

UNIVERSITÀ DEGLI STUDI DI VERONA



## RELAZIONE E DOCUMENTAZIONE

## ELABORATO INGEGNERIA DEL SOFTWARE

A.A. 2016-2017

*Music Store 1.0*

BERTONCELLI GIOVANNI (VR399929)

GIRELLI ALBERTO (VR397173)

RIGHI EDOARDO (VR398499)

# SOMMARIO

## DESCRIZIONE DEL PROGETTO

- *SCELTE PROGETTUALI.....03*
- *STRUMENTI DI SVILUPPO UTILIZZATI.....03*
- *GERARCHIA DEL CODICE.....04*

## DOCUMENTAZIONE PER LO STUDIO DEI REQUISITI DEL SOFTWARE

- *USE CASE DIAGRAM E RELATIVE SCHEDE DI SPECIFICA.....06*
- *SEQUENCE DIAGRAM (DEI CASI D'USO PRINCIPALI).....11*
- *ACTIVITY DIAGRAM.....16*

## DOCUMENTAZIONE TECNICA (IMPLEMENTAZIONE DEL SOFTWARE)

- *CLASS DIAGRAM.....18*
- *SEQUENCE DIAGRAM.....19*
- *DESIGN PATTERN UTILIZZATI.....23*
- *INTERFACCIA GRAFICA (GUI).....26*

## TESTING DEL SOFTWARE

- *UNIT TESTING.....38*
- *INTEGRATION TESTING.....39*
- *ALTRE MODALITÀ DI TEST.....40*

# DESCRIZIONE DEL PROGETTO

Il progetto, secondo i *requisiti forniti*<sup>1</sup>, prevedeva di realizzare un sistema informativo per gestire le informazioni relative alla gestione di un negozio virtuale di CD e DVD musicali. Tramite software di supporto adeguati alla programmazione ad oggetti era necessario progettare e implementare, tramite anche l'utilizzo di pattern di progettazione, un prototipo, avente interfaccia grafica, in grado di soddisfare i requisiti del software da sviluppare.



Figura 1: Music Store Logo

## SCELTE PROGETTUALI

Il prototipo che è stato realizzato fornisce un'interfaccia grafica completa e funzionale per il cliente, mentre per il personale addetto alla gestione del catalogo l'interfaccia e i dettagli progettuali non sono stati implementati<sup>2</sup>, tuttavia ci sono molti richiami alle specifiche del progetto. Un esempio: se un articolo scende sotto i 2 pezzi disponibili viene visualizzato un messaggio sulla console dell'ambiente di sviluppo (Eclipse nel nostro caso o il terminale di esecuzione) con l'ID e il nome del prodotto in esaurimento.

Si è deciso di implementare il progetto tramite l'utilizzo del linguaggio Java<sup>3</sup> correlato alla libreria Java Swing<sup>4</sup> per la creazione dell'interfaccia grafica e alla libreria JUnit<sup>5</sup> per il testing finale del software. A questo riguardo sono stati fatti solo i test riguardanti le classi costituenti del progetto, mentre le classi della GUI (separate in un apposito package) sono state testate manualmente<sup>6</sup>.

## STRUMENTI DI SVILUPPO UTILIZZATI

Per lo sviluppo del progetto sono stati impiegati vari software e strumenti che possono essere suddivisi per tipo di utilizzo:

- Scrrittura e condivisione del codice
  - Eclipse Neon Java IDE - <https://eclipse.org>  
(Ambiente di sviluppo utilizzato per il codice)
  - Bitbucket.org – <https://bitbucket.org>  
(Repository GIT online per il code management per team di sviluppo)
  - GitKraken – <https://www.gitkraken.com>  
(GIT Client per Windows, Mac e Linux per la gestione repository GIT)
- Test del codice
  - JUnit Test - <http://junit.org>  
(Test mirati al codice)

<sup>1</sup> Si possono consultare i file allegati alla documentazione, con la consegna originale del progetto

<sup>2</sup> Tuttavia nella progettazione del software i dettagli dei requisiti riguardanti la gestione del negozio da parte del personale sono stati considerati e inseriti quindi nella documentazione pre-sviluppo.

<sup>3</sup> Cfr. <https://docs.oracle.com/javase/8/docs/api/>

<sup>4</sup> Cfr.: <http://docs.oracle.com/javase/7/docs/technotes/guides/swing/index.html>

<sup>5</sup> Cfr.: <http://junit.org>

<sup>6</sup> Per i dettagli si rimanda al paragrafo “Attività di test”

- ECLemma - <http://www.eclemma.org>  
(*Componente aggiuntivo di Eclipse per il code coverage*)
- Realizzazione diagrammi e documentazione
  - StarUML – <http://staruml.io>  
(*Software per la progettazione orientata agli oggetti tramite UML*)
  - Microsoft Visio  
(*Software per la realizzazione di diagrammi*)
  - Microsoft Word  
(*Software per l'editing di documenti di testo*)

Soprattutto nella parte finale dell'implementazione ci siamo affidati a strumenti come *Git* e *GitKraken* per mantenere in modo semplice e reciproco il codice e aggiornarlo ad ogni modifica dando così un reale flusso di sviluppo del codice e del software stesso.

## GERARCHIA DEL CODICE

Nella pagina successiva viene presentato un riassunto della disposizione delle classi nel progetto e della loro gerarchia nei vari package, che sarà dettagliata nella documentazione e nel Class Diagram. In seguito si riporta la legenda per la corretta comprensione dell'immagine<sup>7</sup>:

- ⌚ *it.RGB.is.Classes*: Classi di progettazione<sup>8</sup>
  - Classi principali
  - Classi per gli artisti
  - Classi per database
  - Enumerazioni
  - Classi di progetto rimanenti
- ⌚ *it.RGB.is.Exceptions*: Classi per le eccezioni del software
  - Classi per eccezioni *unchecked* (estendono `IllegalArgumentException`)
  - **Classe per la gestione delle eccezioni critiche (con *logging* degli errori su files)**
  - Classi per eccezioni *checked* che estendono `Exception`
  - **Classi per eccezioni *checked* che estendono `CriticalException`**
- ⌚ *it.RGB.is.GUI*: Classi per l'interfaccia grafica
  - Classi principali GUI
  - Enumerazioni
  - Classi Listener
  - Classi di interfaccia grafica
  - Modelli delle tabelle

La cartella resources/ contiene tutte le immagini necessarie alla corretta visualizzazione dell'interfaccia grafica.

---

<sup>7</sup>In questa legenda vengono omessi i package di test del codice

<sup>8</sup>Termine forse un po' informale per definire tutte le classi ereditate direttamente dai requisiti e dalla progettazione stessa

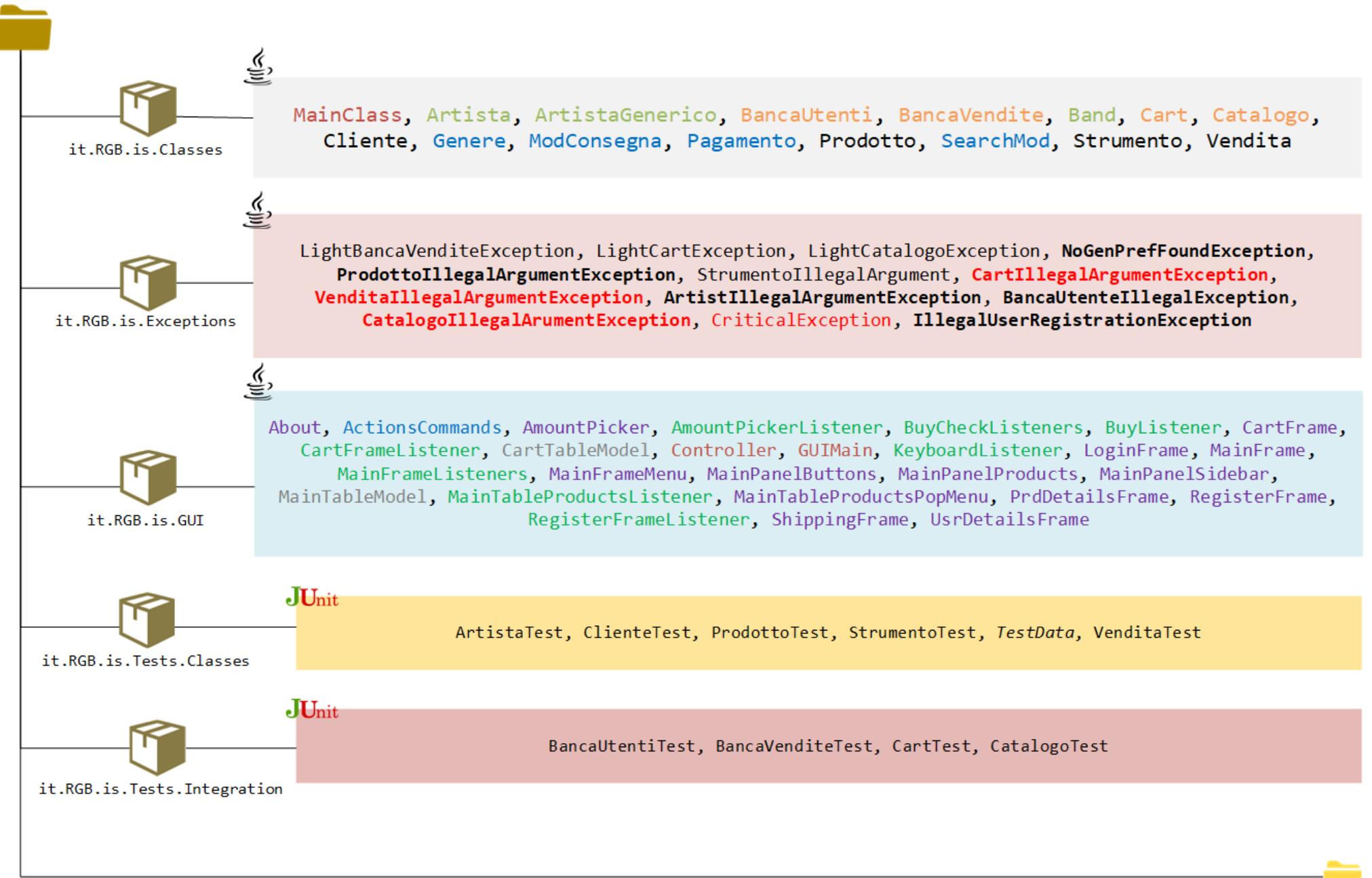


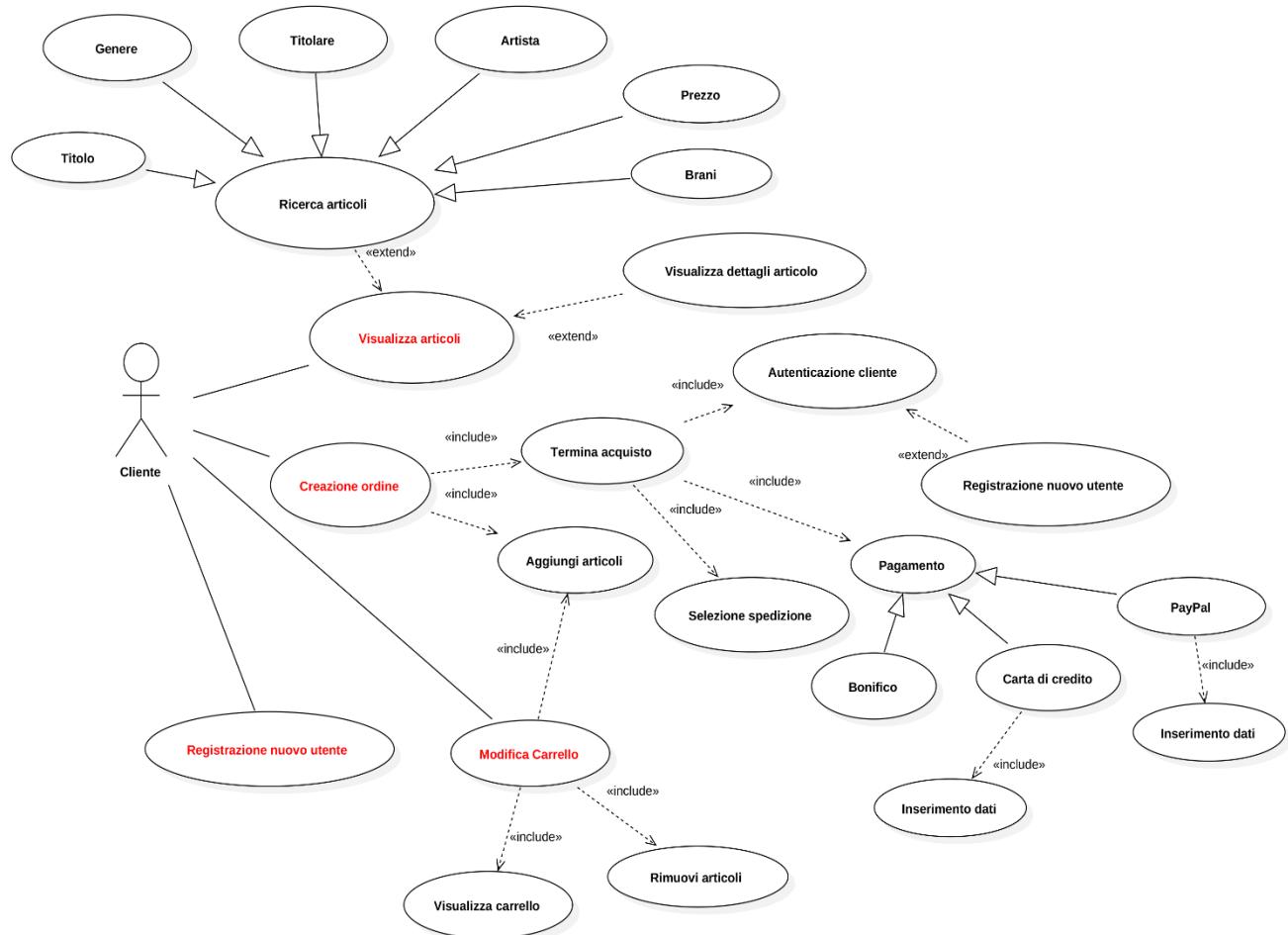
Figura 2: Riassunto gerarchia classi e package

# DOCUMENTAZIONE PER LO STUDIO DEI REQUISITI DEL SOFTWARE

Come primo passo nella realizzazione del progetto, utilizzando StarUML sono stati modellati dei diagrammi per studiare i requisiti del software, necessari per ottenere una precisa concezione di quello che lo sviluppo effettivo del codice deve trattare, e in quale modalità.

## USE CASE DIAGRAM

Nelle prossime pagine vengono descritti i diagrammi dei casi d'uso comprensivi delle loro schede di specifica.



*Figura 3: Use case diagram del cliente.*

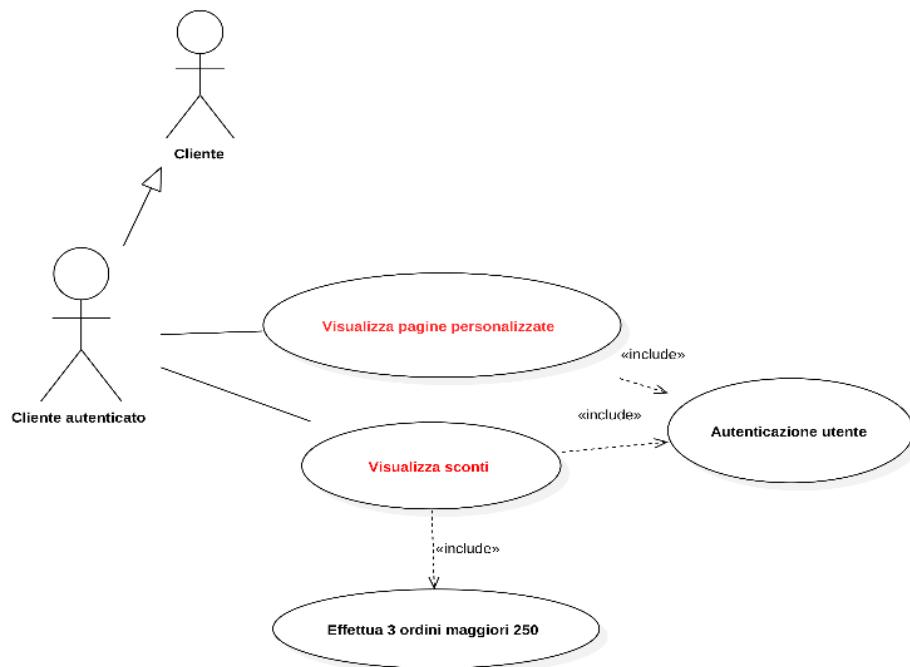
Il cliente può visualizzare articoli, anche attraverso una ricerca personalizzata, e ottenerne i dettagli, modificare il contenuto del carrello e creare un ordine; per far ciò deve essere autenticato dal sistema, registrandosi se ancora non lo ha fatto. L'ordine è completato dopo che il cliente ha inserito le sue informazioni di spedizione e pagamento.

