Documentazione

Progetto di  
Ingegneria del Software

Fabiola Fabretti, VR437439  
Andrea Toffaletti, VR437439  
  
  
Luglio 2020

Indice

[Requisiti ed interazioni utente-sistema 4](#_Toc45235034)

[Casi d’uso e relativi diagrammi di sequenza 4](#_Toc45235035)

[Note generali sui casi d’uso 4](#_Toc45235036)

[Casi dell’User 5](#_Toc45235037)

[Caso 1: Composizione del carrello 5](#_Toc45235038)

[Caso 2: Conferma dell’ordine 6](#_Toc45235039)

[Caso 3: Visualizzare e modificare informazioni nel profilo 7](#_Toc45235040)

[Casi del Worker 8](#_Toc45235041)

[Caso 1: Modificare informazioni sui prodotti 8](#_Toc45235042)

[Caso 2: Visualizzare e modificare stato degli ordini 9](#_Toc45235043)

[Caso 3: Visualizzare e modificare informazioni nel profilo 9](#_Toc45235044)

[Diagrammi delle Attività 10](#_Toc45235045)

[Autenticazione di un utente 10](#_Toc45235046)

[Attività in modalità User 11](#_Toc45235047)

[User – Composizione del carrello 12](#_Toc45235048)

[User – Conferma dell’ordine 13](#_Toc45235049)

[Progettazione dell’architettura e implementazione del sistema 14](#_Toc45235050)

[Note sul processo di sviluppo 14](#_Toc45235051)

[Scelte tecniche e implementative 15](#_Toc45235052)

[Librerie aggiuntive 15](#_Toc45235053)

[Pattern architetturali utilizzati 17](#_Toc45235054)

[Pattern MCV 17](#_Toc45235055)

[Design pattern utilizzati 18](#_Toc45235056)

[Iterator pattern 18](#_Toc45235057)

[Observer pattern 18](#_Toc45235058)

[Decorator pattern 18](#_Toc45235059)

[Data Access Object Pattern 19](#_Toc45235060)

[Nota: Singleton Pattern 19](#_Toc45235061)

[Diagrammi di sequenza del software 21](#_Toc45235062)

[Attività di test e validazione 23](#_Toc45235063)

# Requisiti ed interazioni utente-sistema

## Casi d’uso e relativi 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.

### Casi dell’User

#### Caso *1: Composizione del carrello*

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

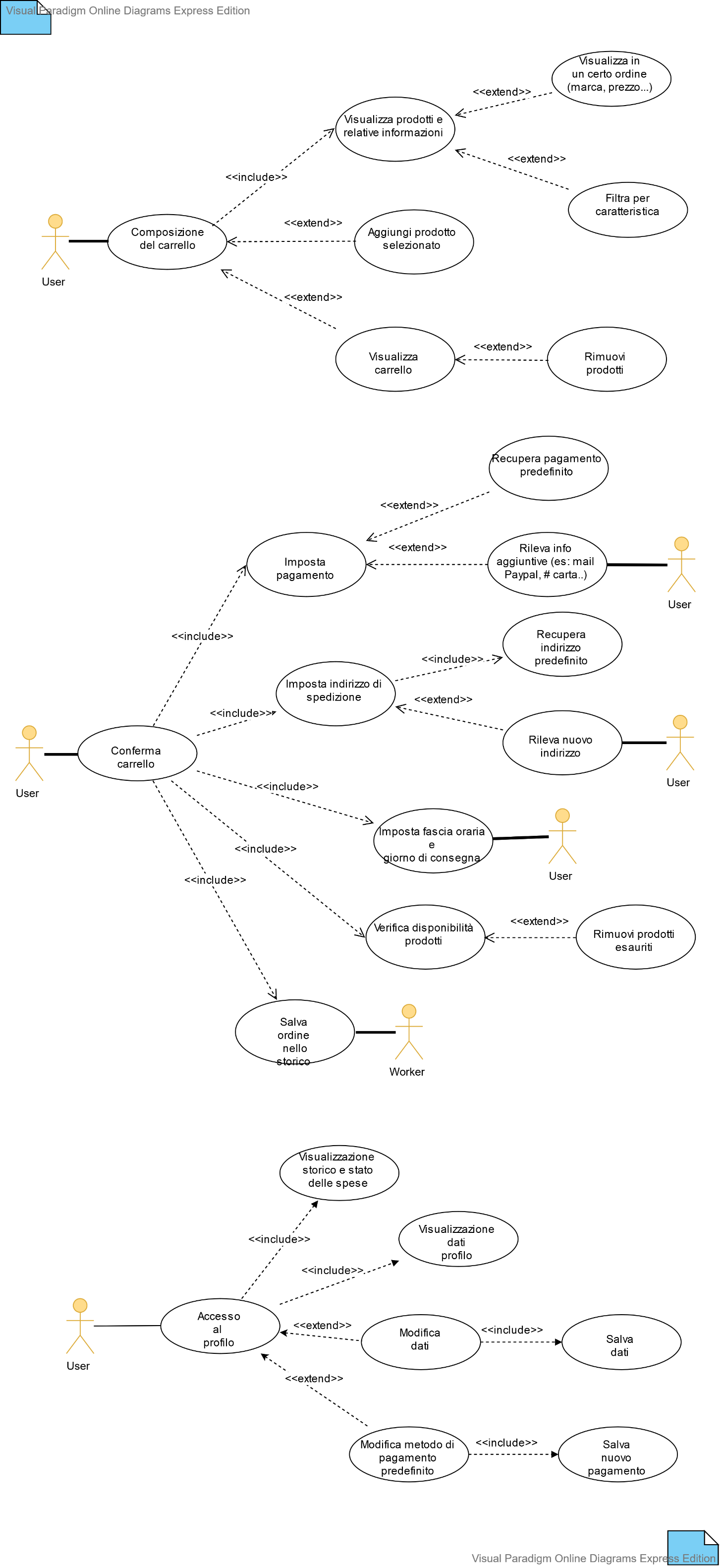


Figura 1. Caso d'uso: composizione del carrello

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

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

***Passi:***

1. *L’utente visualizza i prodotti disponibili selezionando un reparto o effettuando una ricerca*
   1. *Se desiderato, l’utente usa i pulsanti per filtrare i prodotti in base alla caratteristica; sono disponibili filtri per prodotti biologici, senza glutine, vegani e senza lattosio*
   2. *Se desiderato, l’utente cambia l’ordine di visualizzazione; sono disponibili l’ordine per prezzo crescente, per prezzo decrescente e per ordine alfabetico della marca*
2. *Se desiderato, l’utente può aggiungere un prodotto al carrello.*
3. *Se desiderato, l’utente può visualizzare il carrello*
   1. *Da qui, sempre se necessario, è possibile rimuovere dei prodotti.*

