Documentazione

Progetto di  
Ingegneria del Software

Fabiola Fabretti, VR437439  
Andrea Toffaletti, VR437439  
  
  
Luglio 2020

Sommario

[Requisiti ed interazioni utente-sistema 3](#_Toc45154208)

[Casi d’uso e diagrammi di sequenza 3](#_Toc45154209)

[Note generali sui casi d’uso 3](#_Toc45154210)

[Caso 1: Acquisto di una spesa 3](#_Toc45154211)

[Caso 2: Modifica del profilo 3](#_Toc45154212)

[Caso 3: Modifica informazioni sui prodotti. 4](#_Toc45154213)

[Progettazione dell’architettura e implementazione del sistema. 5](#_Toc45154214)

[Note sul processo di sviluppo 5](#_Toc45154215)

[Scelte tecniche e implementative 5](#_Toc45154216)

[Pattern architetturali utilizzati 7](#_Toc45154217)

[Pattern MCV 7](#_Toc45154218)

[Design pattern utilizzati 8](#_Toc45154219)

[Pattern iterator 8](#_Toc45154220)

[Pattern observer 8](#_Toc45154221)

[Pattern singleton(TBI!!!) 8](#_Toc45154222)

[Data Access Object Pattern 8](#_Toc45154223)

[Diagrammi di sequenza del software 9](#_Toc45154224)

[Attività di test e validazione 10](#_Toc45154225)

# Requisiti ed interazioni utente-sistema

## Casi d’uso e diagrammi di sequenza

### Note generali sui casi d’uso

Il sistema proposto deve poter essere utilizzato da due tipi di utente:

* Il cliente che usufruisce del servizio, da qui in avanti denominato *User*
* Il dipendente che gestisce il servizio, da qui in avanti denominato *Worker*.

Questi due tipi di utente saranno i nostri attori. Entrambe le categorie hanno delle credenziali predefinite con cui possono effettuare l’autenticazione ed utilizzare il software nella modalità a loro preposta. Il software rileva automaticamente se le credenziali appartengono a un *User* o a un *Worker*, e si avvia nella modalità appropriata.

Poiché si tratta di un prototipo, la versione attuale del *software* non permette di registrare nuovi utenti, salvo modifiche “manuali” dei file di salvataggio delle credenziali.

### Caso 1: Acquisto di una spesa

Gli utenti devono poter essere in grado di comporre il proprio carrello e confermare l’acquisto.

***Attori:*** *User.*

***Precondizioni:*** *L’utente deve essersi autenticato con una combinazione User.*

***Passi:***

1. *Il medico accede al sistema*
2. *Il medico è introdotto all’interfaccia di base3. Il medico accede all’interfaccia per le segnalazioni 4.*
   1. *Il medico inserisce i dati relativi al paziente*
   2. *Il medico seleziona un paziente tra quelli a suo carico*

*5.*

* 1. *Il medico inserisce una nuova reazione avversa*
  2. *Il medico seleziona una reazione avversa dalla memoria del*

*sistema*

1. *Il medico inserisce la data della reazione avversa*

*Il medico conferma la segnalazione****Postcondizioni:*** *L’ordine è aggiunto alla lista degli ordini e allo storico dell’User. Il carrello viene svuotato.*

### Caso 2: Modifica del profilo

***Attori:*** *User,Worker*

***Precondizioni:*** *L’utente deve essersi autenticato.*

***Passi:***

1. *Il medico accede al sistema*
2. *Il medico è introdotto all’interfaccia di base3. Il medico accede all’interfaccia per le segnalazioni 4.*
   1. *Il medico inserisce i dati relativi al paziente*
   2. *Il medico seleziona un paziente tra quelli a suo carico*

*5.*

* 1. *Il medico inserisce una nuova reazione avversa*
  2. *Il medico seleziona una reazione avversa dalla memoria del*

*sistema*

1. *Il medico inserisce la data della reazione avversa*

*Il medico conferma la segnalazione****Postcondizioni:*** *Se l’utente esce cliccando su Salva, le modifiche vengono salvate su file;* altrimenti ogni cambiamentto viene annullato.

