UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e implementazione in Python di un sistema di gestione di un punto vendita di prodotti a kilometri zero

Design and implementation in Python of a management system for a selling point of zero kilometer products

Candidato:
Riccardo Schiavoni

Relatore:
Prof. Domenico Ursino

Anno Accademico 2020-2021

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione

Progettazione e implementazione in Python di un sistema di gestione di un punto vendita di prodotti a kilometri zero

Design and implementation in Python of a management system for a selling point of zero kilometer products

Candidato:
Riccardo Schiavoni

Relatore:
Prof. Domenico Ursino

Anno Accademico 2020-2021

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'Automazione
Via Brecce Bianche – 60131 Ancona (AN), Italy

Introduzione

Alla base di questo studio vi è l'analisi delle fasi che compongono lo sviluppo di un'applicazione secondo le linee guida e le best practice dell'ingegneria del software. Questa disciplina si occupa dei processi produttivi e delle metodologie implementative finalizzate alla realizzazione di sistemi informatici. Lo sviluppo viene suddiviso in più attività, ciascuna delle quali analizza il software da diversa prospettiva, fornendo, in questo modo, una descrizione progressivamente più dettagliata.

Di seguito sono riportate le fasi principali di cui si compone lo sviluppo:

- *Analisi dei requisiti:* si tratta dello studio preliminare del dominio del software per realiazzare una stima della fattibilità e dei costi.
- *Progettazione:* definizione delle proprietà essenziali della struttura in funzione dei requisiti di sistema; viene realizzata, mediante un'analisi del progetto architetturale e dei processi nel dettaglio, una "soluzione del problema".
- *Codifica:* l'implementazione dei requisti del sistema, seguendo le metodologie scelte per la realizzazione.
- *Collaudo:* in questa fase vengono effettuate le operazioni di ispezione e di debugging per il controllo dei risultati prodotti dal programma durante l'esecuzione.
- *Manutenzione:* comprende le sottoattività necessarie alle modifiche del prodotto successive alla distribuzione.

I modelli e gli strumenti proposti dall'ingegneria del software offrono delle soluzioni generiche e personalizzabili per diverse classi di problemi progettuali; le regole e le tecniche utilizzate, differenti a seconda del tipo e delle finalità del sistema in esame, identificano e formalizzano le attività necessarie alla realizzazione di un software ben strutturato. Tale approccio, oltre a ridurre sensibilmente la complessità in fase di sviluppo, permette sia di diminuire gli errori sia di ottimizzare le prestazioni e i risultati. Infatti, scegliere adeguatamente il modello procedurale da adottare, in accordo con quelle che sono le specifiche e le esigenze del progetto, consente di realizzare un sistema software efficiente, mantenibile e sicuro.

In particolare, il sistema in esame è un applicativo finalizzato alla gestione di un'attività commerciale; nelle varie sezioni dell'elaborato verranno discusse ed analizzate le scelte fatte e le metodologie utilizzate durante le realizzazione delle fasi del progetto.

La prima fase, riportata nel capitolo uno, riguarderà lo studio del contesto di riferimento e, successivamente, la definizione delle *specifiche* del sistema. Ciascuna di esse identifica uno dei *requisiti*, ovvero le proprietà che il software deve garantire e i vincoli che devono essere rispettati. L'analisi e la definizione dei requisti del sistema rappresenta, di fatto, l'attività più importante dell'ingegneria del software; in questa fase, infatti, verranno fornite le linee guida, nonchè le regole e i vincoli progettuali, da seguire durante tutte le successive attività di sviluppo.

Una volta acquisita una conoscenza qualitativa su *cosa* il software deve fare, bisognerà stabilire *come* strutturarlo. Nella fase successiva, quella di *progettazione*, si dovranno definire le caratteristiche strutturali e comportamentali dell'applicativo. Nel secondo e nel terzo capitolo verranno analizzate, rispettivamente, l'organizzazione delle classi e la logica delle funzionalità del sistema.

Innanzitutto, dovrà essere definita l'architettura da utilizzare, individuando, dapprima, quali sono i componenti da cui è formato il nostro sistema. L'analisi effettuata permetterà di definire, in accordo con quelli che sono i vincoli e i requisiti del progetto, sia la struttura che le funzionalità di ciascun sottosistema. In seguito, verranno studiate le singole funzionalità del software; ciascuna verrà descritta esaminando la logica implementativa delle operazioni, il flusso di azioni associato e gli output prodotti.

I risultati ottenuti forniranno i parametri necessari per stabilire le metodologie e gli strumenti dell'ingegneria del software da utilizzare durante la fase di implementazione.

L'ultima sezione riguarderà la *codifica* e il collaudo del software, presentati, rispettivamente, nel quarto e nel quinto capitolo; verranno mostrate le soluzioni implementative adottatte e i successivi controlli delle stesse. La descrizione di questa fase fornirà una panoramica completa sulla realizzazione del back-end e del front-end del programma. Verranno presentato il linguaggio *Python* e le librerie utilizzate durante la codifica e il testing; I moduli illustrati forniranno gli strumenti necessari sia per la memorizzazione, per la gestione e per l'aggiornamento dei dati, sia per lo sviluppo grafico. In seguito, verrà introdotto l'ambiente di sviluppo *Pycharm* e il tool *QtDesigner* per la creazione delle interfacce utente.

Ciascuno degli aspetti analizzati nel corso della trattazione verrà supportato, per una maggiore chiarezza, da immagini, opportunamente contestualizzate, attraverso le quali verranno illustrati i risulati ottenuti, le operazioni effettuate, e i vari esempi forniti. Tutti gli strumenti impiegati sia in fase di analisi e di progettazione, sia durante lo sviluppo e il testing, verranno introdotti e descritti prima di mostrarne l'utilizzo.