***Postcondizioni:*** *L’ordine è aggiunto alla lista degli ordini e allo storico dell’User.*

#### Caso *2: Conferma dell’ordine*

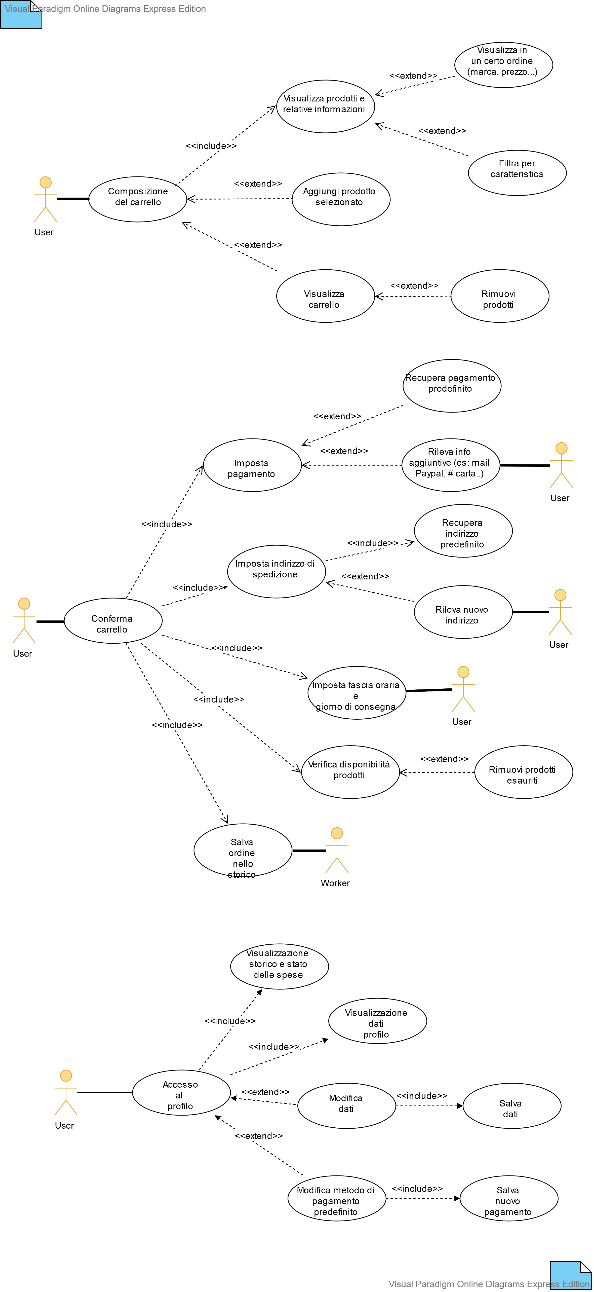


Figura 2. Caso d'uso: conferma dell'ordine

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

***Precondizioni:*** *L’utente deve essersi autenticato con una combinazione User e il carrello non deve essere vuoto.*

***Passi:***

1. L’*User* accede al proprio carrello, dove dovrà compilare i campi relativi alla consegna e al pagamento. Entrambi sono settati, di *default*, al valore salvato nel profilo utente.
2. Si verifica se tutti i prodotti sono ancora disponibili.
3. Se desiderato, l’utente può comunque cambiare indirizzo e pagamento per quest’ordine. I nuovi dati verranno associati all’ordine, ma non verranno modificati quelli “predefiniti” nel profilo.
   1. Se viene selezionato il pagamento alla consegna, non viene richiesta alcuna info aggiuntiva
   2. Se viene selezionato uno degli altri due pagamenti, all’*User* è richiesto di compilare alcune informazioni aggiuntive ad essi legati.
4. All’*User* è richiesto di selezionare un giorno e una fascia oraria per la consegna.

***Postcondizioni:*** *Se il controllo del punto 2 fallisce, i prodotti in eccesso sono rimossi e viene richiesta la ricompilazione. Altrimenti, si procede con gli altri punti e infine il carrello viene svuotato e l’ordine viene salvato nello storico, dove sarà eventualmente gestito da un Worker.*