### Caso 3: Modifica informazioni sui prodotti.

***Attori:*** *Worker.*

***Precondizioni:*** *L’utente deve essersi autenticato con una combinazione User.*

***Passi:***

1. *Il medico accede al sistema*
2. *Il medico è introdotto all’interfaccia di base3. Il medico accede all’interfaccia per le segnalazioni 4.*
   1. *Il medico inserisce i dati relativi al paziente*
   2. *Il medico seleziona un paziente tra quelli a suo carico*

*5.*

* 1. *Il medico inserisce una nuova reazione avversa*
  2. *Il medico seleziona una reazione avversa dalla memoria del*

*sistema*

1. *Il medico inserisce la data della reazione avversa*

*Il medico conferma la segnalazione****Postcondizioni:*** *Se il Worker preme Salva, ogni modifica è salvata su file e vengono gestiti eventuali cambi di sezione. Altrimenti, ogni cambiamento viene annullato.*

# Progettazione dell’architettura e implementazione del sistema.

### Note sul processo di sviluppo

Il processo di sviluppo si è rivelato sicuramente più complesso del previsto a causa delle pesanti restrizioni imposteci dalla situazione mondiale corrente, che non ci hanno permesso di vederci di persona e lavorare insieme. Questo ci ha imposto di utilizzare strumenti di programmazione a distanza, quali:

1. *Plugin* “*Saros*” per *Eclipse*, che permette di programmare in *pair-programming* visualizzando le modifiche al codice in tempo reale;
2. Strumenti di comunicazione a distanza come *Zoom*, *Discord*, *WhatsApp* o anche delle semplici telefonate;
3. *Git* e *GitHub*, che ci hanno permesso di lavorare in parallelo e gestire con facilità le versioni del software.

In generale, abbiamo scritto il codice sia in *pair-programming* che in parallelo; nel primo caso utilizzando il plugin menzionato al punto 1, e nel secondo caso caricando le nostre versioni su Git per poi contattarci a vicenda per illustrare all’altro le modifiche effettuate e il funzionamento del codice nuovo.

In generale abbiamo cercato di seguire un approccio di tipo *agile* ed incrementale, intrecciando fortemente fra di loro le varie fasi ma cercando di seguire l’ordine progettazione->implementazione->validazione all’interno dell’implementazione di un singolo modulo.

E’ necessario notare che abbiamo comunque preferito, nelle fasi iniziali, soffermarci di più sulla fase di progettazione e stabilire un ordine di priorità di lavoro, per permetterci di orientarci meglio nelle fasi successive ed essere certi di aver compreso i requisiti del programma. Questo non è strettamente previsto nello sviluppo *agile*-incrementale, ma ha reso più semplice organizzare il lavoro.

Non sono mancate fasi di *refactoring*, spesso rivelatesi necessarie a seguito di test o, talvolta, dopo aver implementato altre funzioni che hanno reso evidente la possibilità o necessità di semplificare parti di codice già scritto.   
Un esempio di questo tipo è stata l’implementazione della visualizzazione dei pannelli dei prodotti: inizialmente avevamo gestito questo modulo all’interno della visualizzazione *User*, ma implementando il lato *Worker* ci siamo resi conto della necessità di poter riutilizzare e generalizzare quel codice per gestire anche la visualizzazione in formati e casi diversi. Per questo, abbiamo deciso di dedicare del tempo al *refactoring* di questo modulo, creando una classe apposita che gestisse autonomamente e in maniera versatile la visualizzazione dei pannelli.

### Scelte tecniche e implementative

Il sistema è stato sviluppato in linguaggio Java, dunque utilizzando le tecniche di modellazione a oggetti.