Il software ottenuto al termine del progetto, dovrà garantire le caratteristiche che ne identificano la qualità. Le funzionalità sviluppate devono soddisfare i requisiti richiesti, inoltre, la struttura del sistema dovrà essere tale da rendere possibile le successive attività di manutenzione e di aggiornamento.

Indice

Capitolo 1

Il Contesto di Riferimento

Il presente capitolo ha come obiettivo quello di offrire una panoramica generale sul contesto di riferimento, nel quale il software andrà ad operare, per poi poter svolgere un'analisi preliminare volta a stabilire quali siano nel dettaglio le funzionalità e le finalità del programma. Verranno, dunque, fornite tutte le premesse per stabilire delle linee guida durante la fase di progettazione.

1.1 Descrizioni dello Scenario

1.1.1 Descrizione del Software

Il software in questione è finalizzato all'organizzazione e alla gestione dei dati relativi ad un negozio di alimentari.
Si tratta di un punto vendita specializzato nella distrubuzione di prodotti a kilometro 0. La merce in vendita infatti ha un'origine rigorosamente locale: il rifornimento avviene esclusivamente da diverse aziende del posto, garantendo, in questo modo, la qualità degli acquisti. Il cliente può trovare sia alimenti grezzi, o semilavorati, e bevande alcoliche tradizionali sia cibi cucinati pronti per l'asporto.
I prodotti utilizzati in cucina sono gli stessi che possono essere acquistati in negozio, questa strategia permette di ottenere una maggiore diversificazione della merce venduta e, di conseguenza, della tipologia di acquirenti senza, però, rinunciare alla qualità del servizio offerto.
Il negozio, inoltre, offre la possibilità di ordinare i prodotti e stabilire una data per il ritiro, la scelta di mettere a disposizione il servizio d'asporto garantisce una notevole agevolazione sulle modalità di acquisto ma permette, anche, una migliore gestione degli alimenti ed una riduzione della merce invenduta
L'altro aspetto d'interesse è quello riguardante la contabilità del punto vendita: si dovrà tener traccia di tutte le transazioni avvenute, sia in ingresso che in uscita, in modo da poter avere un monitoraggio costante del bilancio totale.
Il software dovrà dunque rappresentare un supporto informatico, grazie al suo utilizzo il titolare potrà facilmente monitorare la merce in negozio e migliorare il rapporto con clienti e fornitori.

Si è optato quindi per implementare un'interfaccia divisa in 4 sezioni principali: gestione dei prodotti in negozio, gestione di tutti gli ordini, gestione dei vari enti che interagiscono con l'attività e gestione di tutte le transazioni avvenute.
La prima sezione contiene tutti i prodotti acquistabili in negozio, per ciascun prodotto, oltre ad essere associato un codice prodotto univoco, si dovrnno indicare: la tipologia, il nome, il prezzo di vendita, l'eventuale percentuale di sconto e la quantità attualmente disponibile disponibile.
L'interfaccia permetterà la visualizzazione di tutti i prodotti registrati, oppure, di selezionarne soltanto una classe d'interesse impostando dei filtri di ricerca sui vari parametri.
Il titolare, inoltre, potrà aggiungere nuovi prodotti o modificare ed eliminare quelli già esistenti, nel caso in cui non siano presenti in ordini il cui pagamento non è stato ancora effettuato.
Per quanto riguarda l'area dedicata alla gestione degli ordini e ai relativi dettagli, essa dovrà offrire la possibilità di visualizzare e gestire in maniera distinta quelli relativi agli acquisti dei clienti e quelli relativi alle richieste per i fornitori.
Tuttavia le funzionalità offerte sono analoghe: possibilità di visualizzare tutti gli ordini oppure, selezionando criteri sulle informazioni registrate, quali data di emissione e ritiro, prezzo, utente associato e stato di pagamento, di mostrarne soltanto una parte; creazione di un nuovo ordine e, nel caso di quelli non ancora pagati, eliminazione degli stessi o visualizzazione e modifica della lista dei prodotti associati; registrare la transazione quando uno degli ordini viene saldato.
La terza interfaccia sarà quella relativa a clienti, dipendenti e fornitori, per ciascuno di essi dovà essere visualizzato: un codice identificativo, il nome e il cognome, il numero telefonico e l'indirizzo email.
Nel caso dei dipendenti sono necessarie anche le informazioni relative allo stipendio, mentre per i fornitori la tipologia degli alimenti prodotti.
Quest'area avrà un'impostazione simile alle precedenti, si portà scegliere ogni volta la specifica classe di utenti da visualizzare e, eventualmente, modificare.
In questo modo il titolare potrà gestire singolarmente ognuna delle categorie. La logica di eliminazione segue una logica analoga a quella dei prodotti: non potranno essere eliminati fornitori o clienti associati ad ordini non ancora pagati.
Per quanto riguarda la sottosezione dei dipendenti, il titolare potrà, nell'apposita area di modifica, registrazione della transazione relativa al pagamento dello stipendio dei singoli impiegati.
L'ultimo slot è quello dedicato alle transazioni, in questa sezione sono registrati tutti i pagamenti avvenuti in negozio, sia quelli relativi agli ordini sia quelli relativi agli stipendi. Inoltre saranno riportati gli importi totali delle uscite e delle entrate delle transazioni correntemente visualizzate nello schermo; in questo modo il titolare avrà una costante panormamica sui bilanci in vari periodi d'interesse.
Un'ulteriore aspetto da evidenziare è quello dell'utilizzo del programma. Esso è, infatti, riservato esclusivamente al titolare del punto vendita o di chi ne fa le veci;

dovranno, dunque, essere messe a disposizione la funzionalità di login e di richiesta di codice di conferma per validare ogni operazione eseguibile.