#### Caso 3: Visualizzare e modificare informazioni nel profilo

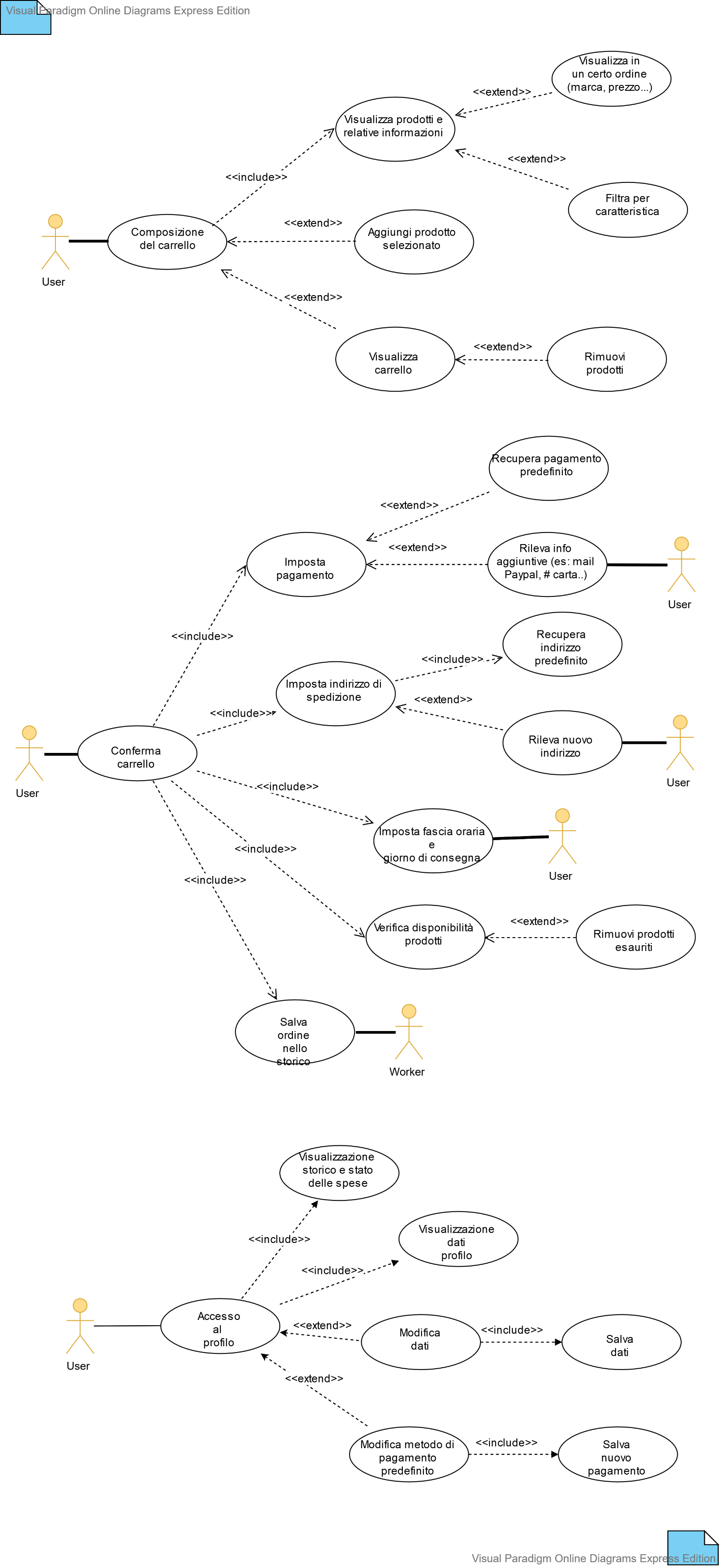


Figura 3. Caso d'uso: visualizzare e modificare informazioni nel profilo.

***Attori:*** *User*

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

***Passi:***

1. *L’utente accede al proprio profilo.*
2. *Vengono visualizzate e caricate tutte le informazioni relative al profilo e , se presenti, relative allo storico degli ordini.*
3. *Se desiderato, l’utente può modificare i propri dati di login/anagrafici e modificare il pagamento.*

***Postcondizioni:*** *Premendo sul tasto salva, ogni modifica viene salvata su file.  
Premendo sul tasto esci, ogni modifica è annullata.*