## *SPECIFICA DEL CASO D'USO (UC1, UC2, UC3)*

<b>Caso d'uso: Visualizzazione articoli</b>
ID: UC1
Attori: Cliente
Pre-condizioni: Nessuna
<p style="text-align: center;"><b>Sequenza di eventi:</b></p> <ul style="list-style-type: none"> <li>- Un utente può visualizzare gli articoli in modo sparso</li> <li>- Si possono visualizzare gli articoli dettagliatamente</li> <li>- Tramite ricerca specifica l'utente può avere risultati mirati</li> </ul>
Post-condizioni: Nessuna

<b>Caso d'uso: Creazione ordine</b>
ID: UC2
Attori: Cliente
Pre-condizioni: Devono esserci articoli nel carrello
<p style="text-align: center;"><b>Sequenza di eventi:</b></p> <ul style="list-style-type: none"> <li>- Termina l'acquisto</li> <li>- L'utente si registra oppure accede</li> <li>- Seleziona una modalità di pagamento</li> <li>- Seleziona i dettagli di spedizione</li> </ul>
Post-condizioni: Conferma dell'ordine

<b>Caso d'uso: Modifica carrello</b>
ID: UC3
Attori: Cliente
Pre-condizioni: Nessuna
<p style="text-align: center;"><b>Sequenza di eventi:</b></p> <ul style="list-style-type: none"> <li>- L'utente visualizza/modifica/elimina elementi dal carrello</li> </ul>
Post-condizioni: Nessuna

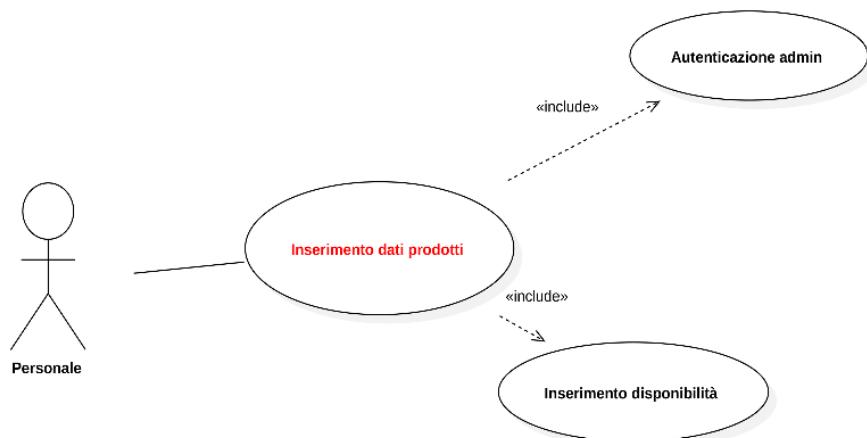


*Figura 4: Use case del cliente autenticato.  
Il cliente può visualizzare delle pagine personalizzate per lui e degli sconti, se idoneo.*

## SPECIFICA DEL CASO D'USO (UC4, UC5)

Caso d'uso: Visualizzazione pagine personalizzate
ID: UC4
Attori: Cliente autenticato
Pre-condizioni: Il cliente è registrato e ha effettuato l'accesso
<b>Sequenza di eventi:</b> - L'utente può visualizzare pagine fornite dal sistema con suggerimenti basati sullo storico dei suoi acquisti
Post-condizioni: Nessuna

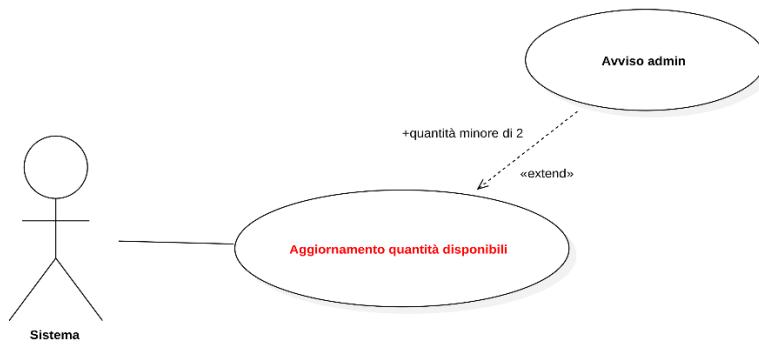
Caso d'uso: Visualizzazione sconti
ID: UC5
Attori: Cliente autenticato
Pre-condizioni: Il cliente è registrato e ha effettuato l'accesso e ha effettuato almeno 3 ordini superiori ai 250 euro l'uno entro l'anno
<b>Sequenza di eventi:</b> - L'utente può ottenere sconti e spedizioni gratuite per i suoi prossimi acquisti
Post-condizioni: Nessuna

*Figura 5: Use case del personale.*

*Il personale può aggiungere nuovi prodotti al catalogo o modificare la disponibilità di quelli presenti.*

## SPECIFICA DEL CASO D'USO (UC6)

Caso d'uso: Inserimento dati prodotti
ID: UC6
Attori: Personale
<b>Pre-condizioni:</b> L'utente è autenticato come amministratore dopo l'autorizzazione del negozio
<b>Sequenza di eventi:</b> - Il personale può inserire nuovi prodotti e le disponibilità relative
<b>Post-condizioni:</b> Nessuna



*Figura 6: Use case del sistema.  
Il sistema notifica al personale la carenza di prodotti.*

## SPECIFICA DEL CASO D'USO (UC7)

Caso d'uso: Aggiornamento quantità disponibili
ID: UC7
Attori: Sistema
Pre-condizioni: Nessuna
<b>Sequenza di eventi:</b> <ul style="list-style-type: none"> <li>- Il sistema tiene monitorati le disponibilità dei prodotti</li> <li>- Il sistema avvisa il personale per disponibilità inferiore a 2 pezzi</li> </ul>
Post-condizioni: Il personale è al corrente della carenza di disponibilità

## SEQUENCE DIAGRAM

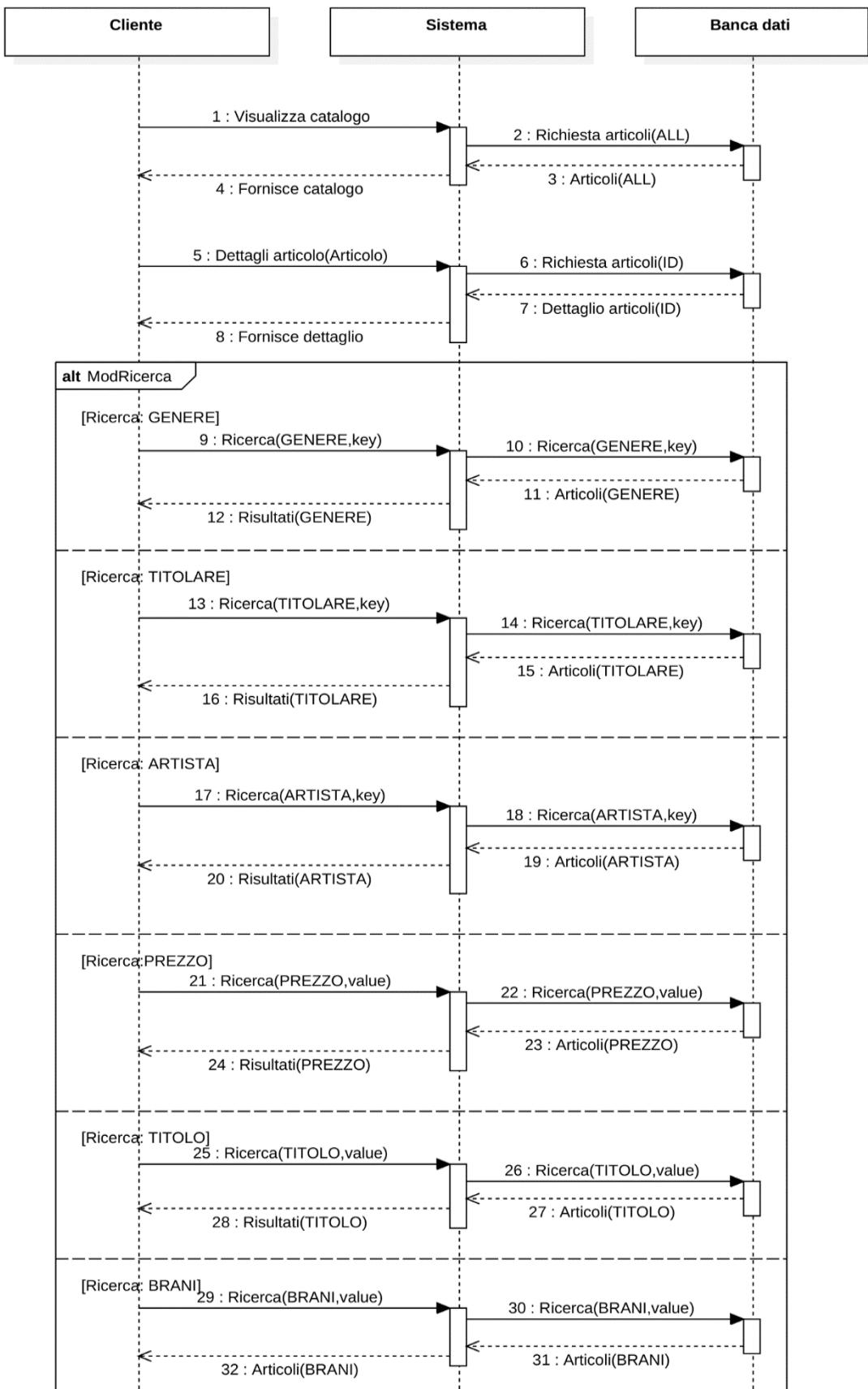
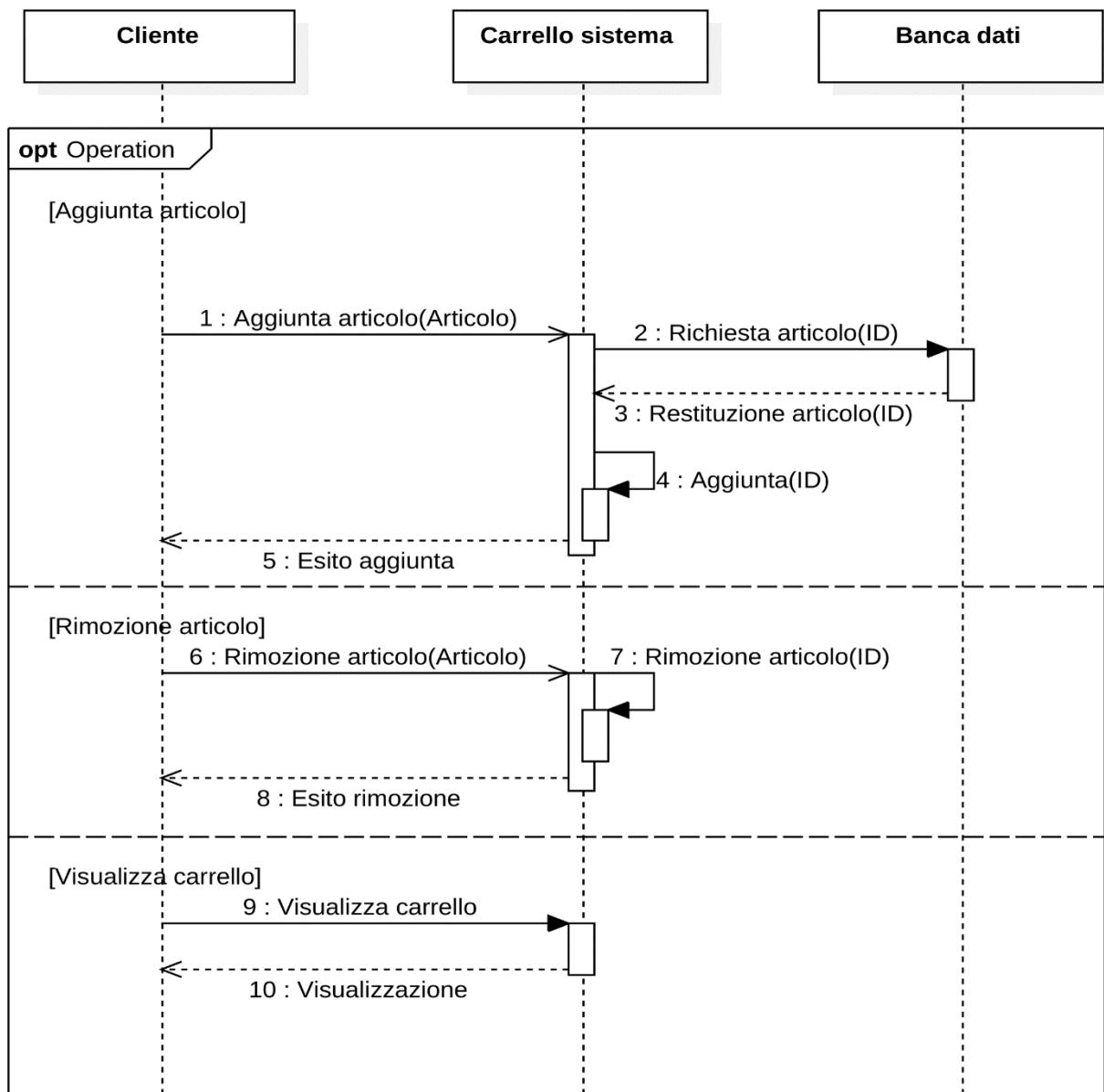


Figura 7: Visualizzazione catalogo. Il sistema riceve le richieste dell'utente e le inoltra alla Banca dati.