1.1.2 Gloassario dei Termini

Nella descrizione del software e dell'attività commerciale sono stati inidviduati alcuni termini specifici che richiedono di essere ulteriormente analizzati in modo da non creare ambiguità in fase di sviluppo del software.
Per tal regione è stato realizzato un glossatio dei termini, riportato nella figure 1.1

Termine	Descrizione	Tipologia	Sinonino
Negozio di Alimentari	Negozio specializzato nella vendita di generi alimentari, può essere specializzato in una determinata tipologia di prodotti oppure generico (grocery)	Business	Alimentari, Drogheria
Alimenti Grezzi	Prodottl agricoll in condizioni uguali a quelle originarie al momento della raccolta	Tecnino	Alimenti naturali, Alimenti non lavorati
Semilavorati	Prodotti ottenuti attraverso un parziale processo di lavorazione e trasformazione di prodotti provenienti da attività primarie quali l'agricoltura	Tecnico	Nessuno
Bevande alcoliche	Bevande contenti alcool etilico in diverse gradazioni: alcolici ottenuti attraverso la fermentazione, ovvero vini bianchi o rossi, e bevande spiritose, cioè amari, distillati e liquori	Tecnico	Alcolici
Tipologia	Si tratta della tipologia di gruppo alimentare a cui appartiene il prodotto	Tecnico	Classe Alimentare
Fornitori	Aziende o singoli specializzate nella produzione di bevande o alla realizzazione di altri prodotti finiti e semilavorati	Fornitori	Grossista
Ordini	Lista dei prodotti ordinati dal cliente oppure di prodotti, di una specifica tipologia, ordinati dal negozio ai relativi fornitori	Business	Nessuno
Transazioni	Registrazione dei pagamenti, in ingresso e in uscita, avvenuti in negozio	Business	Nessuno
Codice Prodotto	Identificatore univoco del prodotto: si tratta di un codice numerico generato in automatico dal sistema	Tecnico	Nessuno
Codice Identificativo	Identificatore univoco dell'utente: può essere il codice fiscale nel caso di clienti e dipendenti o partita iva nel caso delle aziende e i fornitori	Tecnico	Nessuno

Figura 1.1: Glossario dei Termini

1.2 Raccolta e Analisi dei Requisiti

La descrizione generale dell'attività e del software ci ha permesso di comprendere quali sono le funzionalità che esso deve garantire e i vincoli da rispettare affichè l'utilizzo sia conforme alle esigenze del commitente.
In questa sezion vengono illustrati i requisiti del sistema; ogni parte precedentemente descritta deve prevedere le operazioni di visualizzazione, totale o parziale, creazione, modifica ed eliminazione dei relativi elementi gestiti.
La tabella, nella figura 1.3, elenca tutte le funzionalità che verranno implementate descrivendo se sono presenti ulteriori sottovincol;: l'eliminazione dei prodotti, dei clienti e dei fornitori l'eliminazione non deve essere permessa qualora siano associati ad ordini che non sono stati ancora saldati.
Invece la modifica e l'eliminazione degli ordini sono permesse solo nel caso in cui il pagamento non sia stato ancora effettuato. Inoltre saranno elencati anche i requisiti non funzionali, ovverole caratteristiche del software non richieste dal cliente, ma che stabiliscono delle regole progettuali che dovranno essere rispettate nella fase di sviluppo (figura 1.4).

Requisiti Funzionali	Descrizione
RF1: Visualizzazione Lista Prodotti	Il sistema dovrà gestire la visualizzazione di tutti prodotti
RF2: Visualizzazione Lista Prodotti Filtrata	Il sistema dovrà gestire la visualizzazione di tutti i prodotti che soddisfano determinati requisiti
RF3: Inserimento Prodotto	Il sistema dovrà gestire l'inserimento di un nuovo prodotto
RF4: Modifica Prodotto	Il sistema dovrà gestire la modifica dei dati relativi ai prodotti
RF5: Eliminazione Sicura Prodotto	Il sistema dovrà gestire l'eliminazione dei prodotti controllando che questi non siano associati a nessun ordine non saldato
RF6: Visualizzazione Lista Clienti	Il sistema dovrà gestire la visualizzazione di tutti i clienti
RF7: Visualizzazione Lista Clienti Filtrata	Il sistema dovrà gestire la visualizzazione di tutti i clienti che soddisfano determinati requisiti
RF8: Inserimento Cliente	Il sistema dovrà gestire l'inserimento di un nuovo cliente
RF9: Modifica Cliente	Il sistema dovrà gestire la modifica dei dati relativi ai clienti
RF10: Eliminazione Sicura Cliente	Il sistema dovrà gestire l'eliminazione dei clienti controllando che questi non siano associati a nessun ordine non saldato
RF11: Visualizzazione Lista Dipendenti	Il sistema dovrà gestire la visualizzazione di tutti i dipendenti
RF12: Visualizzazione Lista Dipendenti Filtrata	Il sistema dovrà gestire la visualizzazione di tutti i dipendenti che soddisfano determinati requisiti
RF13: Inserimento Dipendente	Il sistema dovrà gestire l'inserimento di un nuovo dipendente
RF14: Modifica Dipendente	Il sistema dovrà gestire la modifica di un dipendente
RF15: Eliminazione Dipendente	Il sistema dovrà gestire l'eliminazione di un dipendente
RF16: Visualizzazione Lista Fornitori	Il sistema dovrà gestire la visualizzazione di tutti i fornitori
RF17: Visualizzazione Lista Fornitori Filtrata	Il sistema dovrà gestire la visualizzazione di tutti i fornitori che soddisfano determinati requisiti
RF18: Inserimento Fornitore	Il sistema dovrà gestire l'inserimento di un nuovo fornitore
RF19: Eliminazione Sicura Fornitore	Il sistema dovrà gestire l'eliminazione dei fornitori controllando che questi non siano associati a nessun ordine non saldato
RF20: Modifica Fornitore	Il sistema dovrà gestire la modifica di un fornitore