### Casi del Worker

#### Caso 1: Modificare informazioni sui prodotti

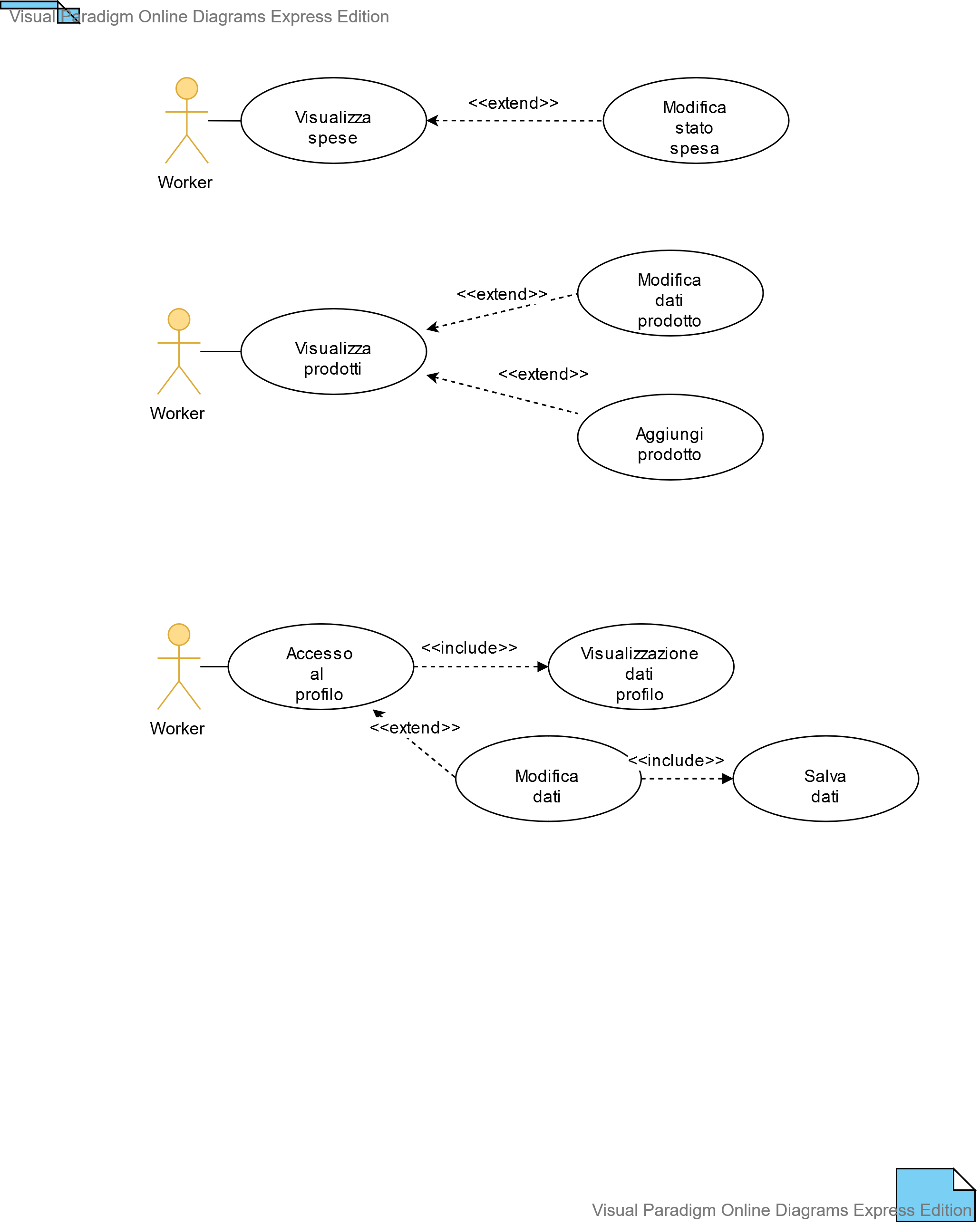


Figura 4. Caso d'uso: modificare informazioni sui prodotti

***Attori:*** *Worker*

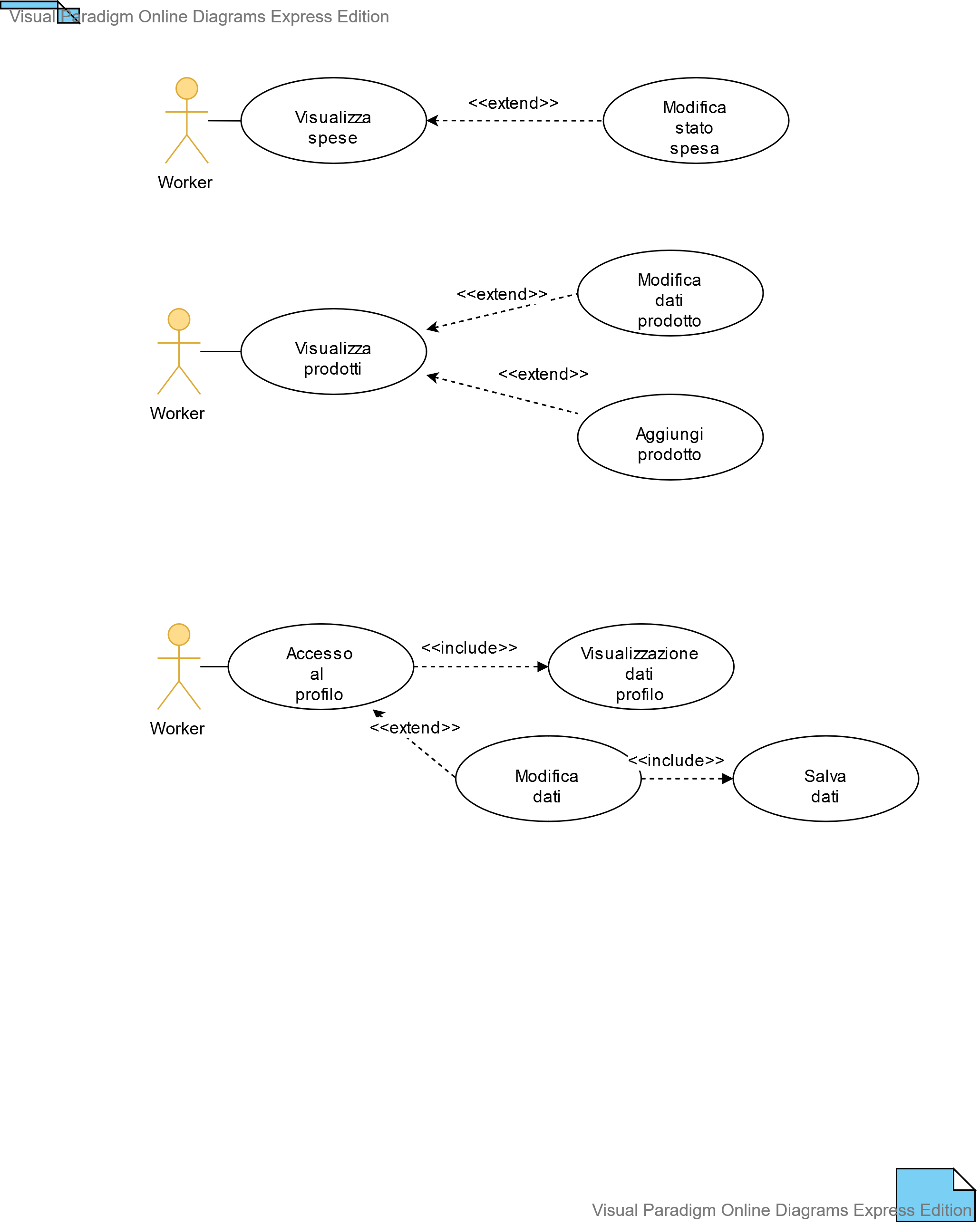
***Precondizioni:*** *L’utente deve essersi autenticato con una combinazione Worker.*

***Passi:***

1. *L’utente accede al modulo di gestione prodotti.*
2. *Viene caricato l’elenco dei prodotti attualmente memorizzati dal sistema.*
   1. *Se desiderato, l’utente può cliccare su un prodotto per visualizzarne l’elenco completo delle informazioni e modificarle.*
   2. *Se desiderato, l’utente può cliccare sul pulsante “Aggiungi nuovo prodotto” per aggiungere un nuovo prodotto.*

***Postcondizioni:*** *Se il Worker preme il tasto Salva ogni modifica è salvata su file. In caso contrario, ogni modifica viene annullata.*

#### Caso 2: Visualizzare e modificare stato degli ordini



***Attori:*** *Worker*

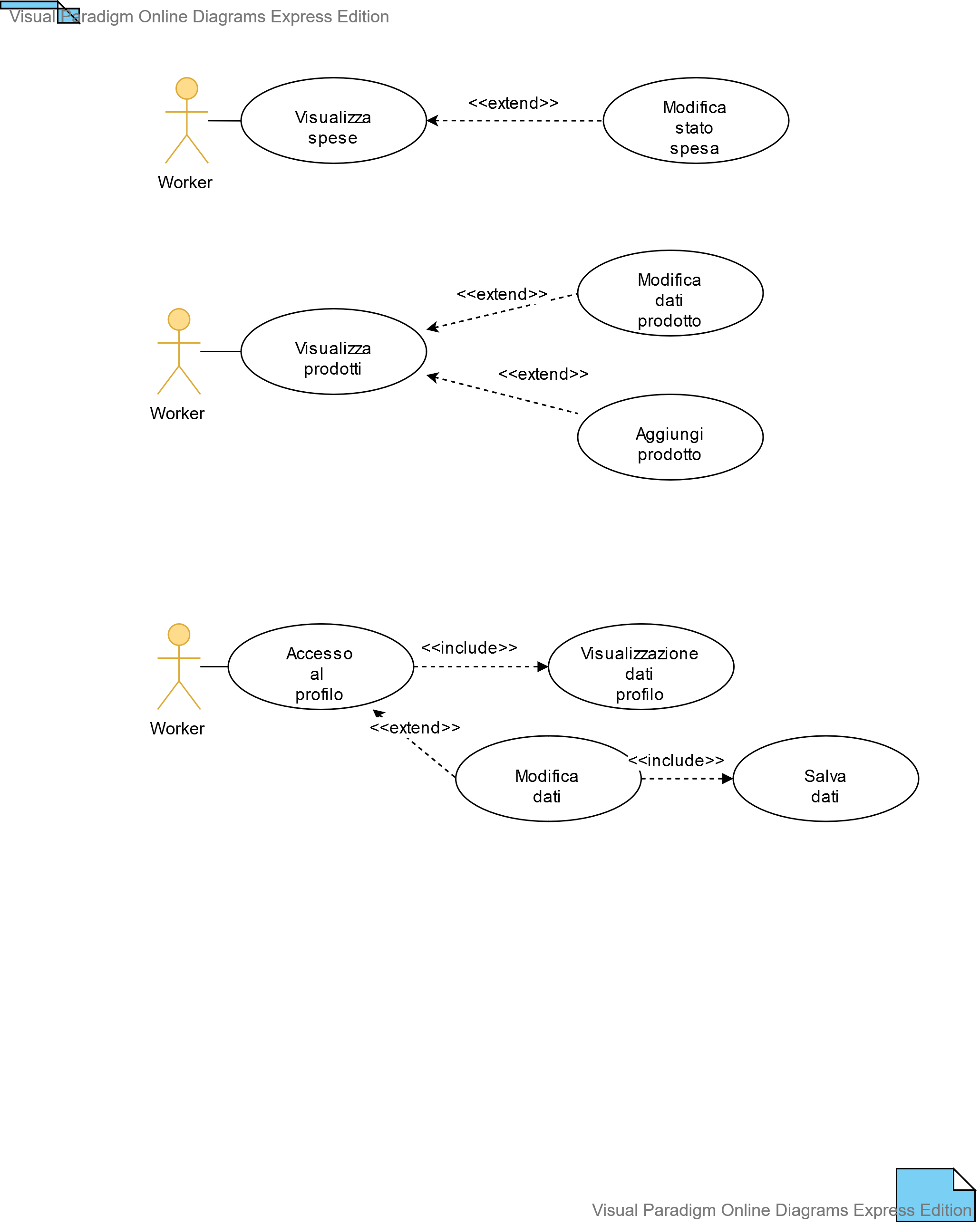
***Precondizioni:*** *L’utente deve essersi autenticato con una combinazione Worker.*