#### Librerie aggiuntive

Oltre alle classi scritte da noi, ci siamo serviti di alcune librerie aggiuntive:

* JavaFX : [ <https://www.oracle.com/java/technologies/javase/javafx-overview.html> ]  
  è una delle due librerie Java consigliate dalla professoressa Migliorini per la realizzazione di un’interfaccia grafica.   
  Abbiamo scelto di utilizzare JavaFX in quanto ci è sembrata estremamente più intuitiva da usare per numerosi motivi:
  + Gestione (quasi del tutto) automatica del sistema dei Listener, rimpiazziato con i ben più intuitivi Controller.
  + Possibilità di integrazione di codice FXML, e conseguente esistenza di software che permettono di realizzare graficamente le interfacce e tradurle in automatico; in particolare, noi ci siamo serviti di SceneBuilder.
  + Possibilità di aggiunta di file CSS, che permettono di definire facilmente e una sola volta lo stile dell’intero programma.
  + Esistenza di librerie grafiche esteticamente gradevoli, come JFoenix.
* JFoenix : [ <https://github.com/jfoenixadmin/JFoenix> ]  
  Si tratta di una libreria open source che implementa il Google Material Design usando componenti di JavaFX. Il suo fine è puramente estetico.
* minimalJson : [ <https://github.com/ralfstx/minimal-json> ]  
  Si tratta di una libreria che esegue il parsing e la scrittura di file JSON. La abbiamo scelta in quanto è leggera e facile da importare. Inoltre, secondo le statistiche pubblicate dall’autore, è molto più efficiente di altre librerie che svolgono la stessa funzione.

## Pattern architetturali utilizzati

### Pattern MCV

Dal punto di vista architetturale abbiamo applicato il *pattern* *MVC*. Questa architettura è caratterizzata da tre componenti:

* **Modello**: è la sottoparte del sistema che riguarda i dati e la loro rappresentazione in memoria. Lo abbiamo realizzato nel *package* “*data*”.
* **Vista**: è la sottoparte del sistema che riguarda la rappresentazione visiva del modello. Per realizzarla abbiamo deciso di utilizzare *JavaFX*, arricchita con la libreria grafica *JFoenix*. Per permetterci di sfruttare al massimo le potenzialità di entrambe, abbiamo affidato a *SceneBuilder* la realizzazione del codice “grafico” in *FXML*.
* **Controllore**: è la sottoparte del sistema che definisce come il sistema deve reagire agli *input*. Li abbiamo realizzati attraverso dei Controllori, identificati dal suffisso   
  *-Controller­* nel nome della classe, e in generale abbiamo associato un controllore a ciascuna “unità grafica” – per esempio, a ciascun pannello o a ciascuna finestra.

In generale, le azioni dell’utente sono rilevate dai *listener* interni di *JavaFX*. Questi *listener* chiamano il metodo ad essi associato, il quale apporta le modifiche necessarie al Modello ed eventualmente aggiorna la Vista di conseguenza.

L’unica eccezione a questo meccanismo sono state le *ComboBox*, ovvero i cosiddetti “*menu* a tendina”. A differenza della maggior parte dei *nodes* disponibili in *JavaFX* e *JFoenix*, non sono dotate di un *On-Action Listener*, e abbiamo definito via lambda-espressione un osservatore customizzato che avviasse il metodo da noi richiesto.   
E’ possibile osservare questo meccanismo nella classe che gestisce le barre dei filtri, ovvero *FilterBarController*:

chboxOrder.getSelectionModel().selectedItemProperty().addListener((ObservableValue<? **extends** String> observable, String oldValue, String newValue) -> Globals.*currentView*.changeOrder(newValue) );

Da questo *snippet* di codice si nota come al nodo venga associato un Listener, il cui comportamento sarà chiamare il metodo changeOrder() quando rileva un cambiamento del valore selezionato.

La scelta è ricaduta su questo *pattern* per diverse motivazioni:

* Le componenti di Vista e Controllore sono già parzialmente implementate all’interno di *JavaFX*, ovvero sulla libreria che abbiamo deciso di utilizzare;
* Avevamo bisogno di poter rappresentare gli stessi dati in più modi diversi a seconda dell’operazione che stavamo svolgendo – per esempio, il *Worker* deve poter visualizzare i prodotti in modo diverso rispetto all’*User*, poiché le operazioni che i due possono svolgere sono differenti.