Figura 1.2: Tabella dei Requisiti Funzionali relativa ai prodotti e agli utenti

RF21: Visualizzazione Lista Ordini Clienti	Il sistema dovrà gestire la visualizzazione di tutti gli ordini dei clienti
RF22: Visualizzazione Lista Ordini Clienti Filtrata	Il sistema dovrà gestire la visualizzazione di tutti gl ordini dei clienti che soddisfano determinati requisiti
RF23: Inserimento Ordine Cliente	Il sistema dovrà gestire l'inserimento di un nuovo ordine cliente
RF24: Modifica Ordine Cliente	Il sistema dovrà gestire la modifica di un ordine cliente non permettedola qualora sia stato saldato
RF25: Eliminazione Controllata Ordine Cliente	Il sistema dovrà gestire l'eliminazione di un ordine cliente non permettedola qualora sia stato saldato
RF26: Visualizzazione Lista Ordini Fornitori	Il sistema dovrà gestire la visualizzazione di tutti gli ordini dei fornitori
RF27: Visualizzazion Lista Ordini Clienti Filtrata	Il sistema dovrà gestire la visualizzazione di tutti gl ordini dei fornitori che soddisfano determinati requisiti
RF28: Inserimento Ordine Cliente	Il sistema dovrà gestire l'inserimento di un nuovo ordine fornitore
RF29: Modifica Ordine Cliente	Il sistema dovrà gestire la modifica di un ordine fornitore non permettedola qualora sia stato saldato
RF30: Eliminazione Controllata Ordine Cliente	Il sistema dovrà gestire l'eliminazione di un ordine fornitore non permettedola qualora sia stato saldato
RF31: Paga Stipendio Dipendente (Inserimento Nuova Transazione 1)	Il sistema dovrà gestire il pagamento dello stipendio di un dipendente (l'inserimento di una nuova transazione), nel caso in cui la mensilità corrente non sia ancora stata saldata
RF32: Salda Ordine Cliente (Inserimento Nuova Transazione 2)	Il sistema dovrà gestire il pagamento di un ordine cliente non saldato (l'inserimento di una nuova transazione)
RF33: Salda Ordine Fornitore (Inserimento Nuova Transazione 3)	Il sistema dovrà gestire il pagamento di un ordine fornitore non saldato (l'inserimento di una nuova transazione)
RF34: Visualizzazione Lista Transazioni	Il sistema dovrà gestire la visualizzazione di tutte le transazioni
RF35: VisualizzazioneLista Transazioni Filtrata	Il sistema dovrà gestire la visualizzazione di tutte le transazioni che soddisfano determinati requisiti

Figura 1.3: Tabella dei Requisiti Funzionali relativa agli ordini e alle transazioni

Requisiti Non Funzionali	Descrizione
RNF1	interfaccia dovrà essere implementata attraverso l'utilizzo di un tool grafico
RNF2	il linguaggio di riferimento dovrà essere python
RNF3	Il sistema dovrà richiedere l'utilizzo di un codice per validare le operazioni

Figura 1.4: Tabella dei Requisiti Non Funzionali

1.3 Casi d'uso

Si descriveranno ora tutti i possibili utilizzi del programma (Figura 1.5) elencando, per ciascuno, la sequenza di azioni associata ed gli errori che potrebbero compromettere il corretto esito.
Inoltre è necessario considerare che il software verrà implementato per essere utilizzato esclusivamente dal gestore del negozio o dai dipendenti da lui incaricati: il titolare sarà, dunque, considerato l'unico attore che potrà interagire con il sistema.

1.3.1 Casi d'uso relativi alle Transazioni

La prima area descritta è relativa alla gestione delle transazione (Figura 1.6).
La creazione di nuove transazioni avviene in automatico nel momento in cui viene registrato il pagamento di un ordine o dello stipendio di un dipendente.
La sequenza di operazioni che mostra come avviene la creazione di una nuova transazione sarà, dunque, fornita nelle descrizione degli eventi che la provocano.