***Passi:***

1. *L’utente accede al modulo di gestione degli ordini.*
2. *Viene caricato l’elenco degli ordini attualmente gestiti nel sistema.*
   1. *Se desiderato, l’utente può modificarne lo stato.*

***Postcondizioni:*** *Se il Worker preme il tasto Salva ogni modifica è salvata su file. In caso contrario, ogni modifica viene annullata.*

#### Caso 3: Visualizzare e modificare informazioni nel profilo



***Attori:*** *Worker*

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

***Passi:***

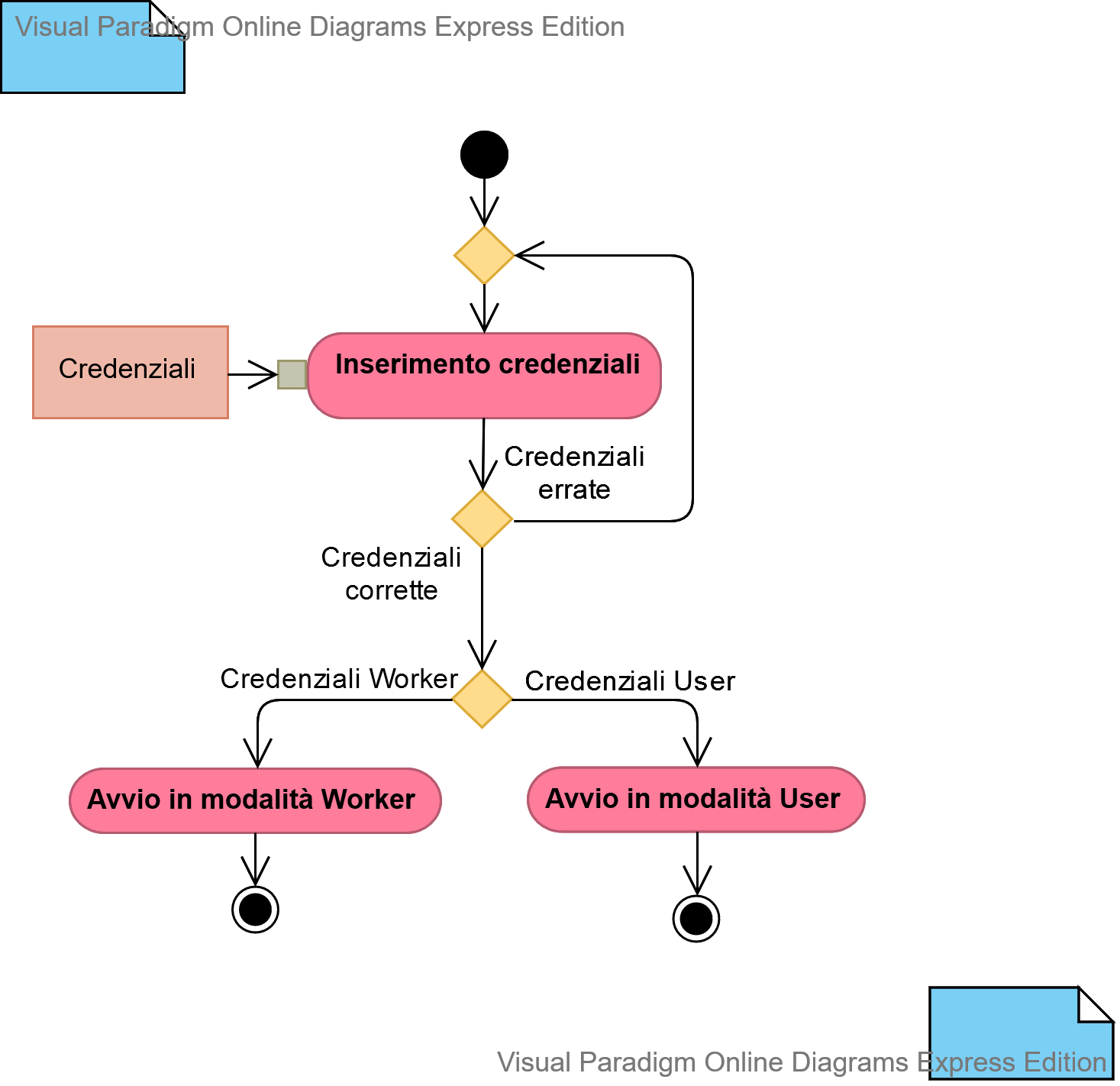
1. *L’utente accede al proprio profilo.*
2. *Vengono visualizzate e caricate tutte le informazioni relative al profilo e , se presenti, relative allo storico degli ordini.*
3. *Se desiderato, l’utente può modificare i propri dati di login/anagrafici.*

***Postcondizioni:*** *Se il l’utente preme il tasto Salva ogni modifica è salvata su file. In caso contrario, ogni modifica viene annullata.*

