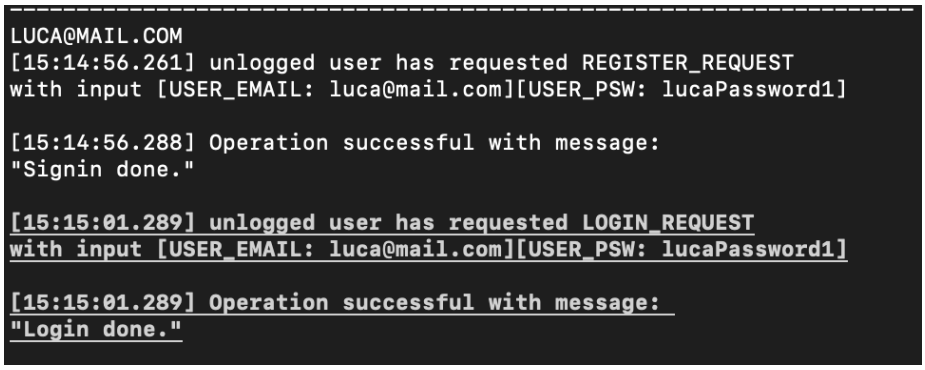


Figure 24: Il Client, effettuata la registrazione al sistema, richiede la procedura di Login attraverso l'apposito Button.
A schermo viene stampato il risultato della richiesta.

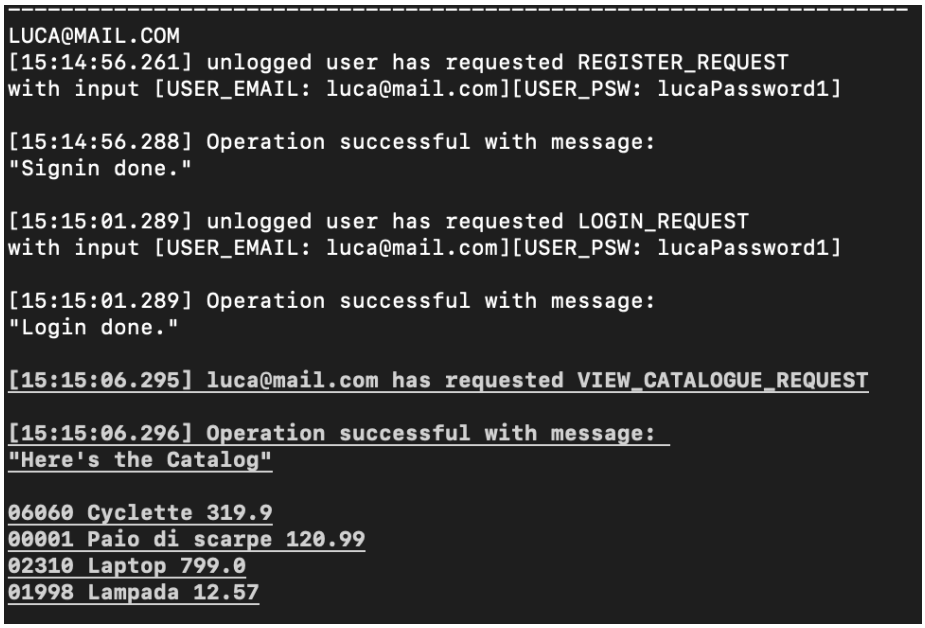


Figure 25: Il Client richiede il catalogo dello Store che quindi gli viene restituito sotto forma testuale.