*Figura 8: Modifica del carrello. La Banca dati viene contattata solo nell'aggiunta di un prodotto, per ottenerne l'ID.*

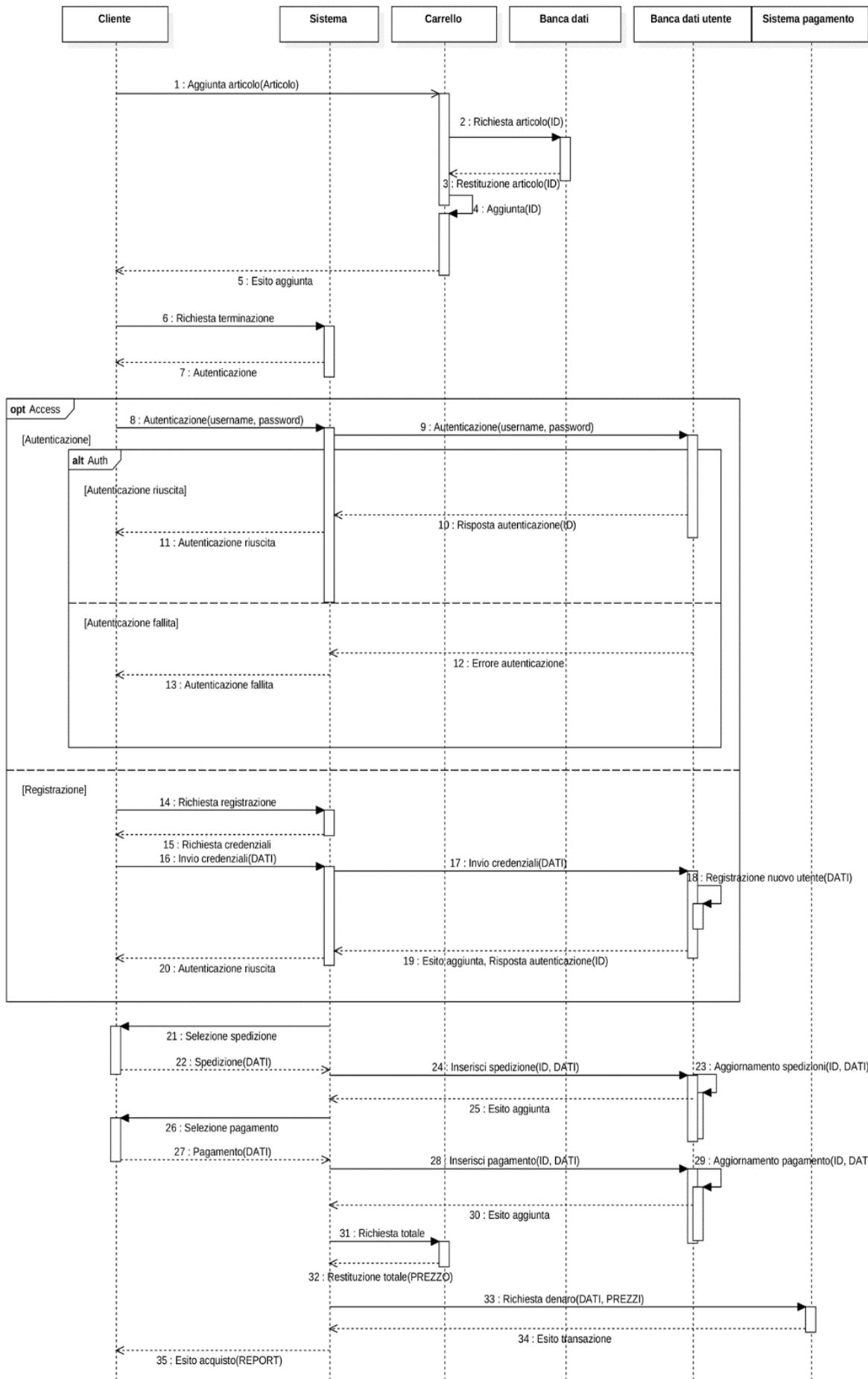


Figura 9: Completamento acquisto. Necessario avere almeno un oggetto nel carrello ed essere autenticati (attraverso accesso o registrazione).

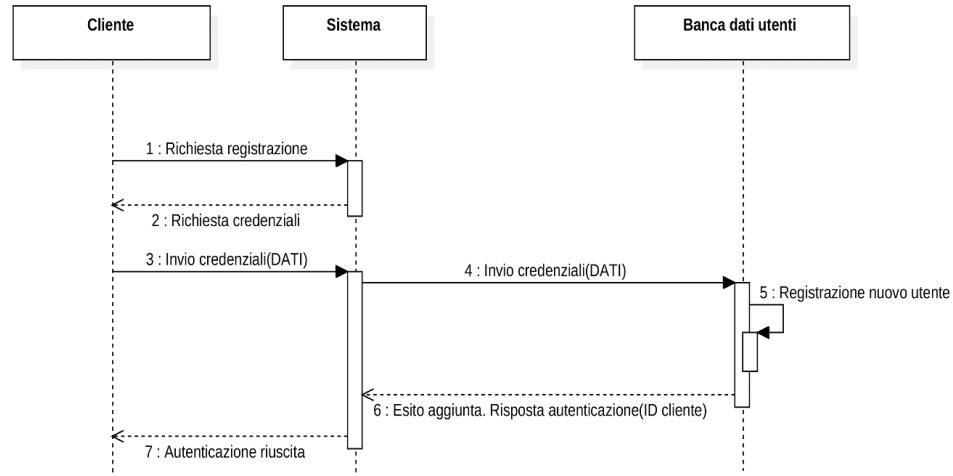


Figura 10: Registrazione nuovo utente.

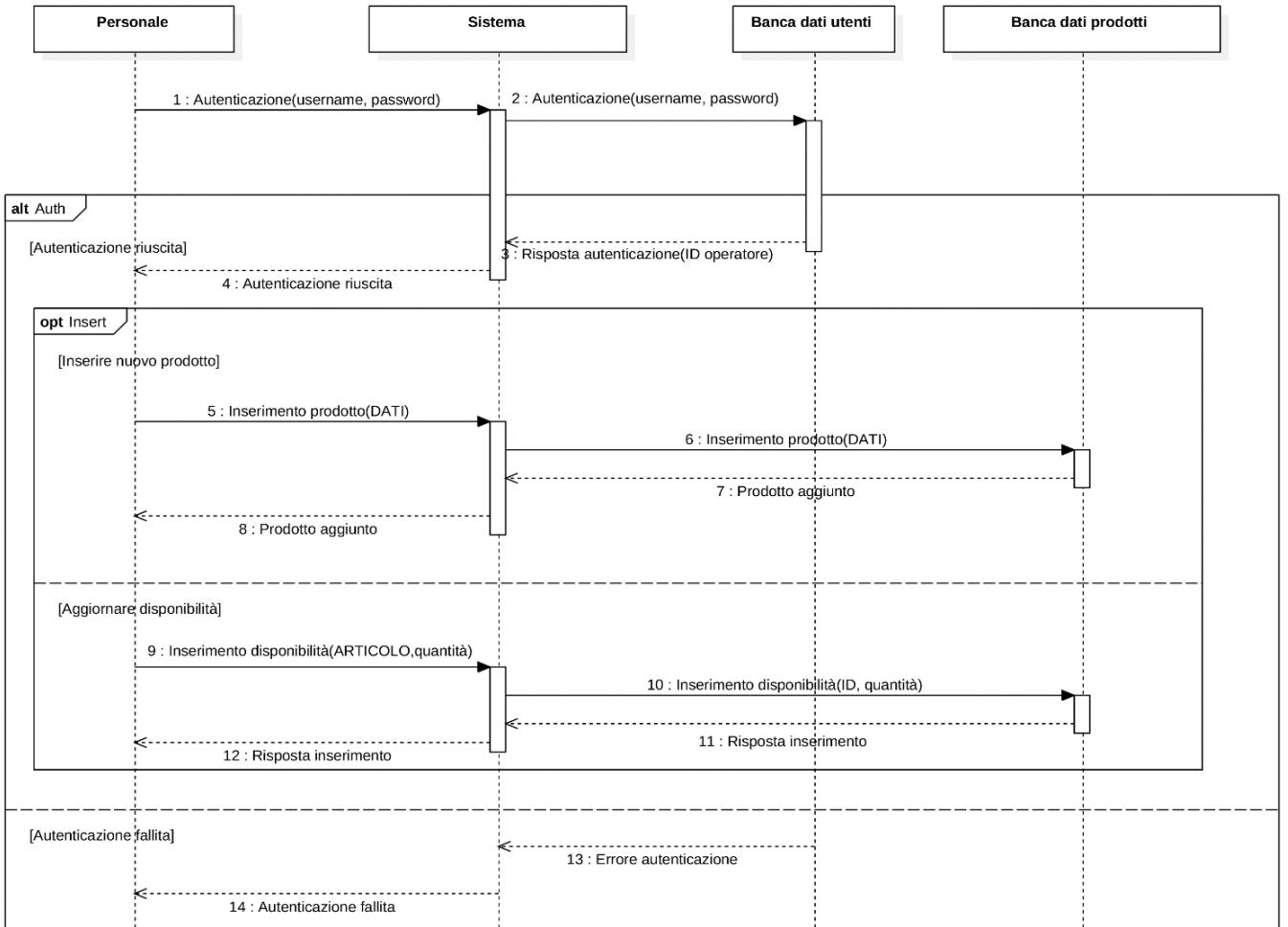


Figura 11: Aggiunta prodotto (Personale). Il Personale può aggiungere prodotti o modificare la disponibilità di quelli presenti.

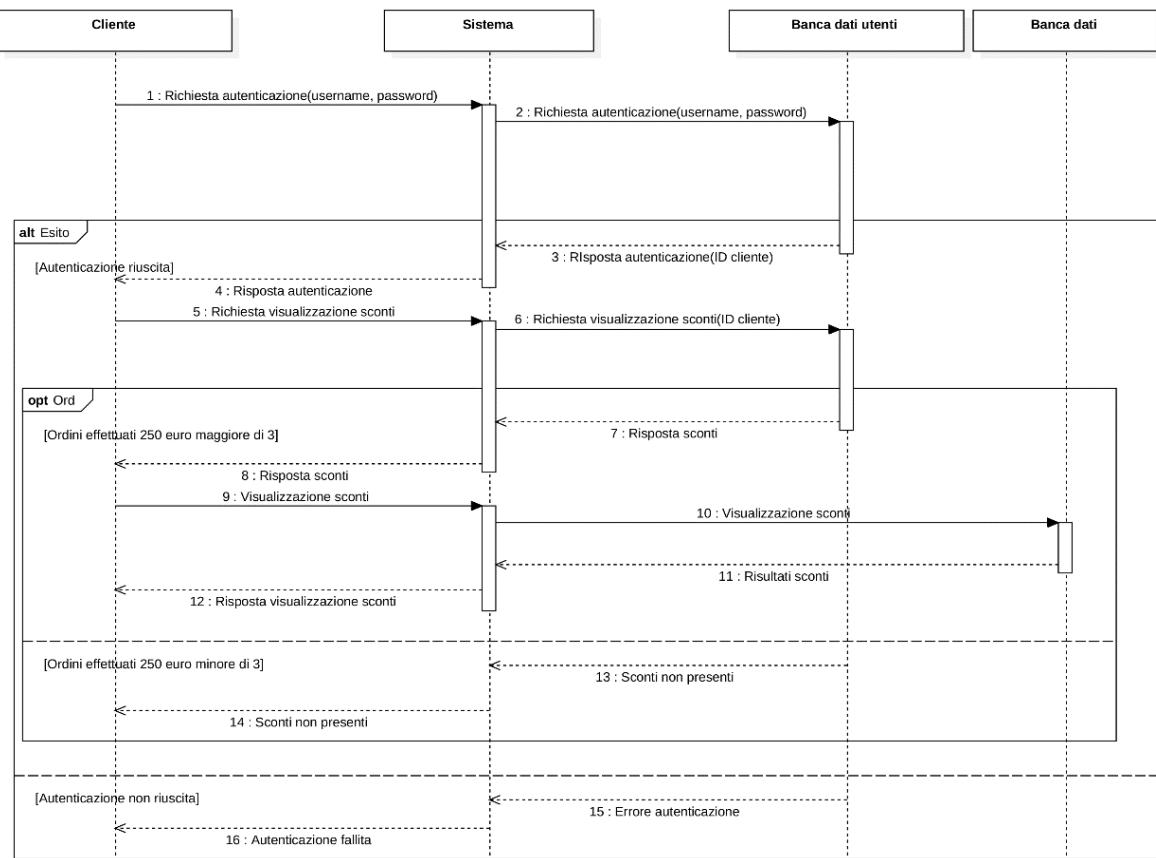


Figura 12: Visualizzazione sconti da parte di un utente autenticato.

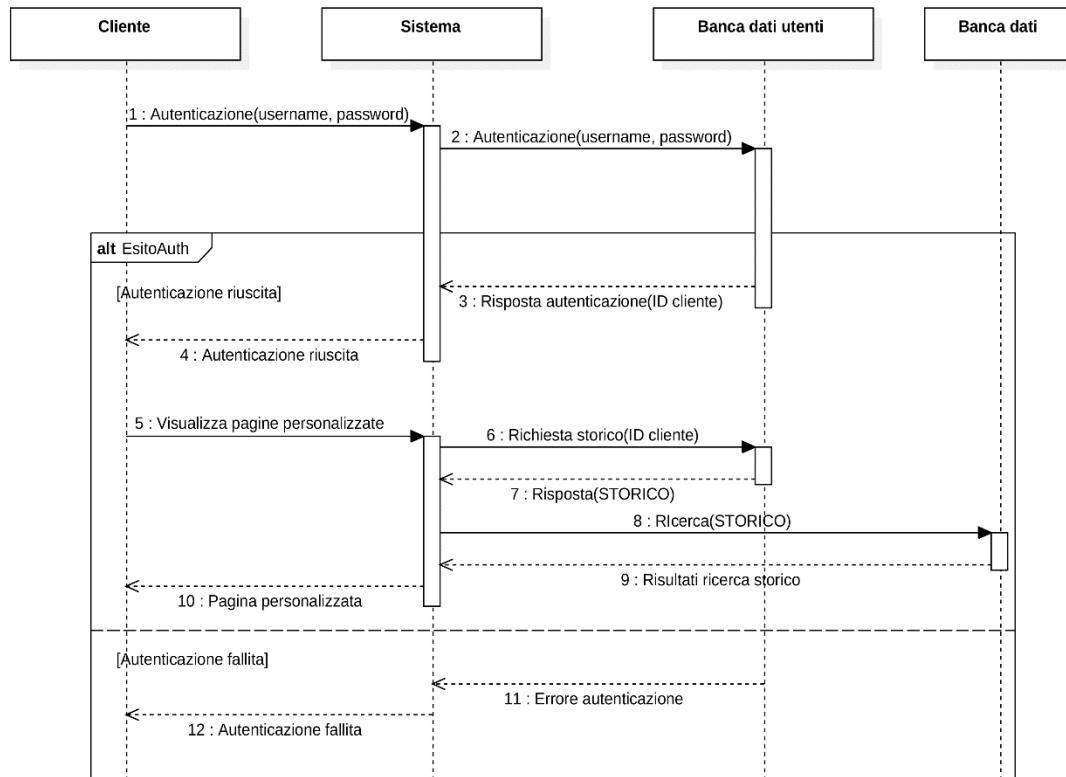


Figura 13: Visualizzazione pagine personalizzate per un utente autenticato. Queste pagine sono modellate secondo lo storico del cliente.

## ACTIVITY DIAGRAM

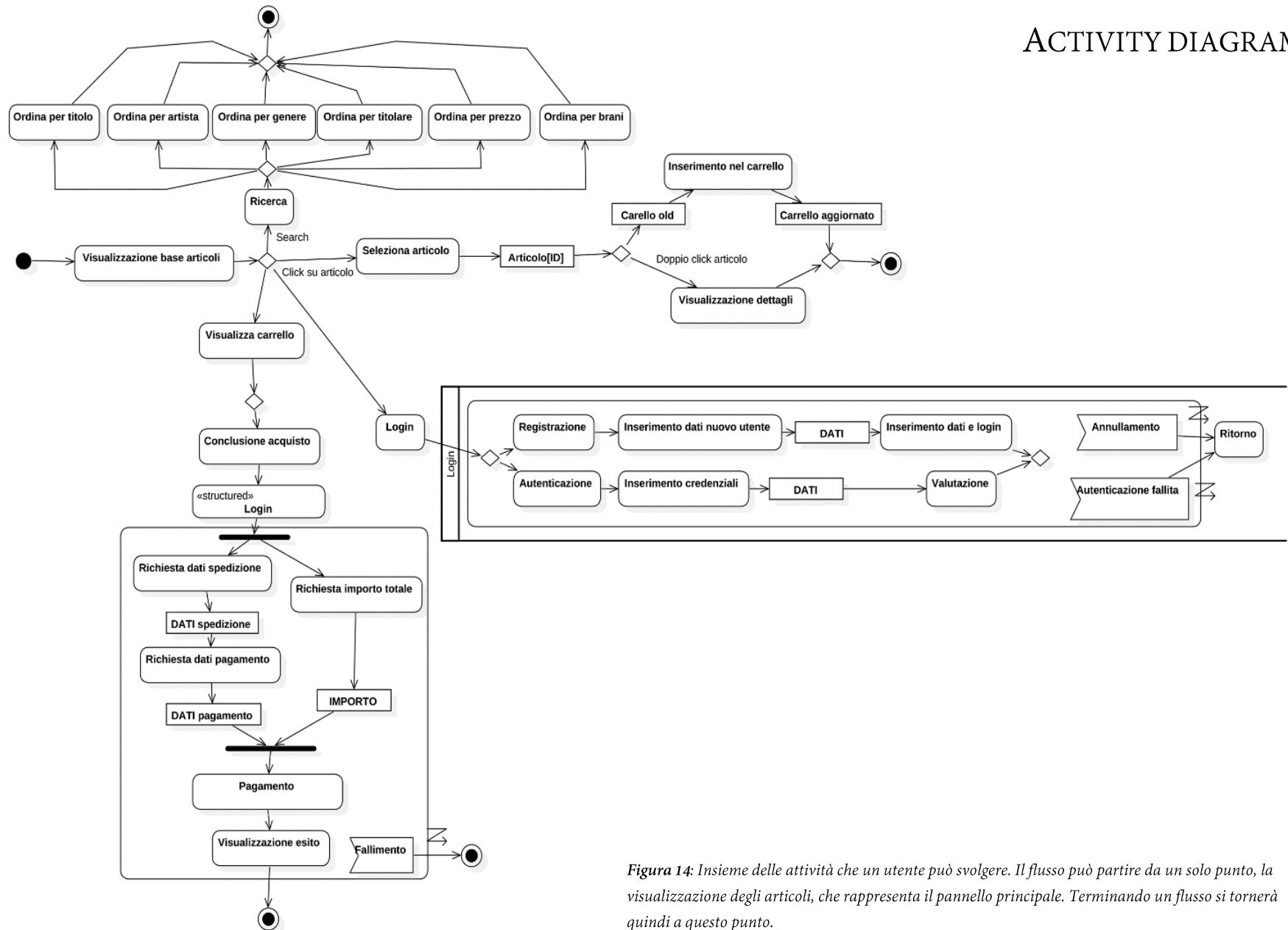


Figura 14: Insieme delle attività che un utente può svolgere. Il flusso può partire da un solo punto, la visualizzazione degli articoli, che rappresenta il pannello principale. Terminando un flusso si tornerà quindi a questo punto.