## Diagrammi delle Attività

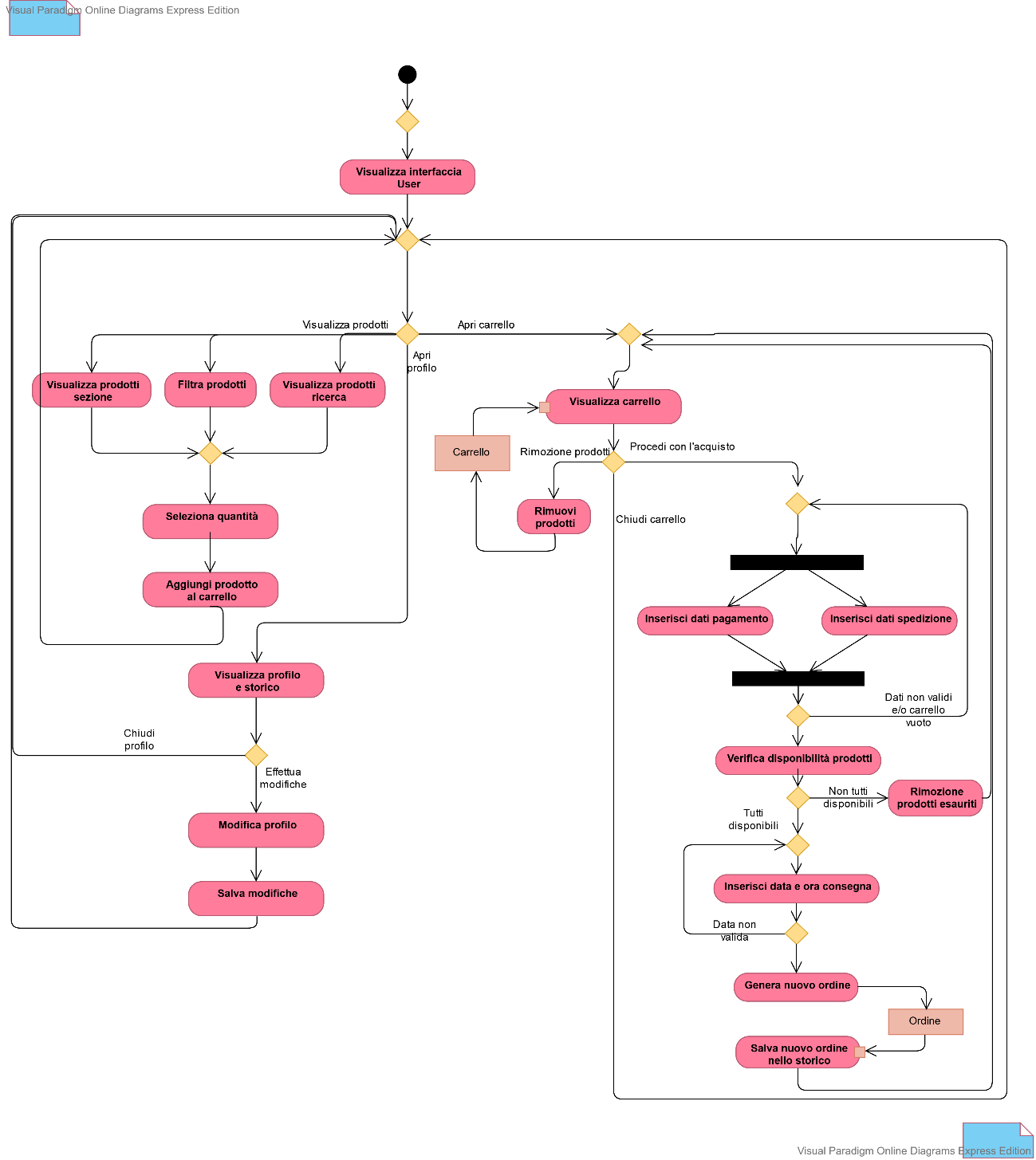
### Autenticazione di un utente

Questo diagramma rappresenta il flusso di esecuzione dell’autenticazione. Lo si può interpretare come un diagramma al quale “attaccare”, in corrispondenza dei nodi finali, i diagrammi delle modalità User e Worker – qui riportati successivamente per questioni logistiche.