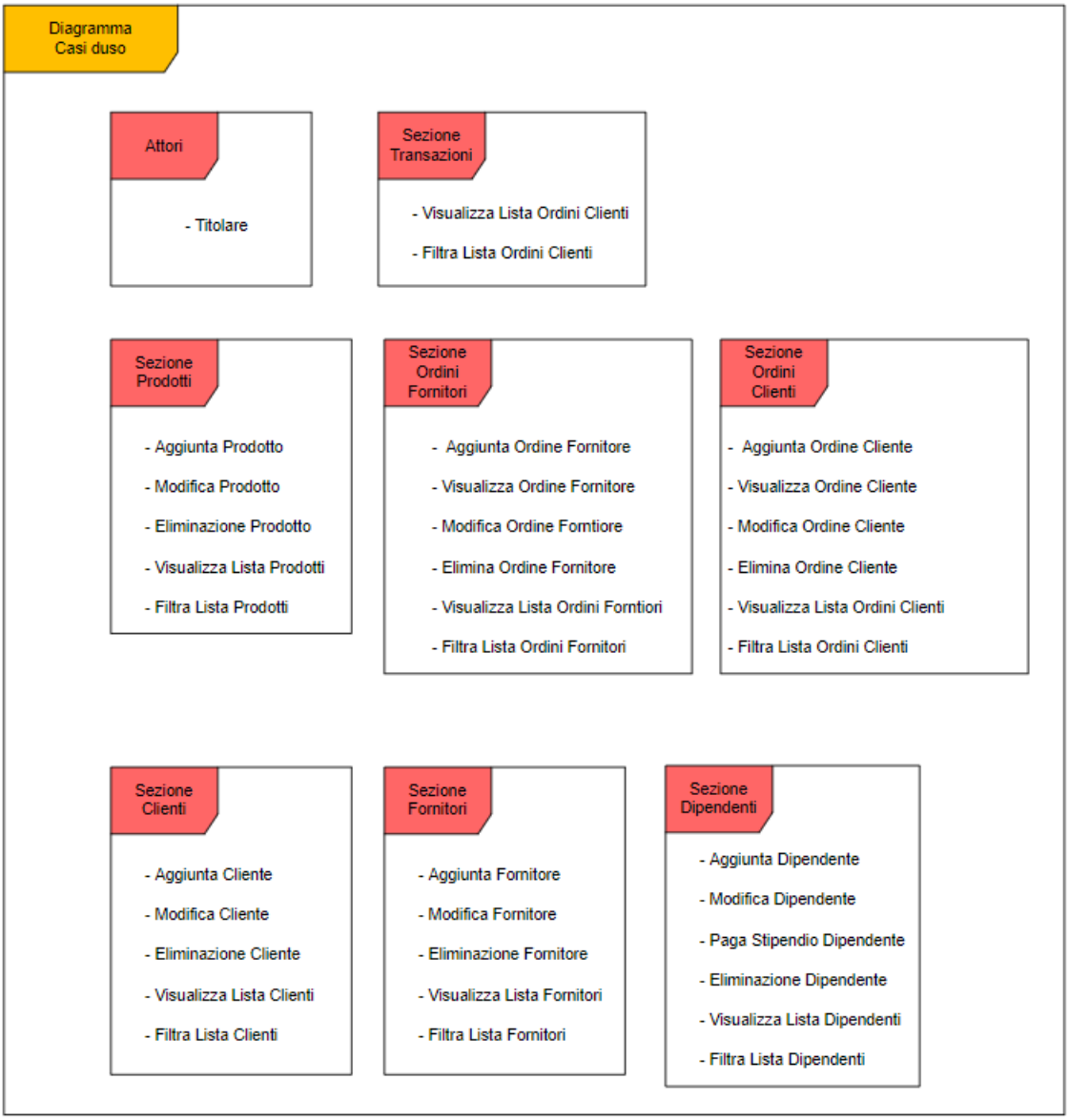


Figura 1.5: Diagrammi dei Casi d'Uso

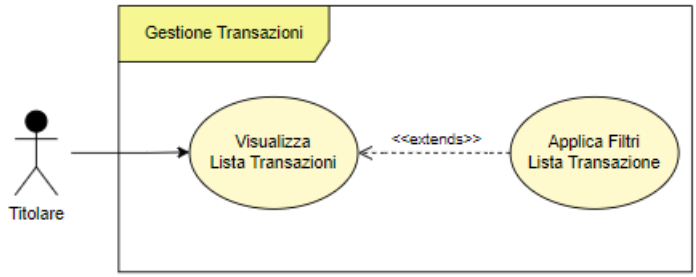


Figura 1.6: Casi d'Uso relativi alle Transazioni

Capitolo 2

Progettazione delle Classi

La progettazione e la descrizione di una struttura riguardano l'organizzazione di base di un sistema. Affinchè il software possieda determinate caratteristiche, che ne determinano la qualità, è necessario definire quali siano le componenti, le loro relazioni e le regole che le descrivono. Nel presente capitolo si analizzerà come strutturare il sistema, individuando i modelli architetturali più adatti all'implementazione

2.1 Architettura

2.1.1 Scelta dell'architettura

L'architettura del sistema, o system architecture, è un modello concettuale che descrive la struttura e il comportamento di un software.

Un sistema può essere visto come un insieme di più sottosistemi e componenti che interagiscono per svolgere diversi compiti.

Ciascuno dei singoli elementi, a seconda dei requisiti del programma a cui si riferisce, implementa determinate funzionalità ed è dotato di un'interfaccia pubblica, quest'ultima rappresenta l'insieme dei servizi messi a disposizione delle altre parti del sistema.

L'utilizzo di un'architettura ha come obiettivo quello di ridurre sensibilmente la complessità della fase di sviluppo.
La realizzazione di singole sottostrutture, comprensiva della successiva integrazione delle stesse, risulta molto più facile da organizzare rispetto a quella di un unico sistema monolite.
Un modello di progettazione, oltre ad agevolare la realizzazione di un'applicazione, offre soluzioni generiche e personalizzabili per diverse classi di problemi, in questo modo il software sarà sviluppato avvalendosi di modelli esistenti e non dovrà essere creato da zero.
Le linee guida dell'architettura permettono di ottenere un sistema ben strutturato e dotato delle seguenti caratteristiche:

- *Manutenibilità*: in caso di modifiche nei requisiti è possibile circoscrivere le modifiche da apportare ai singoli componenti coinvolti.

- *Riusabilità*: i singoli componenti possono essere riutilizzati in altri programmi, effettuando eventuali modifiche, per fornire le stesse funzionalità. Bisogna evidenziare che, oltre a ridurre i tempi di sviluppo, questa tecnica garantisce anche una certa affidabilità: i componenti in questione sono già testati e corretti da eventuali bug.

- *Comprensibilità*: L'organizzazione del codice lo rende facilmente comprensibile a chiunque debba apportare modifiche o effettuare test.

Ricordiamo che il programma, a seconda delle interazioni con un singolo attore, dovrà gestire, manipolare e visualizzare le informazioni riguardanti un determinato contesto.
Risulta, quindi, logico pensare che una prima scomposizione del nostro sistema si basi sulla tipologia dei dati.
Ogni sottosistema dovrà essere destinato alla gestione di una determinata classe di informazioni e al soddisfacimento di una parte dei requisiti.