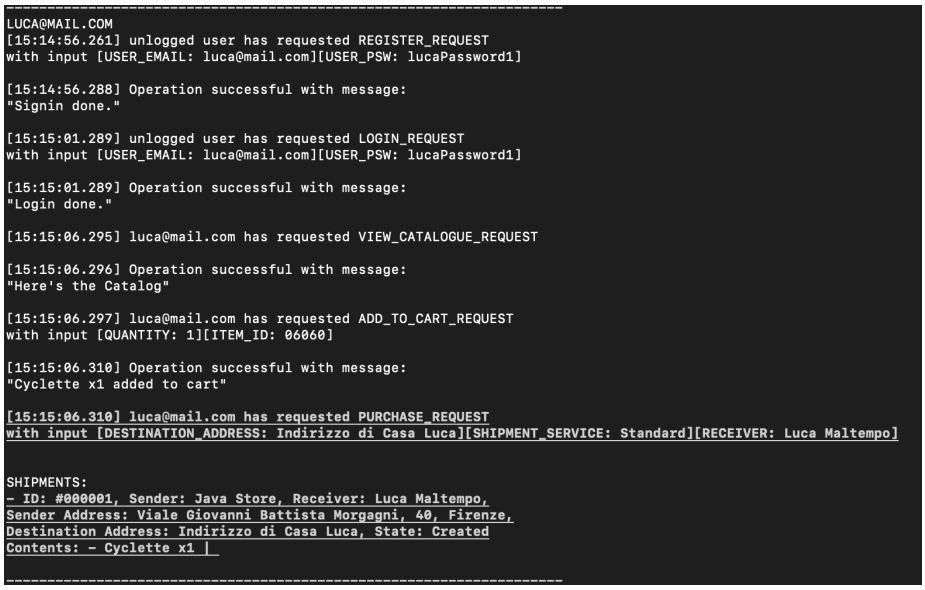


Figure 26: Il Client, visionato il catalogo, decide di acquistare una Cyclette, quindi la inserisce nel carrello e successivamente completa la Procedura di Acquisto inserendo le informazioni necessarie e confermando l'ordine.
A schermo sono riportate le operazione eseguite e lo stato delle spedizioni attive del Cliente, tutte le sue spedizioni sono aggiornate in tempo reale.

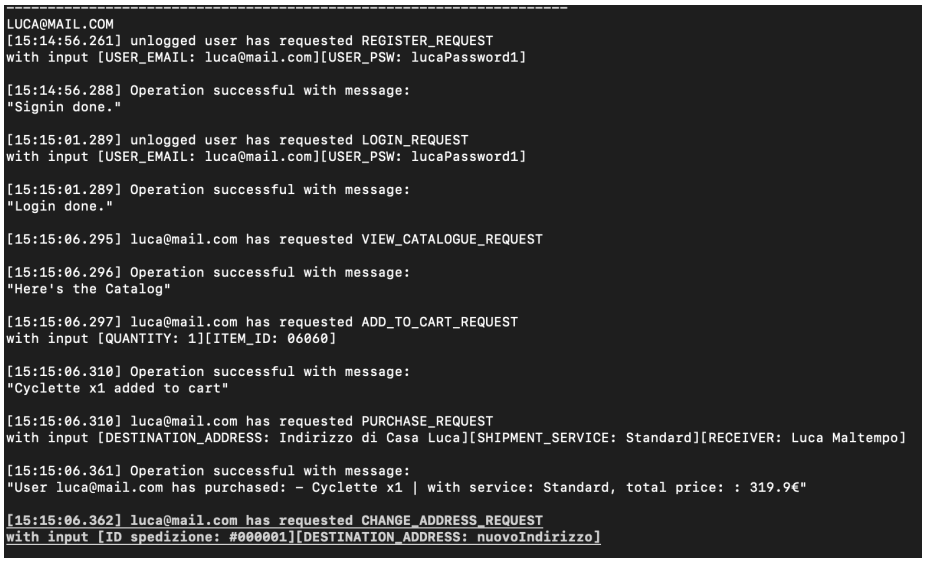


Figure 27: Il Cliente richiede un cambio di indirizzo di consegna. Il Sistema prende in carico la richiesta di cambio indirizzo.