# DOCUMENTAZIONE TECNICA

Conclusa la fase di implementazione del codice, sono stati rivisti i *sequence diagram*, realizzati per la progettazione del software, ed è stato realizzato il *class diagram*, contenente le classi appartenenti al pacchetto it.RGB.is.Classes.

La differenza che salta subito all'occhio nei nuovi *sequence diagram* è l'introduzione di un Controller<sup>9</sup> e della sezione dedicata alla GUI<sup>10</sup>, infatti durante la fase di implementazione del codice, è sorta la necessità di avere una classe centrale di controllo in grado sia di aggiornare l'interfaccia grafica che operare sulle classi di progettazione in base all'operazione compiuta dal cliente (ad esempio la rimozione di una certa quantità di prodotto dal catalogo dopo che viene inserito nel carrello e di conseguenza il relativo refresh della tabella con le quantità aggiornate). Quindi come si evince dai diagrammi allegati, il Controller svolge una funzione di gestione del programma in base alle azioni del cliente.

Come accennato sopra, è stata introdotta anche la sezione dedicata alla GUI<sup>11</sup> in quanto, essendo parte integrante del software, tramite i *sequence diagram* è possibile osservare come evolve l'interfaccia in base alle richieste dell'utente e quindi avere una panoramica sul comportamento generale.

---

<sup>9</sup> Colorato di verde nei diagrammi

<sup>10</sup> Colorato di rosso nei diagrammi

<sup>11</sup> Si rimanda alla sezione dedicata **Interfaccia grafica (GUI)** per ulteriori approfondimenti

# CLASS DIAGRAM

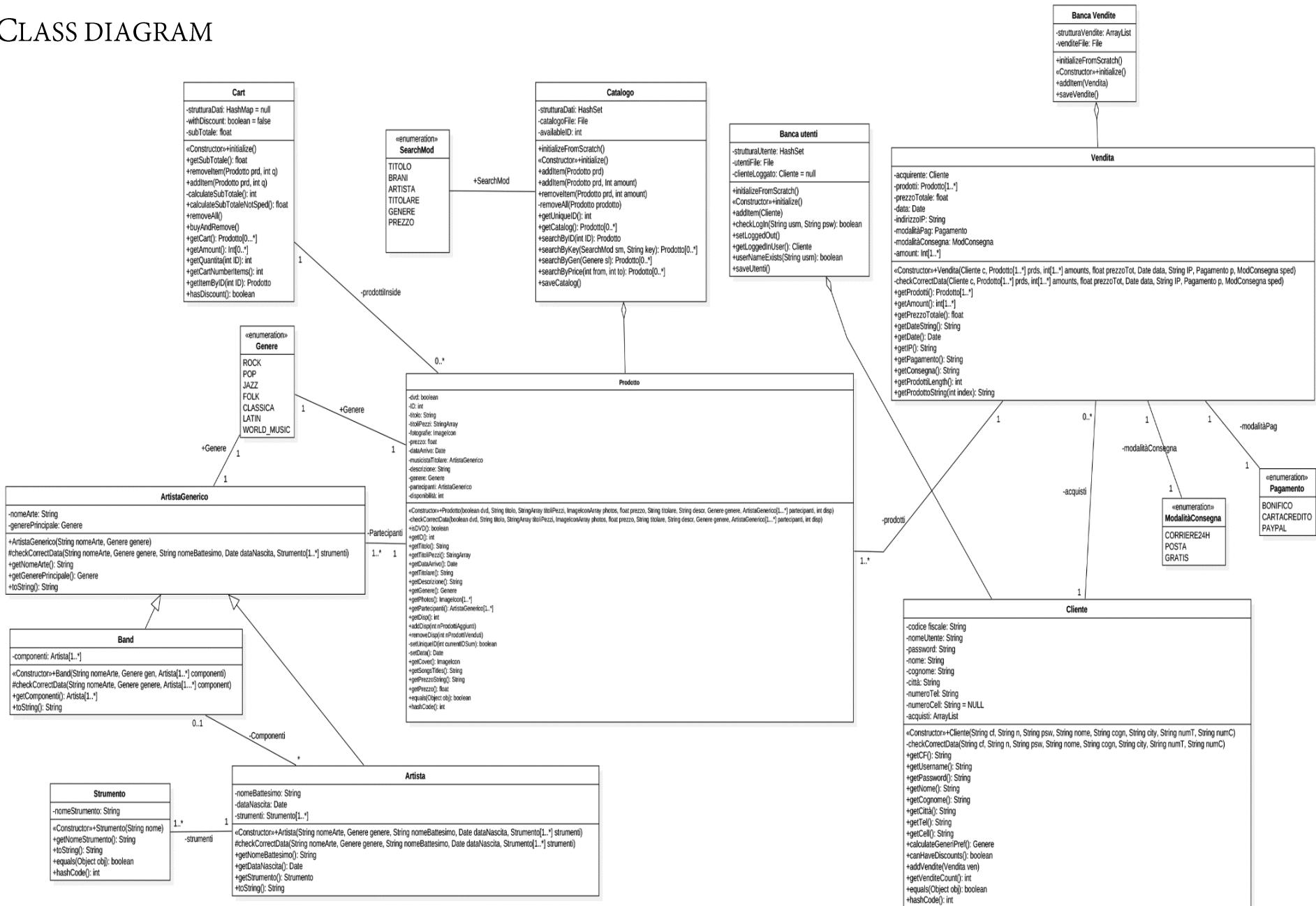


Figura 15: Class diagram delle classi di progettazione

## SEQUENCE DIAGRAM

Nelle prossime pagine vengono allegati i *sequence diagram* realizzati dopo lo sviluppo del codice.

Colore	Ruolo nell'MVC pattern
Verde	Controller ( <i>gestione del cambiamento dell'interfaccia</i> )
Rosso	View ( <i>Interfaccia grafica e modelli delle tabelle</i> )
Blu	Model ( <i>struttura dati</i> )