2.1.2 Utilizzo del Pattern Model-View-Controller

Una volta individuate quali sono le diverse aree logistiche in cui può essere suddiviso il sistema l'operazione successiva consiste nell'effettuare un'ulteriore analisi per determinare, secondo le tecniche dell'ingegneria del software, quali sono i componenti che formano ogni sottosistema, quali operazioni devono effettuare e in che modo avviene la comunicazione fra di essi e con le altre parti del programma.
Considerando le funzionalità e le caratteristiche richieste per il nostro software, la scelta più adatta è quella di determinare il modo in cui realizzare i componenti seguendo il pattern architetturale Model-View-Controller.
L'idea dell' MVC è quella di separare la logica di presentazione da quella di gestione; questa metodologia identifica i vari componenti dei sottosistemi applicando un criterio di classificazione rispetto al tipo di operazioni eseguite sui dati.
In particolare il pattern prevede tre tipologie di elementi; ovvero:

- *Model*: gestisce la rappresentazione strutturale dei dati e definisce i metodi, ovvero le regole, per accervi.

- *View*: si occupa di come visualizzare i dati contenuti nel model e dell'interazione con gli attori del sistema.

- *Controller*: funge da tramite tra gli altri due componenti, analizza gli input e li converte in comandi per il modello o la vista.

L'utilizzo del pattern MVC permette di realizzare un software che possiede le caratteristiche di qualità citate precentemente.
L'indipendenza delle varie componenti, infatti, permette la suddivisione del lavoro nel caso di svilippatori con competenze diverse.

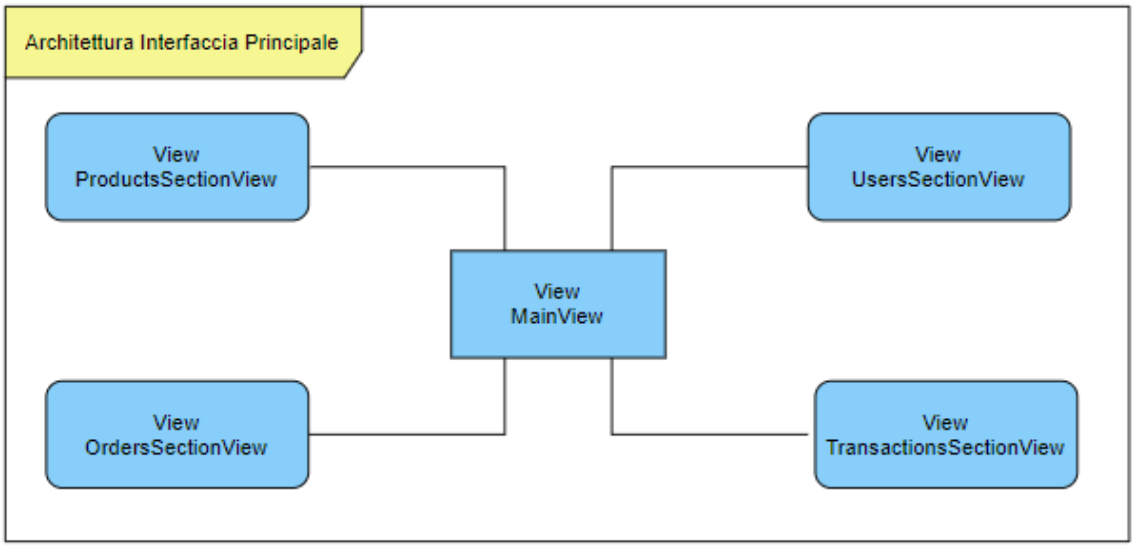


Figura 2.1: Architettura dell'Interfaccia

Inoltre, l'utilizzo di regole nella stesura del progetto facilitano un'eventuale manutenzione e migliorano la comprensibilità del codice.
Esiste anche la possibilità di scrivere viste e controllori diversi, utilizzando lo stesso modello di accesso ai dati e, quindi, di riutilizzare parte del codice già scritto in precedenza.

2.1.3 Scheletro dell'Architettura

La rappresentazione dell'architettura realizzata, per una migliore comprensione, è suddivisa in più diagrammi, ciascuno dei quali si riferisce ad una delle macroaree di interesse.
L'interfaccia principale (descritta in Figura 2.1) permette di muoversi tra le 4 sezioni (view) che verranno gestite durante l'utilizzo del programma: area relativa ai prodotti (in Figura 2.1), area dedicata alla gestione degli ordini dei clienti e dei fornitori (in Figura 2.2), area destinata al controllo di tutti gli enti che interagiscono con il punto vendità (in Figura 2.3) e area dedicata alla visualizzazione del bilancio (in Figura 2.4).

2.2 Diagramma delle classi

In questa sezione forniremo una descrizione strutturale degli oggetti che compongono il sistema, identificandone gli attributi e le operazioni, nonchè le loro relazioni.
In primo luogo, per una migliore comprensione dei diagrammi, verranno definiti alcuni concetti essenziali .
Gli oggetti in analisi sono le classi, nella Figura 2.6, ovvero i modelli degli elementi implementati all'interno del software, e le relazioni tra le loro istanze.