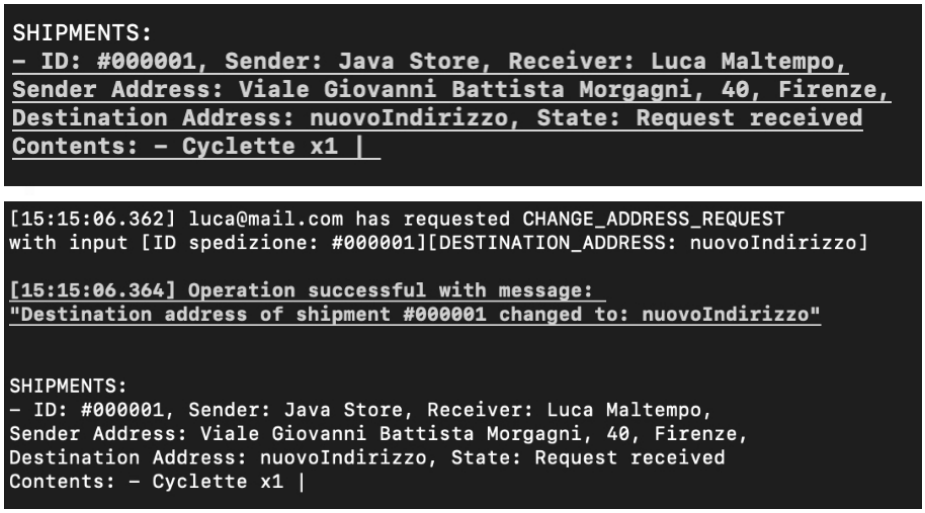


Figure 28: Il Sistema, presa in carico la richiesta di cambio indirizzo, la elabora e stampa a schermo l'esito.

```
SHIPMENTS:
- ID: #000001, Sender: Java Store, Receiver: Luca Maltempo,
Sender Address: Viale Giovanni Battista Morgagni, 40, Firenze,
Destination Address: nuovoIndirizzo, State: Address Changed
Contents: - Cyclette x1 |
```

```
SHIPMENTS:
- ID: #000001, Sender: Java Store, Receiver: Luca Maltempo,
Sender Address: Viale Giovanni Battista Morgagni, 40, Firenze,
Destination Address: nuovoIndirizzo, State: Sent
Contents: - Cyclette x1 |
```

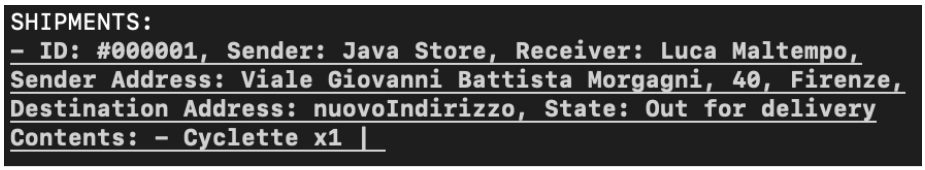
```
SHIPMENTS:
- ID: #000001, Sender: Java Store, Receiver: Luca Maltempo,
Sender Address: Viale Giovanni Battista Morgagni, 40, Firenze,
Destination Address: nuovoIndirizzo, State: In transit
Contents: - Cyclette x1 |
```

Figure 29: Il Sistema comunica al Corriere il Cambio di Indirizzo di Spedizione e stampa a schermo lo stato della Spedizione. Ogni volta che viene cambiato lo Stato della Spedizione questo viene notificato all'Utente tramite l'aggiornamento della propria vista.

```
[15:15:14.368] luca@mail.com has requested CHANGE_ADDRESS_REQUEST
with input [ID spedizione: #000001][DESTINATION_ADDRESS: indirizzoNuovo]

[15:15:14.369] Operation failed with message:
"Destination address of shipment #000001 cannot be changed as state is In transit"
```

Figure 30: Il Cliente richiede nuovamente il cambio di indirizzo di consegna. Il Sistema, elaborata la richiesta, stampa a schermo l'esito negativo e comunica al Cliente il motivo per cui la Richiesta non è andata a buon fine.



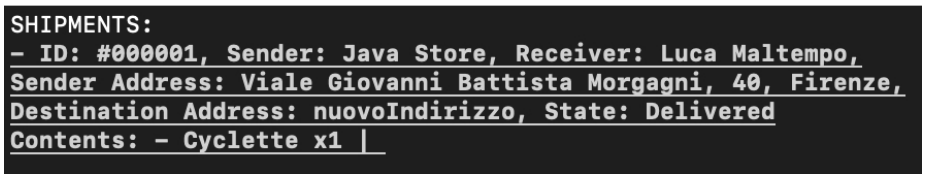


Figure 31: Il Corriere aggiorna lo Stato della Spedizione, il Sistema aggiorna in tempo reale la vista del Client. La Spedione viene consegnata al Client.

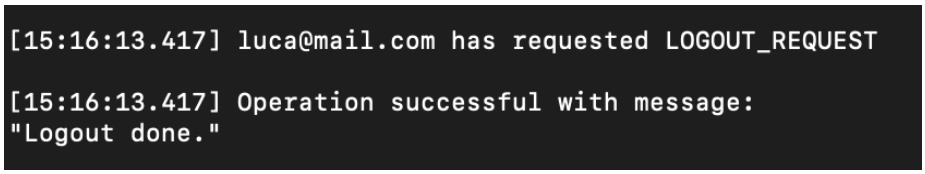


Figure 32: Il Client richiede il Logout dal Sistema.