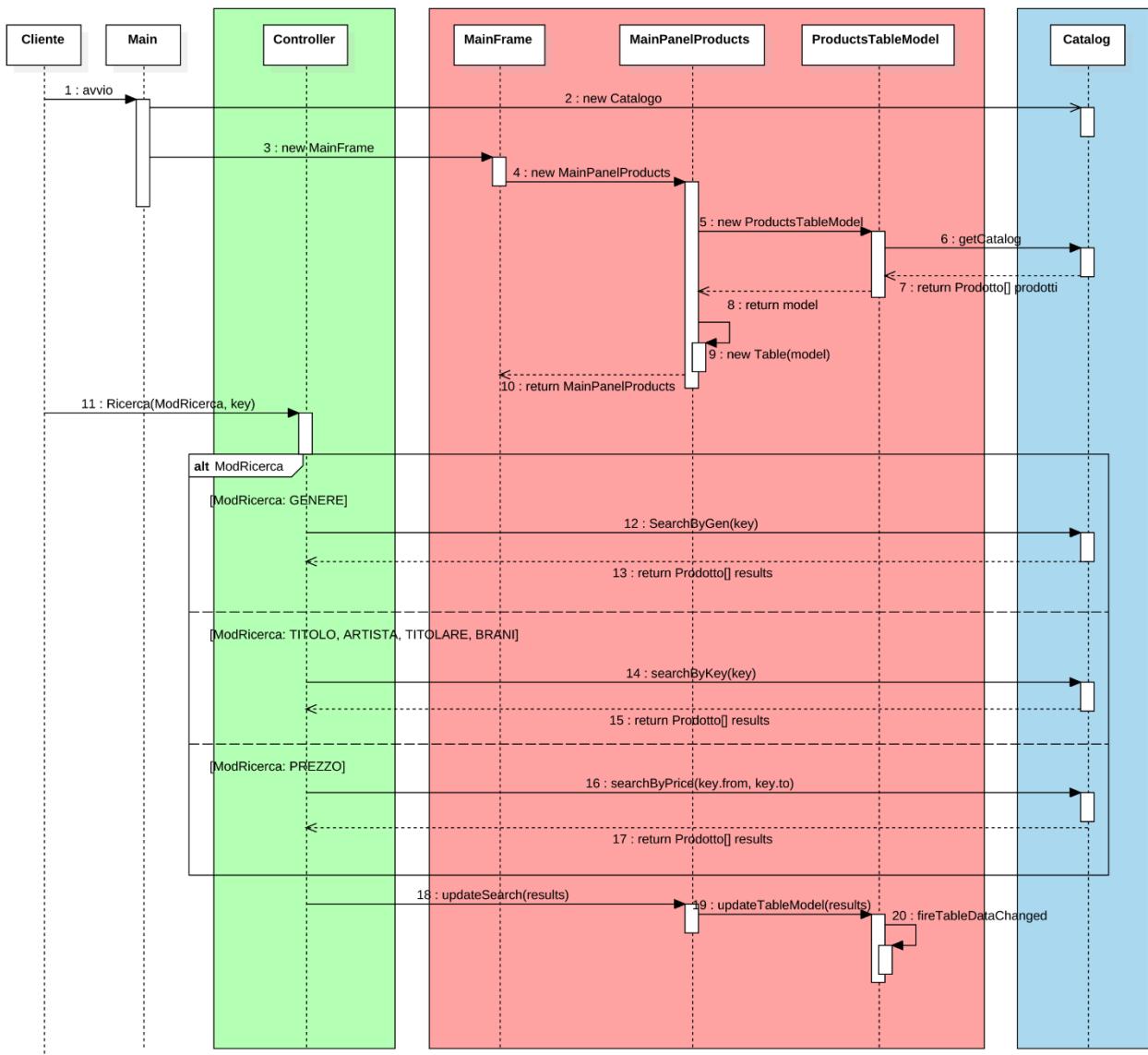


Figura 16: Visualizzazione catalogo

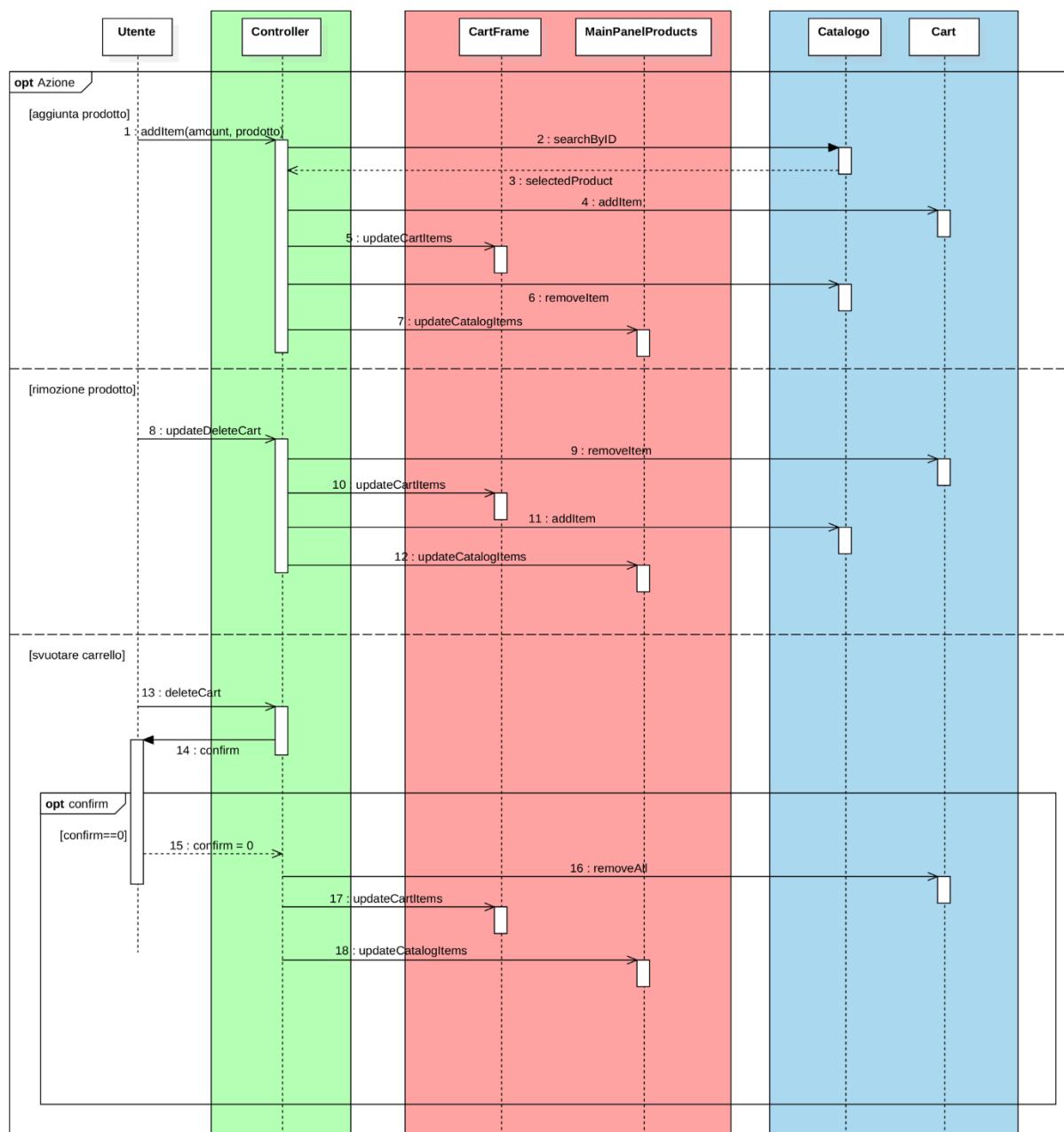


Figura 17: Modifica carrello

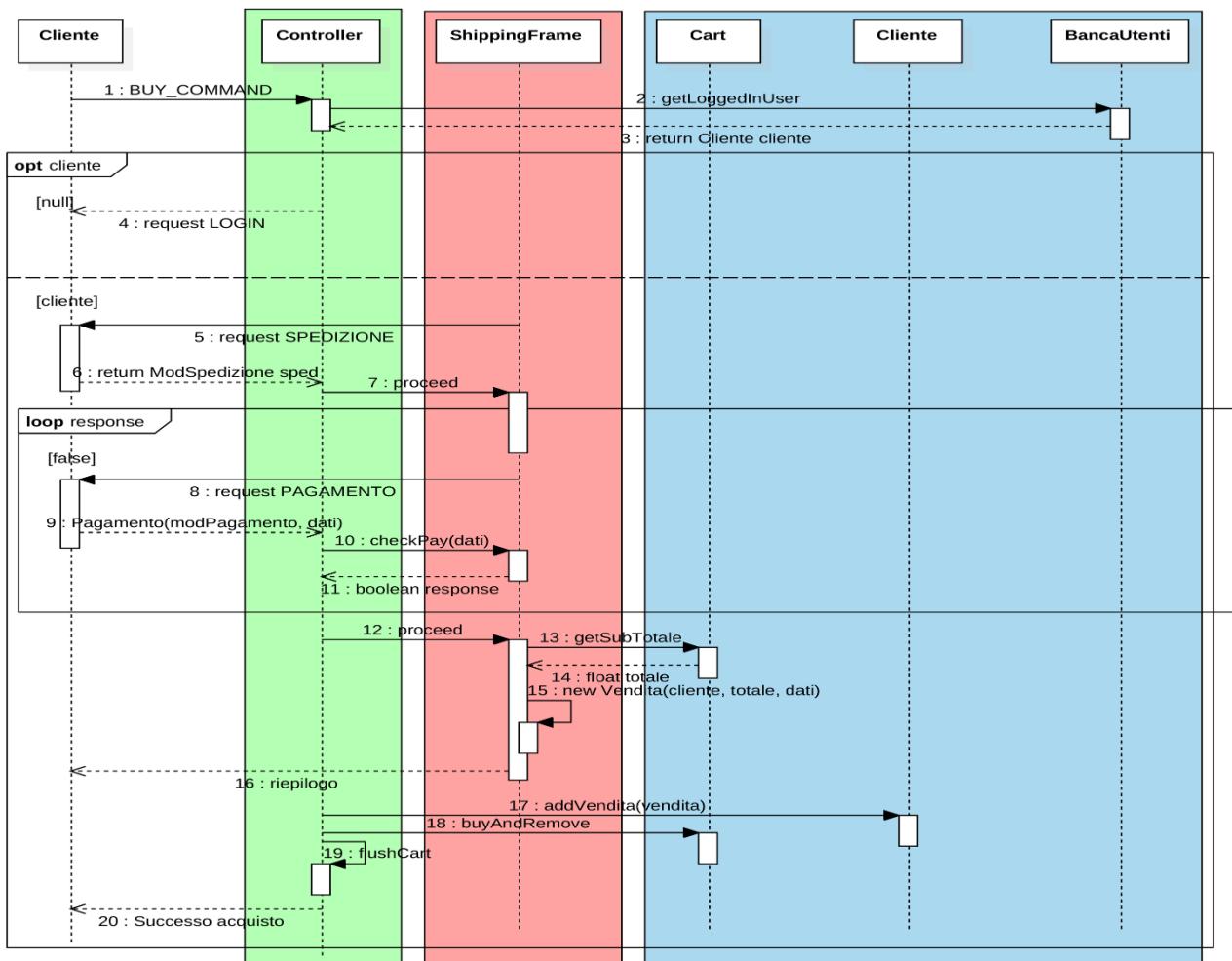


Figura 18: Completamento acquisto

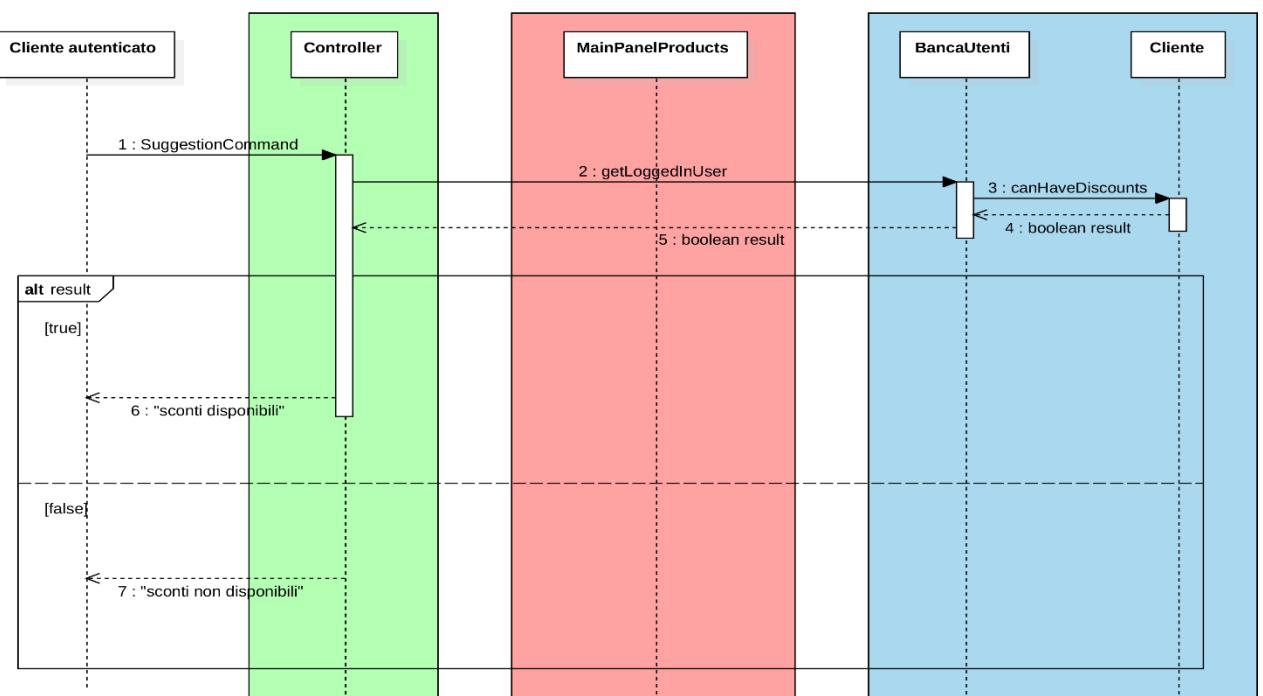


Figura 19: Visualizzazione sconti

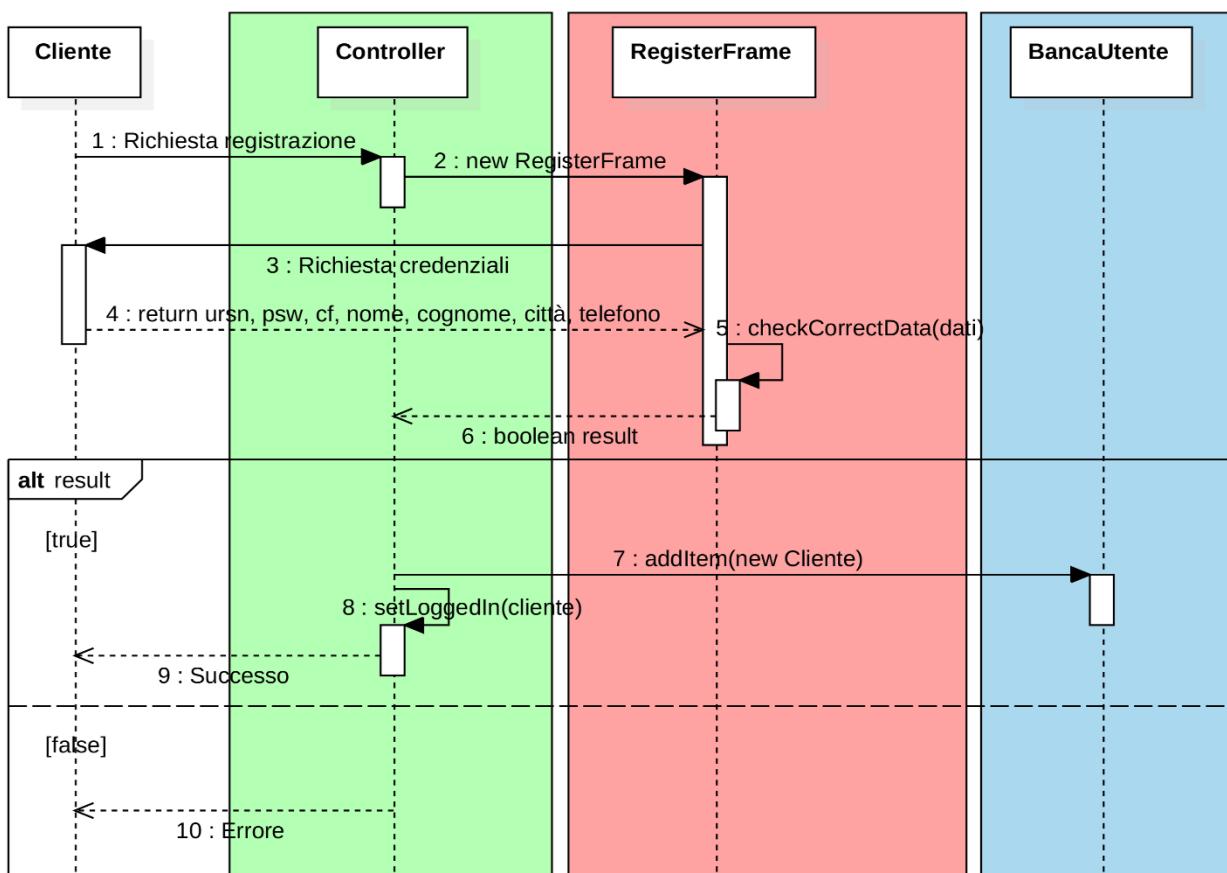


Figura 20: Registrazione nuovo utente

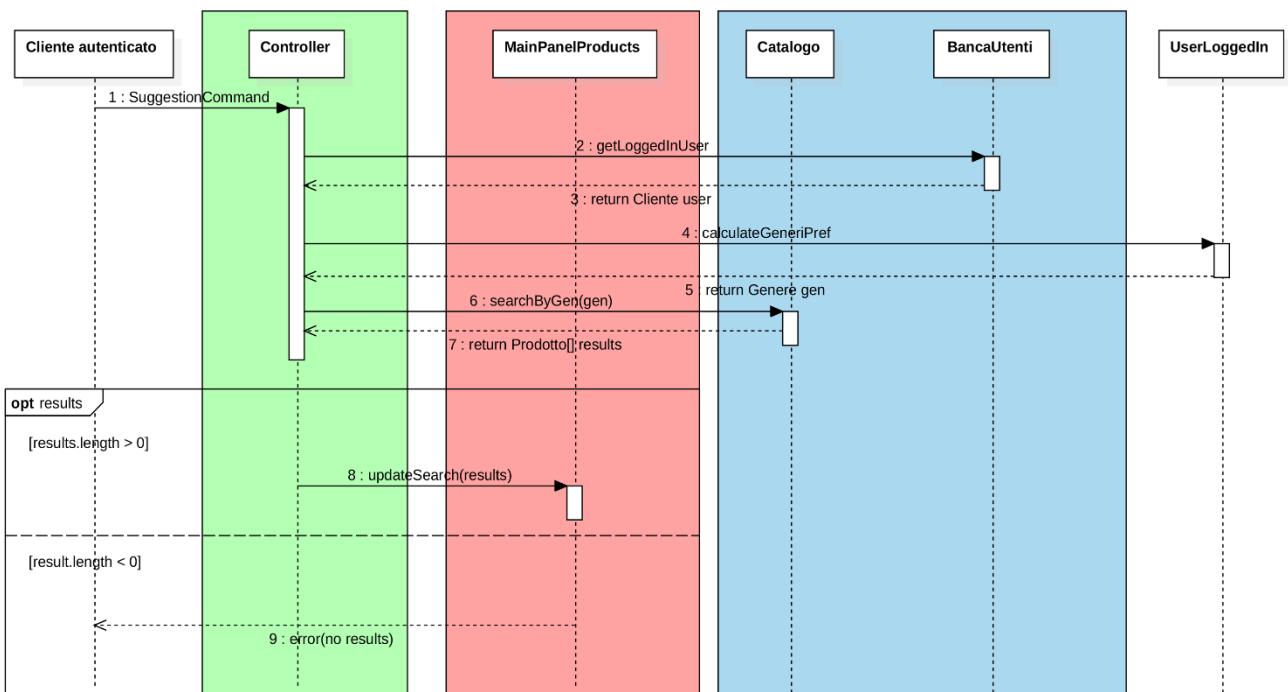


Figura 21: Visualizzazione suggerimenti

## DESIGN PATTERN UTILIZZATI

All'interno del sistema che siamo andati a implementare ci siamo fatti aiutare da alcuni pattern di programmazione che ci potevano permettere di dare una struttura solida e valida al software in generale e alle varie parti costituenti.

### PATTERN ARCHITETTURALI

Il principale pattern architettonico che abbiamo utilizzato è certamente quello del **Model View Controller (MVC)** che “ingloba” interamente il software che abbiamo creato. La classe *Controller* è appunto il fulcro del pattern ed è esattamente, insieme ai vari listener utilizzati, la parte logica che controlla le operazioni fatte sull’interfaccia e le richieste che provengono dall’utente stesso. Non a caso nei *sequence diagram* di “post-produzione” abbiamo cercato di evidenziare il ruolo del *Controller*<sup>12</sup> come l’intermediario tra il sistema e l’utilizzatore, per cui qualunque operazione che richiedesse l’utilizzo delle classi di sistema e non fosse una semplice operazione di visualizzazione (come ad esempio il riordino delle tabelle) passava sempre tramite il *Controller*, facilitando la correzione degli errori, permettendo una gestione del codice più centralizzata e applicando correttamente il pattern MVC. È chiaro che il ruolo dei vari listeners in questo quadro è quello di essere intermediari tra il *Controller* e l’utente, di conseguenza i listeners stessi inoltravano le varie richieste al *Controller*<sup>13</sup> che a sua volta gestiva l’evento correttamente, spesso interagendo con parecchie classi di interfaccia e di sistema.

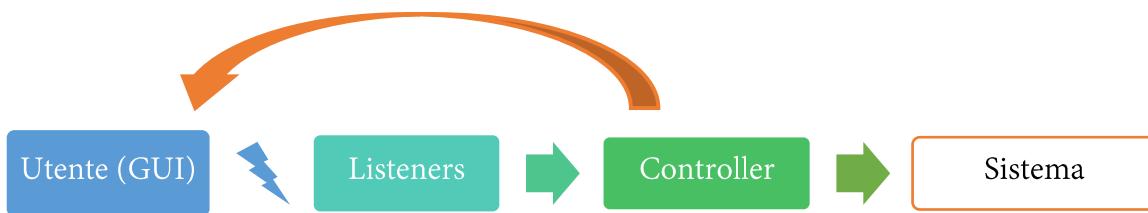


Figura 22: MVC del software implementato

### PATTERN CREAZIONALI

Il principale pattern creazionale che abbiamo implementato è stato il **singleton** anche se in una forma adattata al nostro contesto. Abbiamo cercato di creare delle classi che appunto fossero uniche per tutto il sistema e fossero accessibili in modo univoco. Per questo abbiamo utilizzato il *singleton pattern*. Veniva inizializzata un'unica istanza della classe, salvata nella classe stessa come attributo, e la classe era resa accessibile in modo statico dall'esterno, creando appunto solamente attributi e i metodi statici. Le classi coinvolte sono state Cart,

<sup>12</sup> Colorato sempre in verde nei diagrammi

<sup>13</sup> La maggior parte delle volte, dato che alcune operazioni intrinseche a molte componenti dell’interfaccia erano troppo complesse da inoltrare direttamente al Controller

Catalogo, BancaUtenti, BancaVendite. Una volta inizializzate, queste classi chiamavano il loro costruttore privato, che nella maggior parte dei casi valutava i dati sui files corrispondenti e li salvava negli attributi, ed erano accessibili dall'esterno tramite i metodi statici creati.

## PATTERN STRUTTURALI

All'interno della classe Controller, oltre ad essere visibile l'MVC pattern è possibile intravedere anche il *façade pattern*, il pattern “facciata”. In questo tipo di pattern si cerca di raccogliere in un'unica classe tutta la logica di controllo di altre classi “nascoste”, che se fosse esterna sarebbe complessa e ridondante, in metodi più generici e comprensivi. Durante lo sviluppo ci siamo infatti accorti che, oltre a non aver esplicitato sufficientemente il pattern MVC, succedeva che in parecchie classi (soprattutto nei listener) apparivano sequenze di chiamate a metodi identiche per effettuare operazioni simili se non identiche, rendendo così il codice molto ridondante e rischiando spesso di dimenticare qualche passaggio o qualche chiamata. Per questo grazie al *façade pattern* siamo riusciti a unificare in metodi unici insiemi di operazioni utilizzate spesso e che operavano su varie componenti del sistema e dell'interfaccia, limitando anche il grado possibile di errore. Sono esempi chiari di questo approccio metodi di Controller quali setLoggedIn, setLoggedOut, flushCart, updateDeleteCart, ...