## Design pattern utilizzati

Abbiamo deciso di utilizzare alcuni dei pattern visti in classe poiché sono metodi testati ed efficaci per risolvere problemi comuni. In particolare:

### Pattern iterator

Non è stata una vera e propria scelta, ma avendo utilizzato numerose Collections e oggetti Iterable in generale ci sembrava doveroso far notare che esse sono implementate utilizzando questo pattern.

Pattern observer  
Anche in questo caso, si tratta un pattern utilizzato internamente nell’implementazione dei nodes di JavaFX. Inoltre, come già fatto notare, lo abbiamo implementato manualmente nel caso delle ComboBox.

### Data Access Object Pattern

Per quanto riguarda la rappresentazione in memoria dei dati da gestire, ci siamo ispirato al DAO pattern apportandovi qualche semplificazione.  
In generale, questo pattern prevederebbe tre parti:

* Data Access Object Interface: interfaccia che definisce le operazioni da eseguire su uno o più oggetti
* Data Access Object Concrete Class: classe concreta che implementa l’interfaccia sopradefinita, ed è responsabile di recuperare i dati
* Model Object: è un POJO (Plain Old Java Object) che rappresenta l’oggetto in memoria.

Ci siamo ispirati a questo modello per quanto riguarda l’avere una classe specifica che si preoccupasse di recuperare le informazioni dai file e rappresentarle in maniera opportuna in memoria. A differenza del pattern originale, tuttavia, abbiamo ritenuto superfluo definire anche un’interfaccia, e abbiamo deciso di riunire tutti i “caricatori” in un’unica classe. Inoltre, abbiamo esteso lo stesso concetto anche al processo opposto, ovvero al salvataggio.

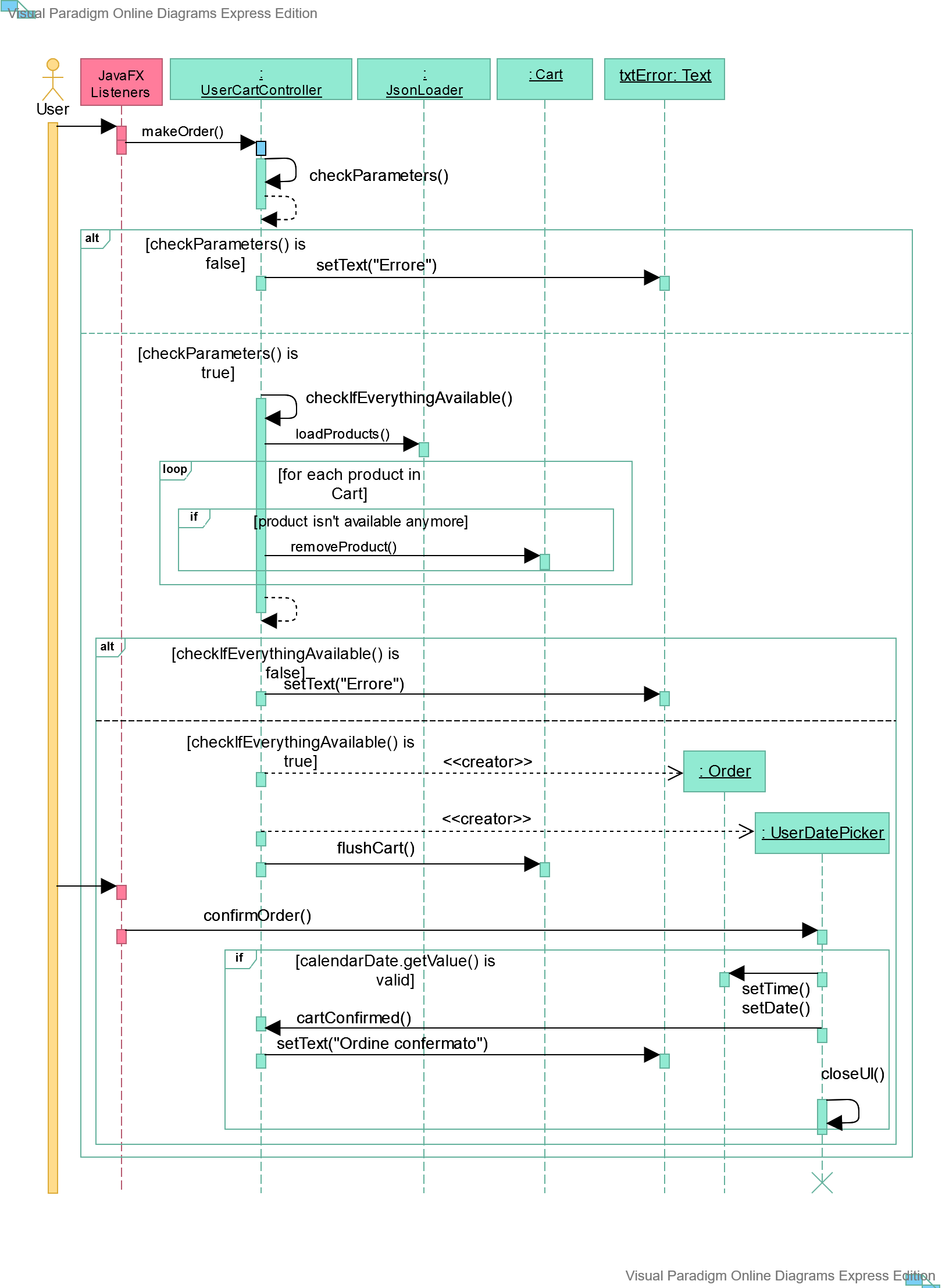
Di fatto abbiamo creato due classi JsonSaver e JsonLoader, le quali si occupano di recuperare le informazioni necessare dai file e tradurli in POJO in memoria. Esse riuniscono “più” DAO pattern, dato che entrambe permettono di gestire la generazione (e il salvataggio) di POJO di più tipi.