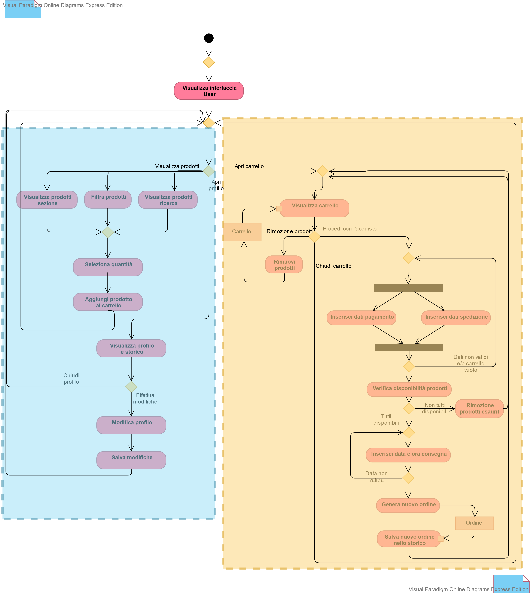


### Attività in modalità User

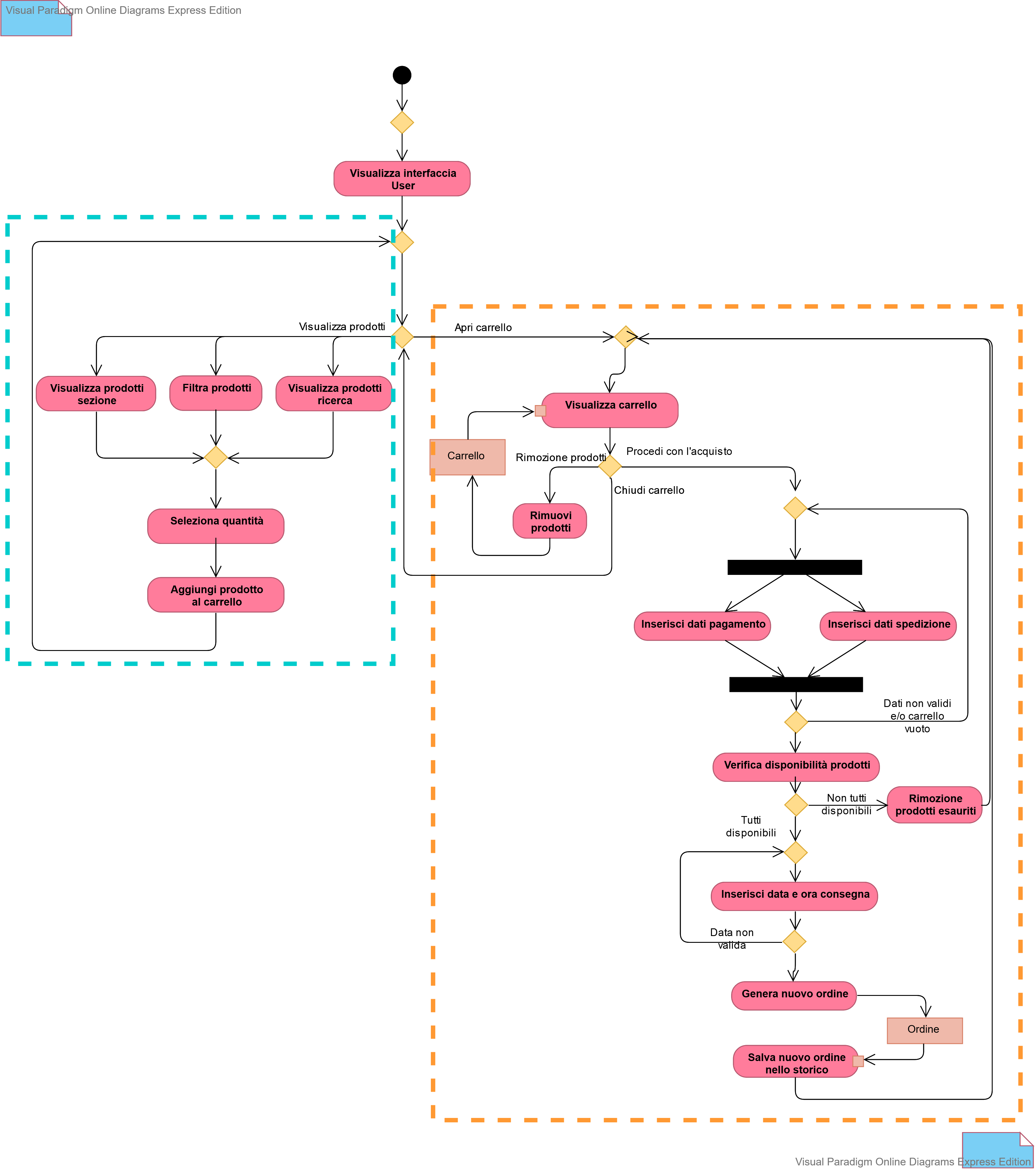
Il diagramma delle attività della modalità user è il seguente:



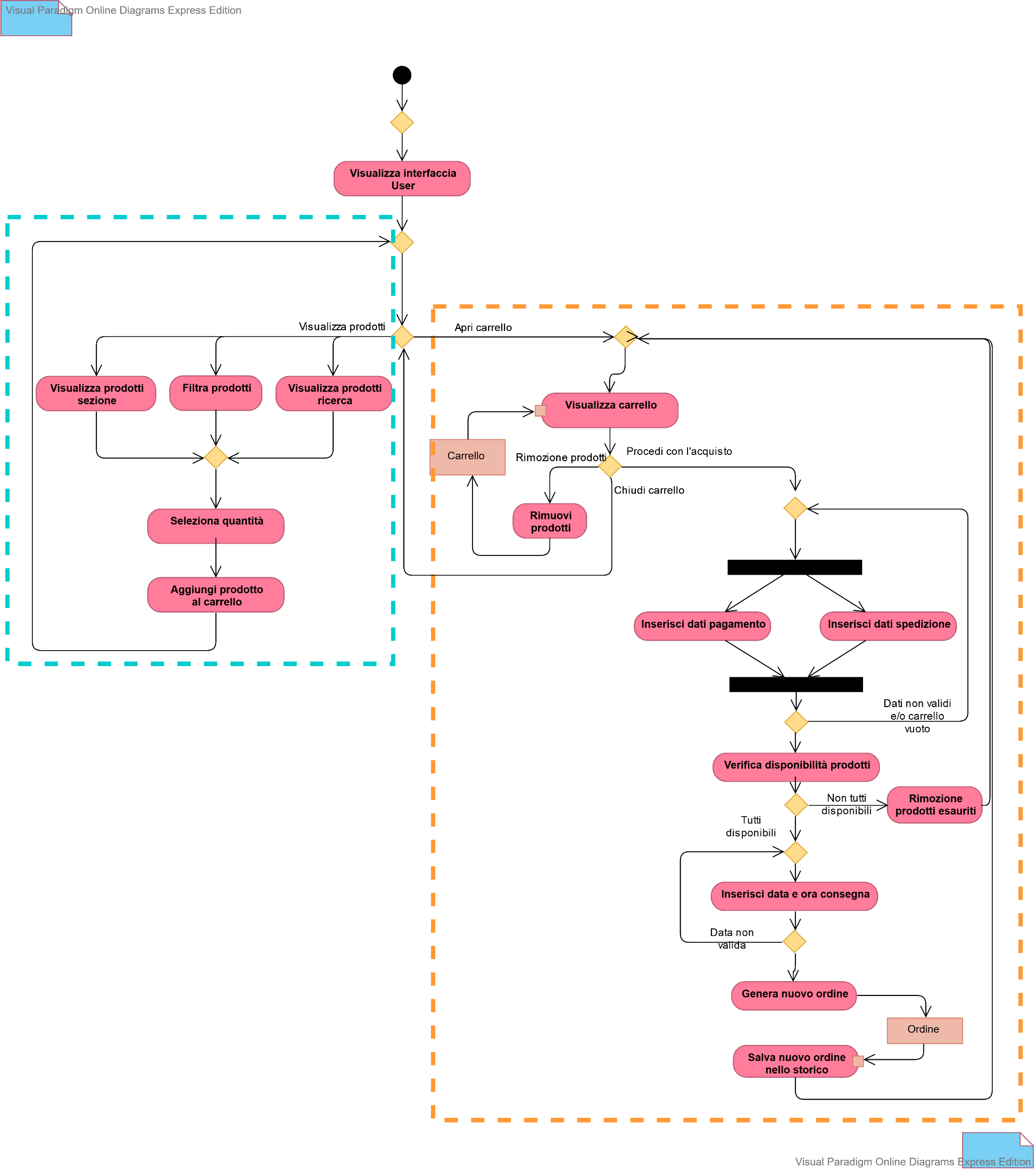
Trattandosi, di uno schema arzigogolato lo riportiamo di seguito separando la parte di composizione del carrello da quella degli acquisti e modifica profilo, come in figura.