## DESIGN PATTERN

Il pattern comportamentale che abbiamo utilizzato di più è sicuramente quello dell'*observer pattern*. Questo pattern è visibile ovviamente nell'implementazione dell'interfaccia grafica e nella programmazione ad eventi che è insita nella struttura stessa del codice. Questo pattern prevede un'insieme di classi *observer* che si mettano in attesa di alcuni particolari eventi che possono essere scatenati dall'uso dell'interfaccia da parte dell'utente; una volta intercettato uno degli eventi gestiti, l'*observer* (o più propriamente in Java, il *listener*) aveva il compito di gestire l'evento o indipendente dagli altri (in caso di eventi particolari) oppure inoltrandolo a chi di dovere, nel nostro caso il Controller. Per i dettagli a riguardo si rimanda alla sezione successiva riguardante l'interfaccia grafica, tuttavia, intanto, possiamo descrivere sommariamente come abbiamo strutturato il tutto:

- Anzitutto abbiamo fatto in modo di creare Listener<sup>14</sup> separati per ogni componente dell'interfaccia, facendo in modo che estendessero o implementassero tutti i tipi di listener necessari per la componente (listener per il mouse, per la tastiera, e così via)
- Abbiamo molto utilizzato il metodo setActionCommand() creando degli appositi comandi comuni (ad esempio nella classe ActionCommands), soprattutto per i bottoni, che nei listener fossero classificabili e gestibili in modo facile e diretto.

---

<sup>14</sup> Per coerenza da questo momento in poi consideriamo il termine observer come sinonimo di listener, dato il taglio tecnico della descrizione.

Un altro design pattern che abbiamo usato è il template pattern. Questo pattern è visibile nel metodo `toString()` di `ArtistaGenerico`: `toString()` nella superclasse è stato lasciato astratto in modo che solamente nelle sottoclassi si potesse inserire un'implementazione concreta, essendo la stampa a video di questi oggetti molto differente e da “particolareggiare” a seconda della classe considerata, dei suoi metodi e dei suoi attributi.

# INTERFACCIA GRAFICA (GUI)

Particolare cura è stata data alla GUI (*Graphical User Interface*), in quanto riteniamo che sia fondamentale per una UX (*User Experience*) funzionale ma allo stesso tempo gradevole nell'utilizzo. La GUI implementata è molto semplice, ma permette di interagire in maniera familiare proprio come ci si aspetterebbe, ad esempio con il clic del tasto destro del mouse sul prodotto si possono avere delle funzioni rapide a portata di mano.

Di seguito viene riportata una descrizione più dettagliata.

## STRUTTURA FINESTRA PRINCIPALE

La grafica del software sviluppato, all'interno del mainFrameContainer, si divide in 4 sezioni principali:

- La barra del menu in alto realizzata con il componente JMenuBar
- Il pannello principale caratterizzato da un layout GridLayout(1,1) contenente la tabella prodotti realizzata con il componente JTable
- La sidebar caratterizzata da un layout BorderLayout e suddivisa in due parti (nord e sud) contenente pulsanti funzione realizzati con i componenti JButton
- L'area di ricerca e dei pulsanti dedicati all'utente realizzata con un layout GridLayout(1,2) e varie componenti inserite nei due FlowLayout (a seguire sarà approfondito)

## BARRA DEL MENU

Realizzata con il componente JMenuBar, include 3 menu (oggetti JMenu) che racchiudono la maggior parte delle funzioni presenti nelle altre sezioni dell'interfaccia grafica descritte sopra.

In particolare ogni menu include una serie di voci (oggetti JMenuItem) che sono riassunte di seguito:

- **Account:** log-in/log-out e Dettagli utente
- **Acquisto:** Visualizza carrello, Svuota carrello, Completa acquisto, Visualizza suggerimenti, I miei sconti
- **Aiuto:** Informazioni (*che contiene informazioni sulla versione e il copyright del software*)

Da notare che non tutti i bottoni dei vari menu sono disponibili all'avvio del programma, poiché richiedono delle condizioni specifiche che verranno espresse a breve.

## PANNELLO CENTRALE

È il pannello che contiene la tabella dei prodotti realizzata con il componente JTable, in particolare la tabella si presenta suddivisa in 8 colonne sulle quali è possibile cliccare per l'ordinamento dei prodotti presenti. Tramite il relativo listener (MainTableProductsListener) al singolo click del mouse la riga viene colorata per indicare il prodotto selezionato. È inoltre possibile visualizzare un

menu a comparsa premendo il tasto destro del mouse sul relativo prodotto, oppure tramite doppio click entrare direttamente nelle informazioni dettagliate.

## SIDE BAR

Come detto in precedenza è suddivisa in 2 parti, la sidebarNord e la sidebarSud, entrambe presentano un layout GridLayout(2,1) e contengono ciascuna 2 buttoni (JButton).

- **SidebarNord:** contiene i pulsanti per il log-in/log-out e per la visualizzazione del carrello
- **SidebarSud:** contiene i pulsanti per l'aggiunta di un prodotto al carrello e per la visualizzazione delle informazioni dettagliate.

L'obiettivo che si vuole raggiungere con la sidebar è quello di rendere accessibili i pulsanti più importanti, ma allo stesso tempo distinguerli dai pulsanti presenti nella parte inferiore della finestra, inoltre tramite le icone utilizzate è possibile intuire a colpo d'occhio, appena avviato il programma, quali sono le funzioni disponibili per il cliente.

## AREA DI RICERCA E DEI PULSANTI DEDICATI

In quest'area sono presenti due pannelli ( JPanel ) con layout FlowLayout dedicati all'utente loggato e alla ricerca di prodotti. In particolare:

- **Ricerca avanzata:** include vari componenti utili alla ricerca per determinate categorie. Tramite il selettore realizzato con il componente JComboBox infatti è possibile selezionare la categoria di ricerca, in base a tale scelta si attivano determinati componenti, il JSpinner (per il prezzo), il JComboBox (per il genere) o un JTextField per tutti gli altri tipi di ricerca. Fatta la selezione è possibile premere con il mouse il pulsante di ricerca sulla destra per ottenere i risultati, mentre per cancellare i parametri di ricerca viene reso disponibile un bottone per il ripristino allo stato iniziale della tabella.
- **Area cliente:** presenta due pulsanti ( JButton ), uno per la visualizzazione degli articoli suggeriti e uno per la visualizzazione degli sconti disponibili.

## FINESTRE SUPPLEMENTARI

Oltre alla finestra principale sono state implementate ulteriori finestre per fornire spazi dedicati a determinate funzioni. Di seguito vengono elencate e descritte tutte le finestre presenti<sup>15</sup>:

- Finestre dedicate all'utente
  - **Login:** Questa finestra (LoginFrame) permette al cliente di accedere alla sua "area riservata", consentendo così di fare degli acquisti, visualizzare sconti e degli articoli personalizzati. Dal punto di vista grafico abbiamo utilizzato un pannello principale con BorderLayout, nella sezione NORD abbiamo inserito il pannello coi campi dell'autenticazione (2 JTextField per l'username e per la password), nella sezione

---

<sup>15</sup> Tutte le finestre considerate estendono per comodità JDialog e sono modali alla finestra principale (quindi la finestra inferiore non è utilizzabile mentre un'altra è aperta)

CENTER due JButton per accedere e per annullare e nella sezione SOUTH un JButton per la registrazione. LoginFrameListener controlla questi bottoni e in caso o chiude la finestra, o apre un frame di registrazione oppure inoltra al Controller la richiesta di accesso.

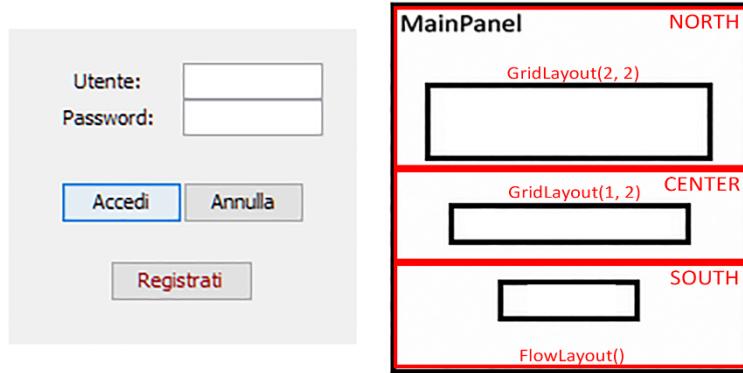


Figura 23: Schema disposizione LoginFrame

- **Registrazione:** Questa finestra (RegisterFrame), accessibile dal Login, permette ad un nuovo cliente di registrarsi. Al suo interno principalmente c'è un solo pannello, con un GridLayout, che contiene su una colonna il JLabel della descrizione del campo, e sull'altra il JTextField corrispondente. RegisterFrameListener invece permette di annullare la registrazione oppure di confermarla inoltrando al Controller la richiesta per il controllo della correttezza dei dati inseriti.

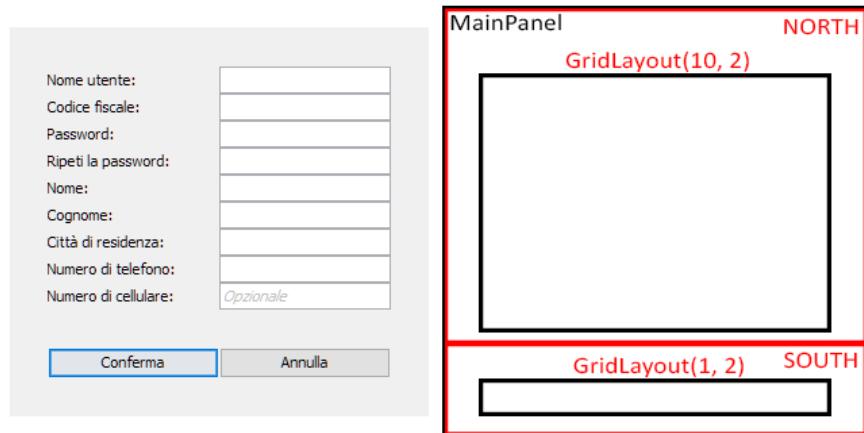


Figura 24: Schema disposizione RegisterFrame

- **Dettagli utente:** Questa finestra (UsrDetailsFrame) contiene un pannello principale con BorderLayout. Nella regione NORD è inserito un pannello contenente il nome dell'utente e un'immagine, su un GridLayout; nella regione CENTER, tramite un GridLayout sono stati inseriti i dati personali dell'utente, nella regione SOUTH invece un JScrollPane (per permettere lo scorrimento verticale del riquadro), contenente un pannello che al suo interno mantiene le informazioni sugli acquisti fatti. Per rendere gradevole la vista di questo ultimo pannello, abbiamo dovuto fare un compromesso sulla sua altezza: utilizzando un GridLayout un acquisto con tanti

prodotti avrebbe fatto in modo di allargare tutte le righe in modo sproporzionato, di conseguenza abbiamo utilizzato un FlowLayout verticale stimando per ogni acquisto fatto 500 px di altezza, in modo da renderlo più uniforme.

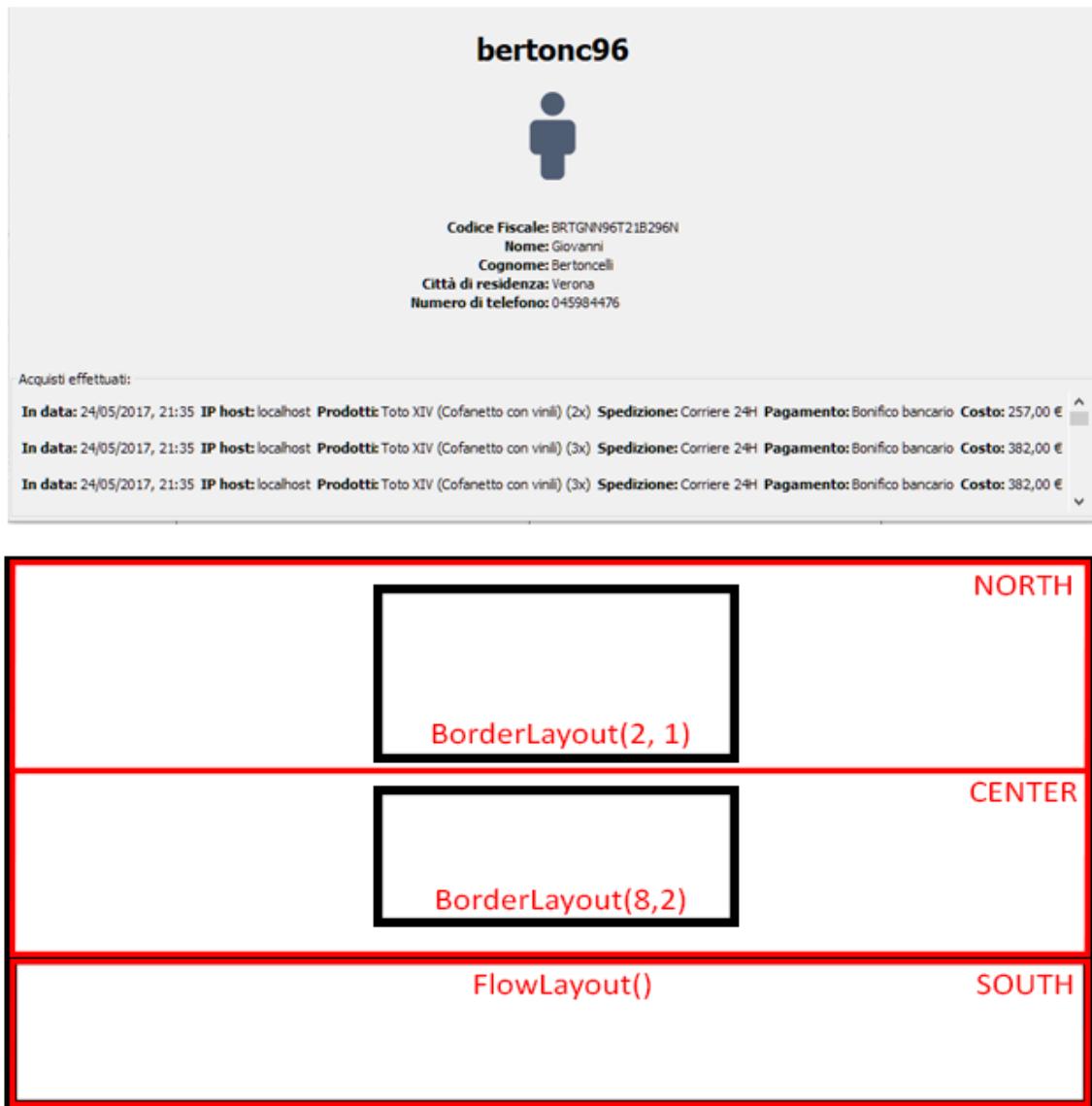


Figura 25: Schema disposizione UsrDetailsFrame

- Finestre per i prodotti e il carrello
  - **Carrello:** Il CartFrame è una JDialog molto simile all'interfaccia principale. Nella parte centrale, all'interno di un JScrollPane, è stata inserita una JTable, con un modello JTableModel annesso, nella parte inferiore invece un FlowLayout che contiene vari pulsanti di azione, mentre sul fianco un riepilogo del carrello corrente. Nel JScrollPane si alternano la tabella stessa, quando il carrello ha degli oggetti, e un'immagine apposita, quando il carrello è vuoto.