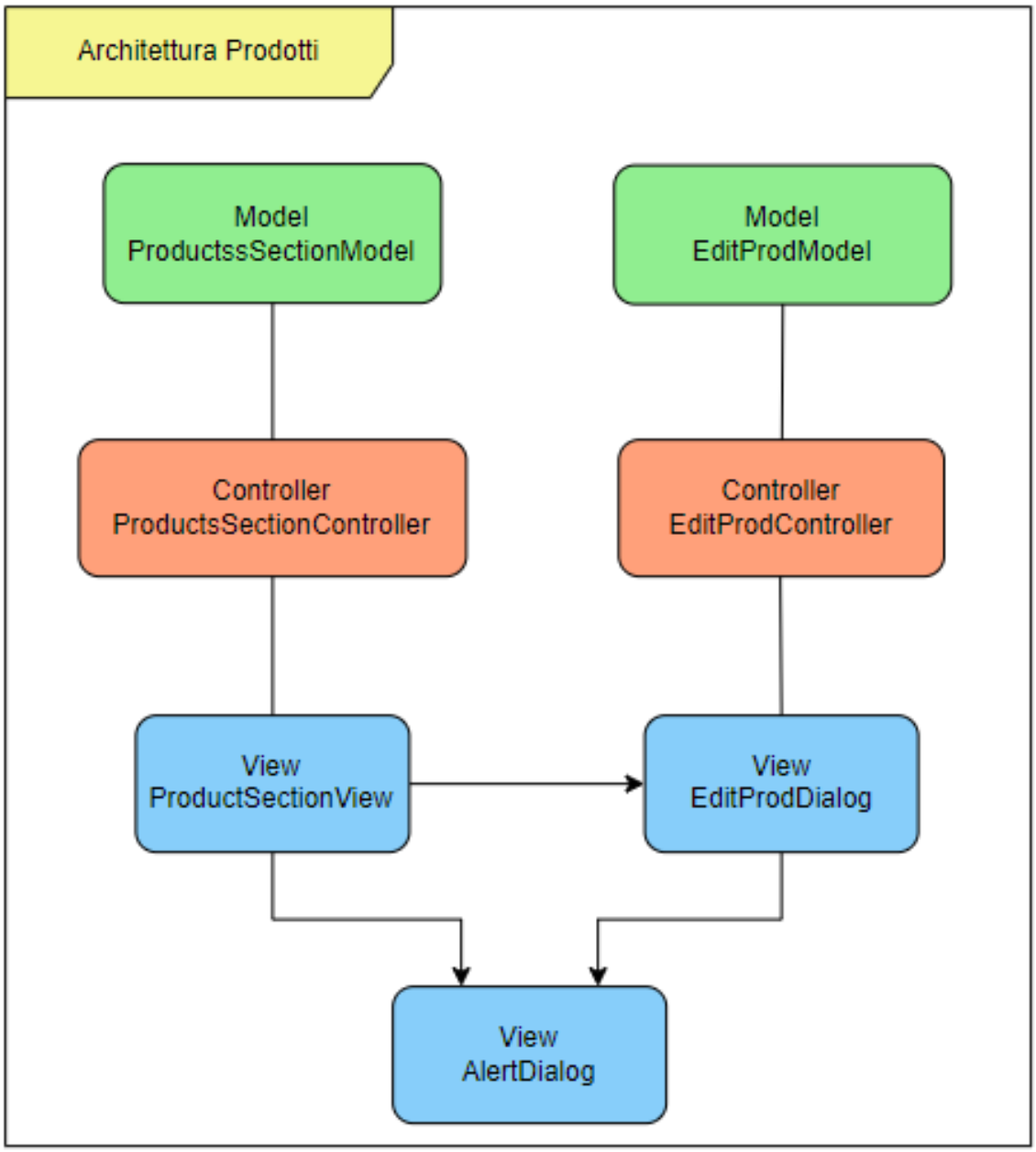


Figura 2.2: Architettura dell'Area relativa ai Prodotti

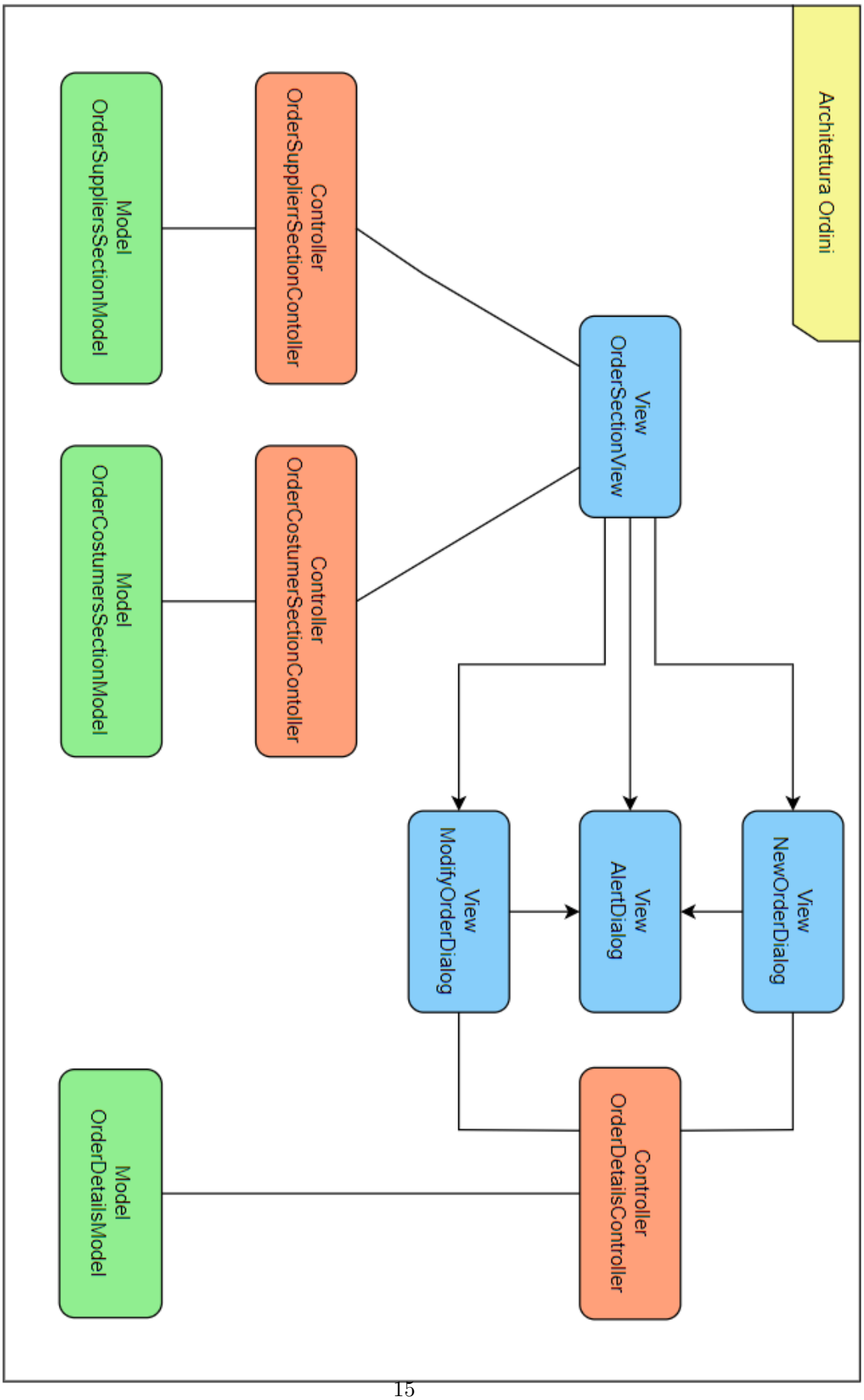


Figura 2.3: Architettura dell'Area relativa agli Ordini