### Nota: Singleton Pattern

Lavorando sulla classe che si occupa di realizzare graficamente i pannelli che rappresentano i prodotti,ci siamo resi conto che – almeno in teoria – essa si presterebbe molto bene al singleton pattern: in quanto classe “simbolica” che si limita a generare pannelli, averne una sola istanza avrebbe dovuto essere più che sufficiente.   
Tuttavia, alla fine abbiamo optato per non applicare il pattern e permettere la creazione di istanze multiple poiché si creavano conflitti nel caso in cui una visualizzazione rimaneva visibile dopo la creazione di un’altra – ad esempio, quando un *User* apre il carrello dopo aver visualizzato i prodotti, ne abbiamo due contemporaneamente! Avere una sola istanza di ProductViewer ci porterebbe a perdere tutte le informazioni sulla prima schermata, rendendo impossibile svolgervi operazioni successivamente.

* In più punti del codice, ovvero a partire da JFrame diversi, si è utilizzato TerapiaFrame (con annesso TerapiaListener) per raccogliere informazioni riguardo a nuove terapie farmacologiche. Queste informazioni sono state poi passate al JFrame chiamante utilizzando un metodo comune (ogni JFrame chiamante implementa l’interfaccia FrameConTerapie). Tuttavia, ogni JFrame tratta in modo diverso la terapia costruita. Per gestire questa variabilità di risposte per lo stesso metodo chiamato, si è utilizzato il pattern strategy: il contesto (TerapiaFrame) chiamerà un metodo comune (addTerapia()) che a seconda della strategia (FrameConTerapia) sarà implementato in maniera differente. 🡪 LO ABBIAMO FATTO INVOLONTARIAMENTE, MA NON LO ABBIAMO VISTO IN CLASSE ME THINKS
* Hhhh

## Diagrammi di sequenza del software

Seguono di diagrammi di sequenza che descrivono come si comporta il sistema nelle interazioni più complesse, ovvero che coinvolgono più classi. 

# Attività di test e validazione

Per verificare la solidità del software, abbiamo effettuato test di vario tipo. Blahblahblah.