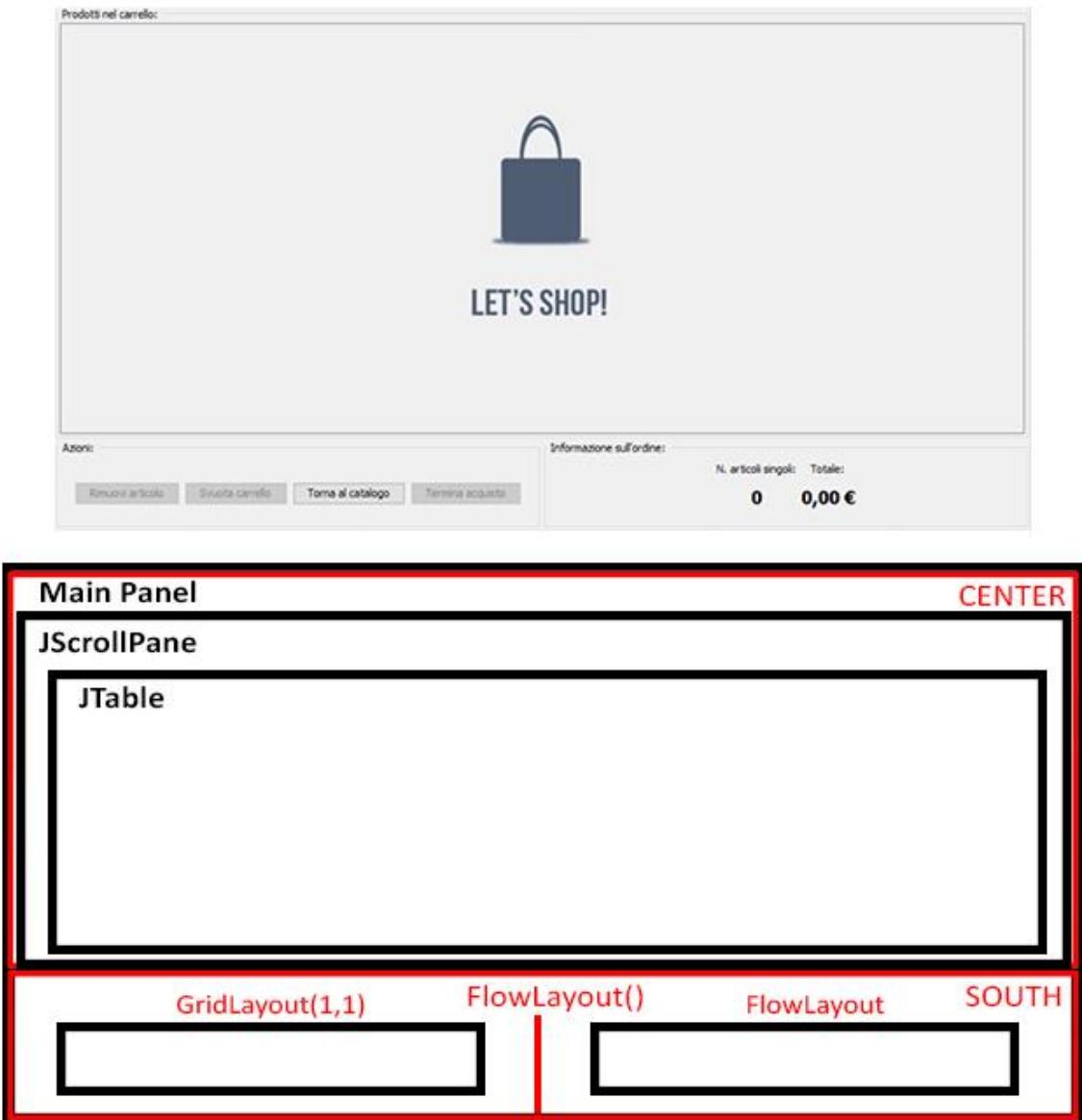


Figura 26: Schema disposizione CartFrame

- **Dettagli prodotto:** PrdDetailsFrame contiene tutti i dettagli di un prodotto selezionato. Il suo costruttore contiene l'ID dell'oggetto selezionato che verrà recuperato dal catalogo all'inizializzazione dell'interfaccia. Al suo interno vengono mostrati: le immagini di copertina del prodotto, il titolo, tutti i nomi dei brani, gli artisti partecipanti, la data di aggiunta, la descrizione, il genere e il prezzo.

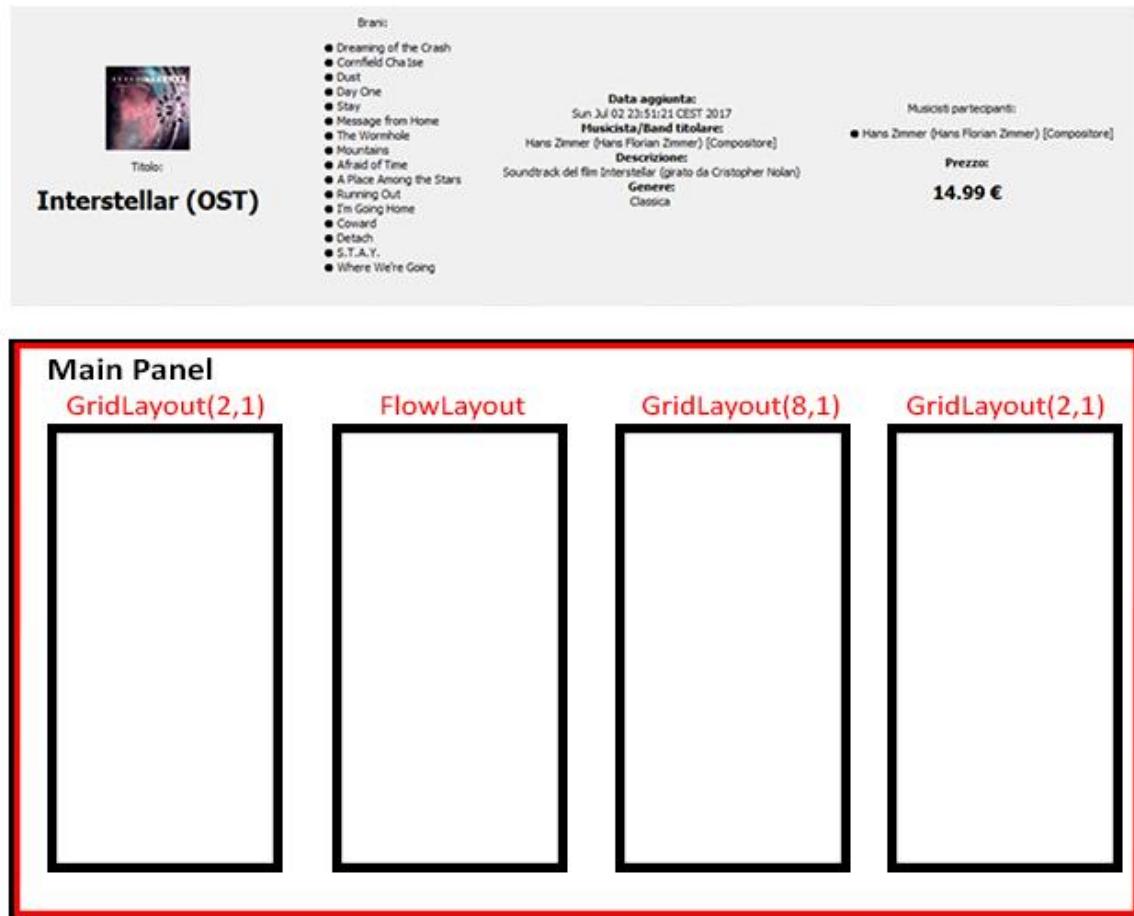


Figura 27: Schema disposizione PrdDetailsFrame

- **Selezione quantità:** In modo simile alla componente descritta prima, anche per questo JDialog il costruttore riceve un Prodotto, assieme ad una quantità massima selezionabile per il JSpinner. Quest'ultima componente permette di selezionare una quantità da 1 alla quantità disponibile nel catalogo.



Figura 28: Schema disposizione AmountPiker

- Finestre per il completamento dell'acquisto

In questo caso il JDialog è unico, tuttavia tramite appositi comandi si può procedere o retrocedere su viste differenti (descritte sotto) e inserire differenti dati, questo viene fatto impostando il pannello principale con differenti panelli.

- Spedizione: selezione della spedizione, opzioni disponibili: gratis (se abilitato), corriere e posta

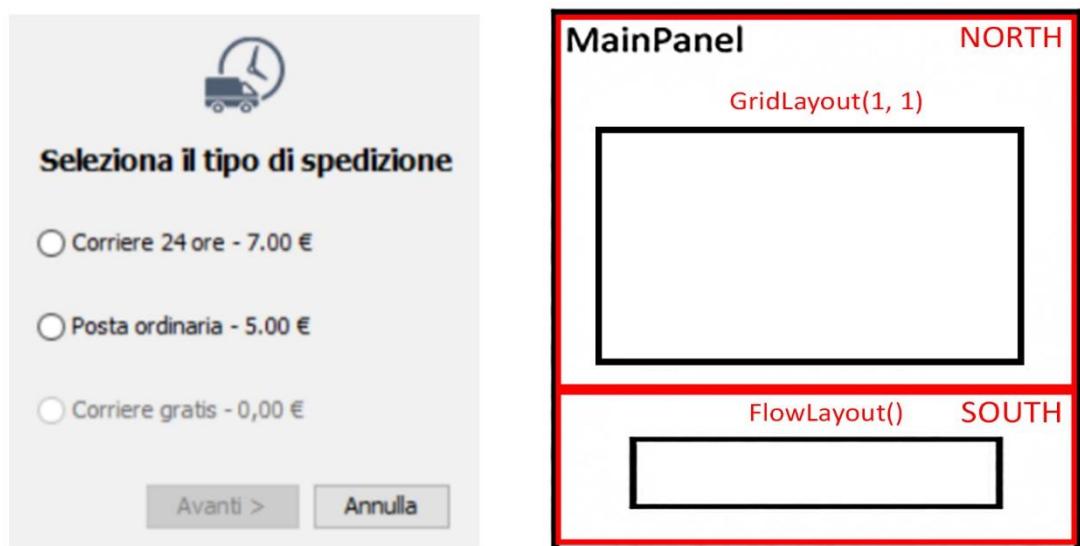


Figura 29: ShippingFrame - Spedizione

- Pagamento: selezione del pagamento, opzioni disponibili: tramite bonifico bancario, paypal oppure tramite carta di credito

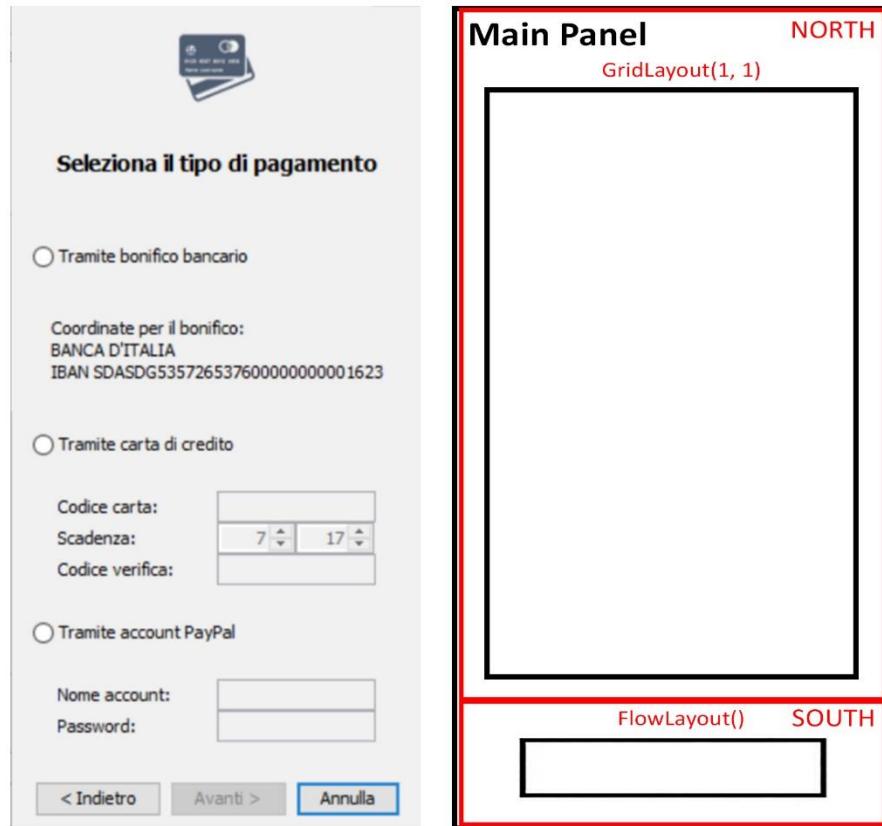


Figura 30: Schema disposizione ShippingFrame - Pagamento

- **Riepilogo ordine:** visualizza un riassunto sul riepilogo e sulle varie modalità, in caso si applica qui lo sconto



Figura 31: Schema disposizione ShippingFrame - Riepilogo

- **Conferma ordine effettuato:** conferma dell'ordine a video

Inoltre sono stati inseriti JOptionPane per la conferma di determinate operazioni come l'uscita dal programma, per avvisi da parte del sistema in caso di errori (ad esempio nella fase di login o nella fase di degli sconti e dei suggerimenti per l'utente registrato).

## LA GUI DINAMICA

Come detto in precedenza, molta cura è stata data all'interfaccia grafica, sia per rendere il programma esteticamente gradevole sia per renderlo funzionale e di semplice utilizzo, dunque per rendere più intuitiva l'esperienza d'uso, sono stati implementati dei piccoli accorgimenti, soprattutto per quanto riguarda le icone e l'attivazione di determinati pulsanti funzione, nonché dell'aggiornamento in tempo reale del catalogo e della lista del carrello.

In particolare, tramite la classe Controller e le relative classi delle varie componenti, sono stati inseriti dei metodi, come setLoggedIn, che modificano l'interfaccia in base alle operazioni svolte dall'utente. Per esempio, con il metodo appena citato, oltre a settare l'utente come autenticato ed a rendere disponibili determinati button (Visualizza suggerimenti, Dettagli utenti, ecc.) sia nella barra del menu in alto che nell'area in fondo, cambia anche l'icona dei pulsanti di log-in che diventano pulsanti di log-out.

Stessa cosa è visibile quando si aggiunge un prodotto al carrello, nella *sidebar*, l'icona del carrello vuoto viene modificata per mostrare un carrello pieno e vengono attivati i pulsanti di svuotamento del carrello e di termina acquisto (sia nella barra del menu che all'interno del carrello stesso). Sempre riguardo al carrello, nella finestra dedicata alla visualizzazione degli articoli selezionati, in caso di rimozione la grafica si aggiorna immediatamente senza dover chiudere e riaprire la finestra.

Questi sono solo alcuni esempi della GUI dinamica che è stata implementata, l'intento è quello di fornire un'interfaccia coerente in tutte le fasi dell'utilizzo del software e un'esperienza intuitiva, ecco perché, ad esempio, molti pulsanti all'inizio sono visibili ma non cliccabili.