#### User – Composizione del carrello



#### User – Conferma dell’ordine



### Attività in modalità Worker

# Progettazione dell’architettura e implementazione del sistema

## Note sul processo di sviluppo

Il processo di sviluppo si è rivelato sicuramente più complesso del previsto, poiché a causa delle pesanti restrizioni imposteci dalla situazione mondiale corrente non è stato possibile vederci di persona e lavorare insieme. Questo ci ha spinto a 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.

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 sistema. 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 e 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 vantaggiosa per i seguenti motivi:

* + Gestione quasi del tutto automatica del sistema dei Listener, rimpiazziati dai 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 molto 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, a seconda delle necessità.

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*, esse 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 le seguenti motivazioni:

* Le componenti di Vista e Controllore sono già parzialmente implementate all’interno di *JavaFX*, ovvero sulla libreria grafica che abbiamo deciso di utilizzare per i motivi illustrati in precedenza;
* 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. Il modello *MCV* si presta perfettamente a questo genere di necessità.

## 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:

### Iterator pattern

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

### Observer pattern

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*.

### Decorator pattern

Abbiamo implementato questo pattern per poter gestire con facilità l’esistenza degli ordini. Nel dettaglio, per rappresentare un ordine avevamo bisogno di poter aggiungere al carrello semplice – il quale contiene solo la lista dei prodotti – ulteriori informazioni, quali l’utente che ha evaso l’ordine, il pagamento o la consegna. Per raggiungere questo scopo senza dover modificare la classe Cart, abbiamo deciso di realizzare la classe Order con il decorator pattern: essa cambia il comportamento esterno di un Cart semplice aggiungendo ulteriori proprietà e metodi senza tuttavia cambiare quelli del Cart originale, e funzionando di fatto come una sorta di classe Wrapper.

Abbiamo applicato lo stesso sistema anche alla classe Draft, che è un carrello al quale aggiungiamo l’utente che lo ha composto.

### Data Access Object Pattern

Per quanto riguarda la rappresentazione in memoria dei dati da gestire, ci siamo ispirati al *DAO* *pattern:* la sua versione originale risultava un po’ troppo complicante rispetto alle noste necessità, quindi ci siamo ispirati al suo funzionamento 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.

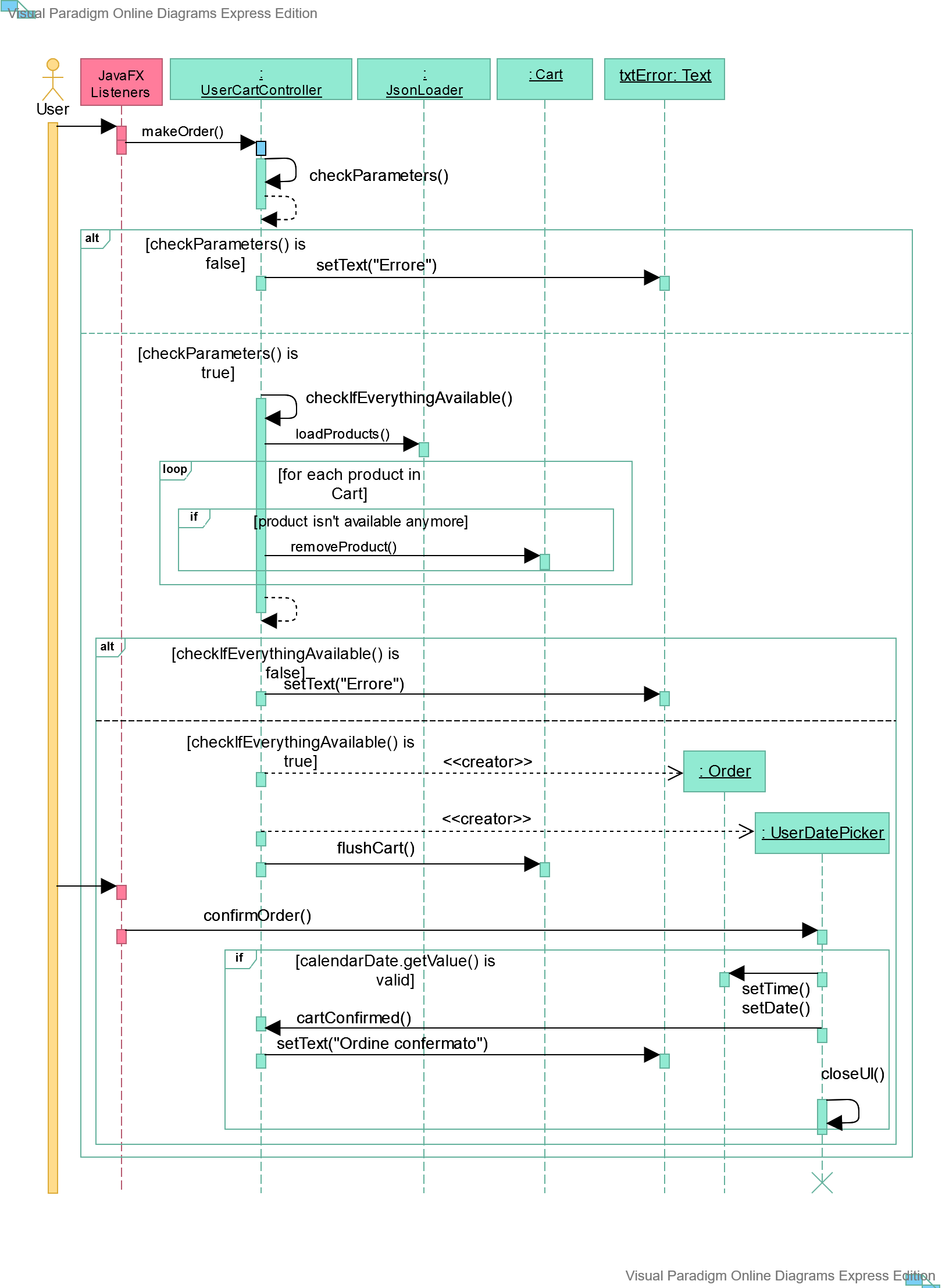
Ci siamo ispirati a questo modello per quanto riguarda l’avere una classe specifica che si preoccupasse di recuperare le informazioni dai file e rappresentarle in maniera opportuna in memoria. A differenza del *pattern* originale, tuttavia, abbiamo ritenuto superfluo definire anche un’interfaccia, e 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.

## 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.