Un ulteriore step, per rendere la UX più familiare, è stato fatto tramite l'introduzione del riconoscimento dei comandi da tastiera, implementato tramite la classe KeyboardListener, in particolare per il tasto ESC che permette di uscire dalle finestre supplementari in maniera più rapida

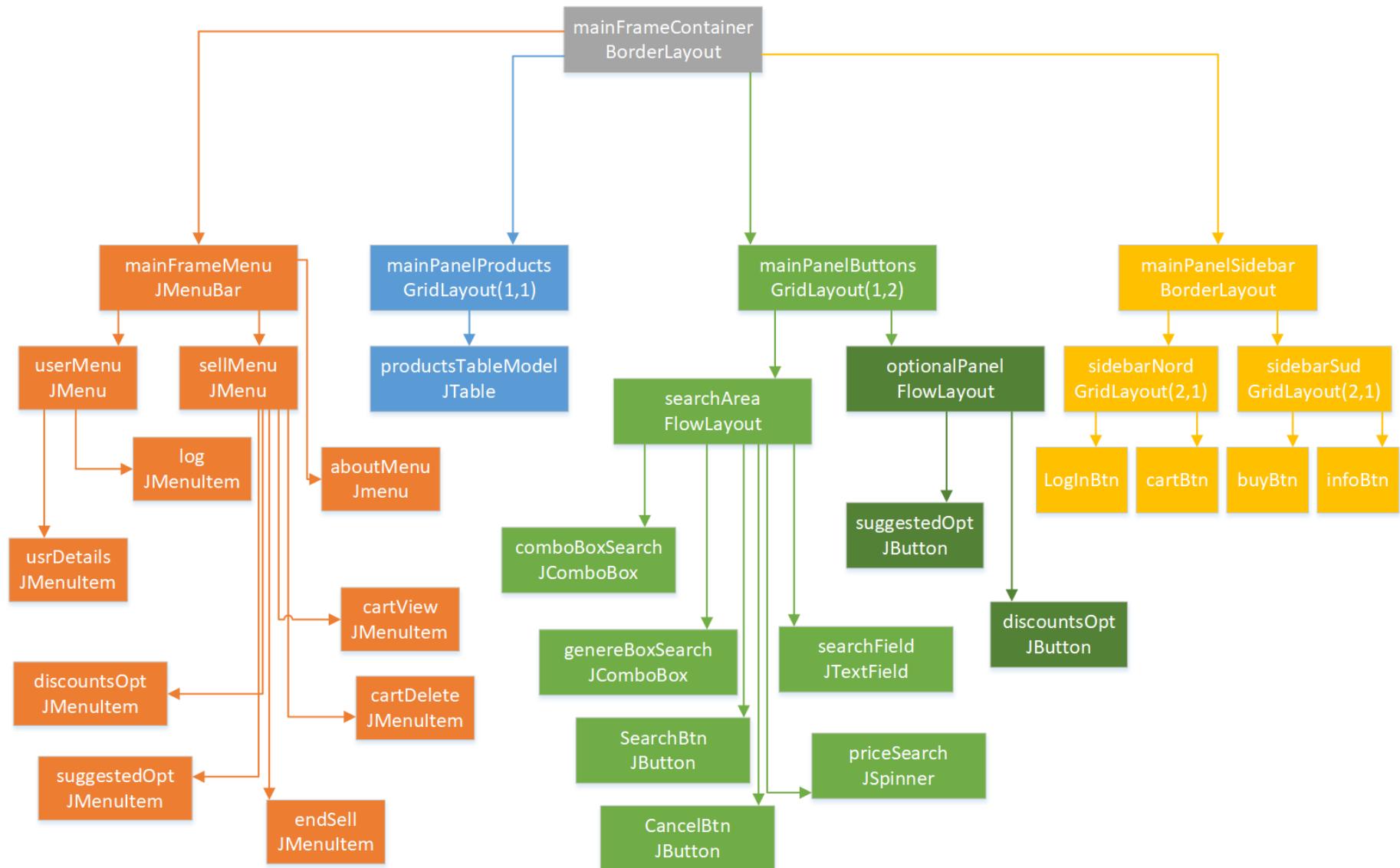


Figura 32: Gerarchia di contenimento dell'interfaccia grafica principale

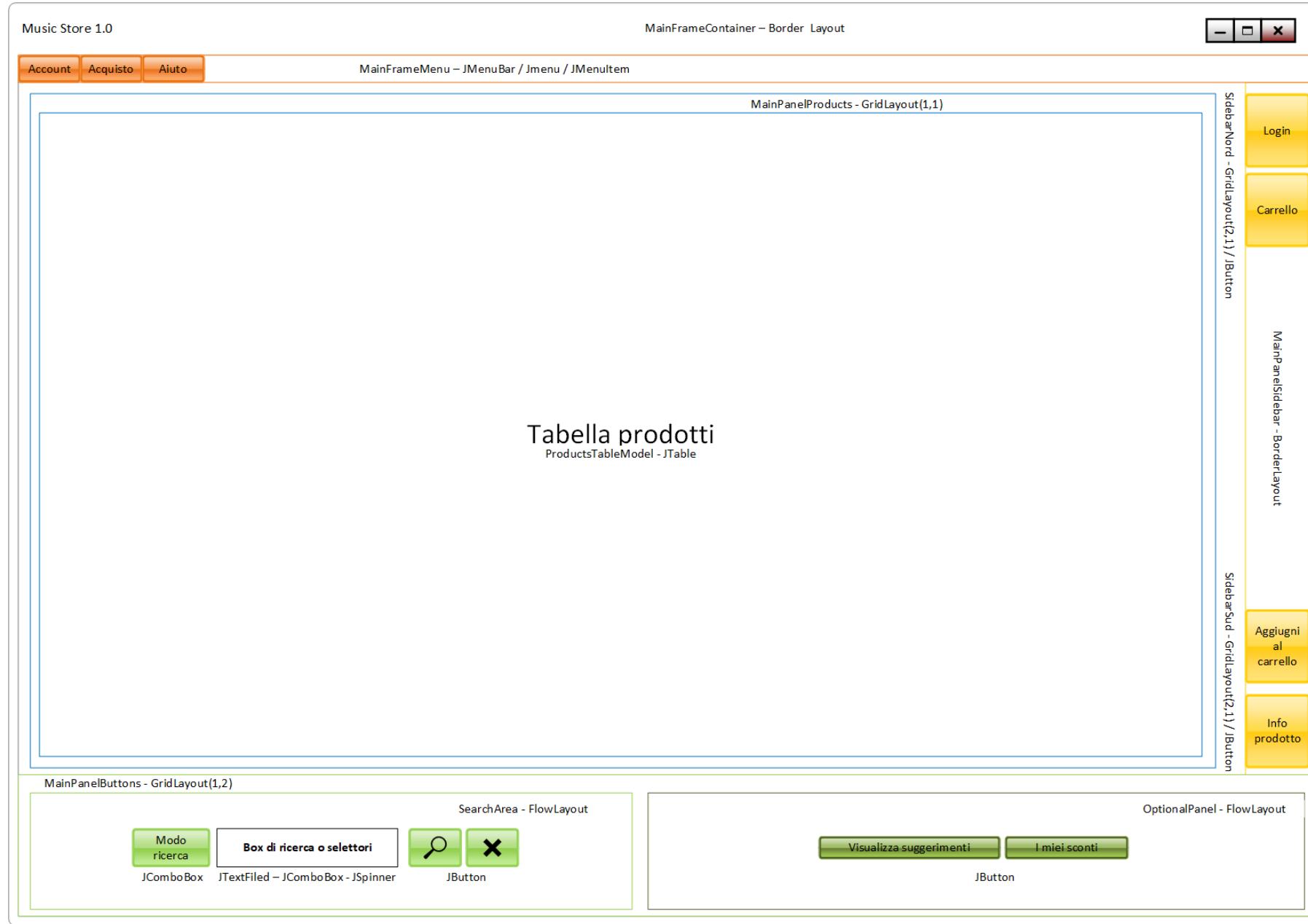


Figura 33: Schema disposizione GUI (principale) – i colori rimandano alla figura 20

# TESTING DEL SOFTWARE

Prima di addentrarci nella fase di testing, il nostro progetto ha dovuto subire parecchie modifiche. Il problema principale era dato dal fatto che nel codice scritto fino a quel momento non erano state gestite situazioni di errore e/o parametri scorretti passati ai costruttori o ai metodi degli oggetti, se non il minimo possibile dal Controller e dalle classi di interfaccia, quindi non era possibile né effettuare dei test corretti sulle classi né effettivamente, cosa ancor più grave, in fase di esecuzione gestire errori che avrebbero potuto portare a risultati imprevedibili. Per questo motivo sono anche stati inserite eccezioni apposite a seconda della classe e del tipo di errore.

Per questo abbiamo dovuto eseguire le seguenti modifiche:

- Abbiamo anzitutto migrato tutto il codice possibile presente nella classe Controller nelle classi corrispondenti, facendo in modo che fossero le classi a controllare i parametri durante l'inizializzazione delle stesse. In particolare questo è avvenuto per la classe Cliente: prima di passare i parametri al costruttore il Controller verificava la loro correttezza e in caso mostrava un messaggio di errore. Dopo la migrazione del codice questo non avveniva più ma, in modo più elegante e corretto, i controlli venivano fatti all'interno del costruttore del Cliente stesso (chiamando il metodo checkCorrectData()), il quale nel caso in cui alcuni parametri non fossero stati corretti lanciava un'eccezione con un messaggio preciso di errore, che a sua volta il Controller mostrava tramite un JOptionPane.showMessageDialog(). Un simile approccio è stato utilizzato anche per le altre classi.
- Abbiamo creato un package apposito per contenere le classi di eccezione. Tra queste distinguiamo:
  - **CriticalException**: classe concreta, estesa da altre classi simili, che viene utilizzata per gli errori più gravi e/o improbabili, come ad esempio errori di IO sui file di dati del programma (all'apertura e alla chiusura di esso), nullPointerException molto improbabili e altre situazioni simili che difficilmente si potevano verificare. CriticalException ha anche il compito di riportare i dettagli (lo stackTrace) di questi errori su file (music\_store\_file/errors/) e terminare il programma.
  - **Eccezioni unchecked** con prefisso "Light" che vengono lanciate da errori più lievi e meno gravosi per il sistema, non visibili all'utente esterno.
  - **Eccezioni checked** per gli errori gravi che estendono CriticalException oppure Exception (in caso non si voglia terminare il programma), che vanno ovviamente gestite dalla logica esterna.
- Nelle classi per ogni metodo abbiamo fatto una serie di controlli comuni:
  - Nei costruttori abbiamo inserito controlli per verificare che i parametri non fossero vuoti o nulli e in seguito che fossero coerenti con la logica della classe.
  - Nei metodi, generalmente, abbiamo verificato che non ci fossero parametri con valori non ammessi (ad esempio quantità negative o nulle e parametri vuoti o nulli)

Dopo questa fase di refactoring e di sistemazione del codice abbiamo proceduto alla fase di testing, distinguendo la fase di Unit Testing e di Integration Testing<sup>16</sup>.

Elementi comuni tra le due modalità sono stati:

1. Utilizzo del framework di testing JUnit
2. **Code coverage:** abbiamo cercato, soprattutto con plug-in appositi come EclEmma, di fare dei test che permettessero di “ricoprire” tutto il codice delle classi in modo da testarlo senza ridondanza e testando la classe nella sua interezza
3. Test univoci e facilmente identificabili: i nomi dati ai test cercavano di ricalcare il più possibile la situazione che veniva verificata. Per questo abbiamo cercato di identificare per ogni classe le situazioni e le circostanze che portavano la classe a lanciare eccezioni oppure ad avere comportamenti differenti rispetto a quelli standard, e separare questi in differenti test.
4. Per ogni test fatto i dati utilizzati per lo stesso venivano inizializzati da zero in modo da consegnare al test solo “dati puliti” e non magari modificati durante l'esecuzione del test nel suo complesso.
5. Per ogni classe testata abbiamo fatto in modo di testare il comportamento *standard* della classe e quello anomalo generato da input incoerenti o scorretti.

A questo proposito abbiamo creato una classe chiamata TestData che permettesse di recuperare facilmente dati validi da utilizzare durante il testing per renderlo realistico e coerente con il resto del progetto.

## UNIT TESTING

In questa fase abbiamo testato le classi singolarmente senza considerare le loro interazioni con altre classi esterne. Abbiamo verificato il comportamento corretto dei costruttori e dei metodi “iniettando” tramite i test parametri o valori non ammessi, ed inoltre abbiamo verificato che con i giusti parametri le classi si comportassero diversamente dai casi standard ma pur sempre in modo corretto.

Per questi motivi sono state incluse in questa fase solamente le classi “singole” che cioè al loro interno non avessero interazioni particolari con altre classi differenti da loro (tramite metodi o attributi).

Passiamo quindi a dare una descrizione dei vari test effettuati e delle loro eventuali particolarità.

### *ARTISTA TEST*

Come in altri casi abbiamo creato test case che ricevessero elementi nulli ed elementi vuoti (array vuoti). In testArtistaSemplice abbiamo controllato il funzionamento della classe Artista con valori standard, per il genere, per il nome d'arte e per il toString corrispondente.

---

<sup>16</sup>Nella fase di unit e di integration testing abbiamo considerato solamente le classi di progetto, cioè ereditate dalla progettazione e contenute nel package *it.RGB.is.Classes*

Abbiamo quindi analizzato, tramite `testArtistaConSecondoNome`, anche il caso più “anomalo” in cui l’artista avesse un nome d’arte differente da quelli di battesimo. I vari test sono stati estesi anche alle sottoclassi di `ArtistaGenerico`.

## *CLIENTE TEST*

Questo è uno degli Unit Test più “complessi” dal momento che i parametri passati al costruttore devono rispettare parecchie regolarità. Oltre ai test per gli argomenti vuoti o nulli e per il comportamento standard abbiamo quindi creato test che verificassero il controllo ad esempio su un codice fiscale con un numero di caratteri invalido, oppure con un formato non corretto<sup>17</sup>; abbiamo identificato il “caso anomalo” in cui il cliente si è registrato con un numero di cellulare<sup>18</sup>. Inoltre abbiamo cercato di testare tutti i metodi della classe, tra cui il metodo per verificare se l’utente potesse avere sconti, il metodo che restituisce il genere preferito del Cliente, e il metodo che aggiunge una Vendita al cliente stesso (anche con parametri scorretti). Infine, cosa fatta anche per altre classi simili, abbiamo verificato la correttezza dei metodi `equals` e `hashCode`, prima in modo separato poi in correlazione.

## *PRODOTTO TEST, VENDITA TEST, STRUMENTO TEST*

Anche in questi casi abbiamo analizzato con un test il funzionamento standard delle classi, i parametri nulli e vuoti per i vari metodi e quindi l’`equals` e l’`hashCode`.

## *TEST DATA*

Questa classe è stata utilizzata per generare dati fintizi ma realistici da inserire nei vari test in modo da avere una classe che fornisca velocemente dati che altrimenti sarebbe molto costoso e ridondante inizializzare a mano in ogni classe. Sono stati inseriti oggetti di tipo Cliente, Prodotto, Band, Artista, Vendita e inoltre la classe inizializza Cart, Catalogo, BancaUtenti e BancaVendite.

# INTEGRATION TESTING

Questa seconda fase è difficile da distinguere dalla prima, tuttavia include tutti i test riguardanti classi generiche che interagiscono con diverse altre classi e fungono per lo più da interfacce tra di loro. Abbiamo cercato anche qui di valutare la robustezza dei costruttori e inoltre di testare come queste interagissero con le classi precedenti. I test sui costruttori tuttavia sono stati molto limitati dato che i parametri passati erano pochissimi e la maggior parte delle classi prelevava i dati da file.

## *BANCA UTENTI TEST*

In questo test abbiamo testato principalmente il log-in e il log-out dell’utente. Abbiamo verificato che un utente inesistente non può loggarsi e fallisce anche un log-in con parametri

<sup>17</sup> Dato che non sono presenti tutti i dati necessari, la verifica sul codice fiscale si limita a controllare, tramite espressione regolare, che la stringa inserita sia conforme alfabeticamente e numericamente ad un codice fiscale italiano generico

<sup>18</sup> Non obbligatorio alla registrazione

nulli, e viceversa. Abbiamo quindi testato il fatto che una volta fatto il log-out il clienteLoggato è pari a null.

## *CARTTEST*

In questo test abbiamo dato particolare importanza ad argomenti scorretti passati ai metodi. Abbiamo infatti verificato che rimozione o aggiunta di quantità negative, di quantità superiori alla disponibilità e/o alla quantità presente nel carrello generassero le eccezioni apposite. Inoltre sono stati controllati tutti i metodi in modo che ad esempio aggiunte al carrello di elementi già presenti si sommassero, che la rimozione di una certa quantità funzionasse e che il carrello fosse vuoto dopo il suo completo svuotamento.

## *CATALOGOTEST*

Anche in questo test abbiamo fatto controlli molto simili a quelli del CartTest. Abbiamo verificato la corretta aggiunta e rimozione, con e senza quantità, dei prodotti, il corretto funzionamento dei vari metodi per la ricerca e in più il corretto funzionamento del metodo getCatalog che appunto restituiva un array rappresentativo del catalogo per i modelli delle tabelle.

## ALTRE MODALITÀ DI TEST

L'interfaccia creata per il progetto è stata testata in maniera manuale, abbiamo cercato di verificare il suo corretto funzionamento producendo diverse sequenze di operazioni che avrebbero potuto produrre malfunzionamenti.

Soprattutto nella fase finale del lavoro abbiamo proceduto con un'ispezione del codice che cercasse di verificare il funzionamento del software senza utilizzare altro software