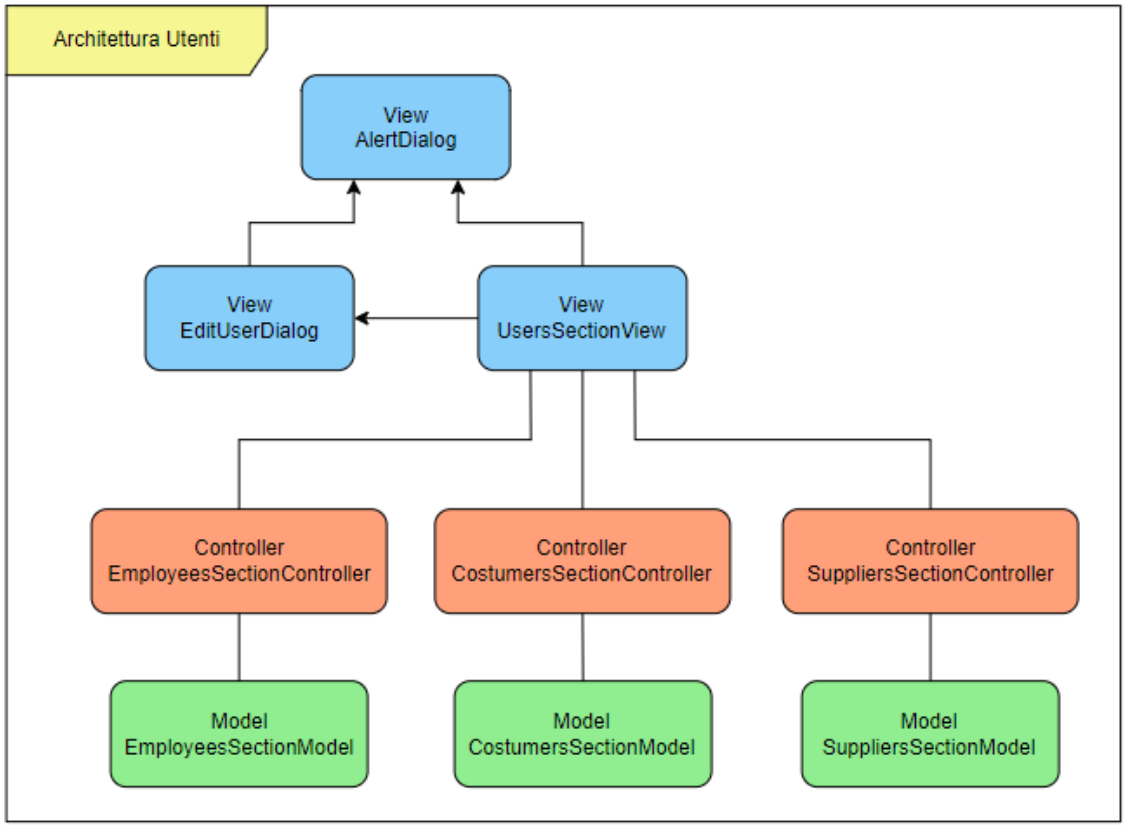


Figura 2.4: Architettura dell'Area relativa agli Utenti

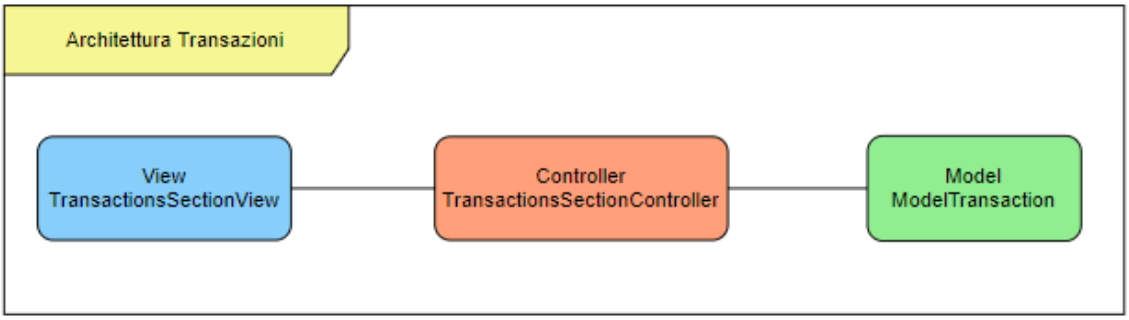


Figura 2.5: Architettura dell'Area relativa alle Transazioni

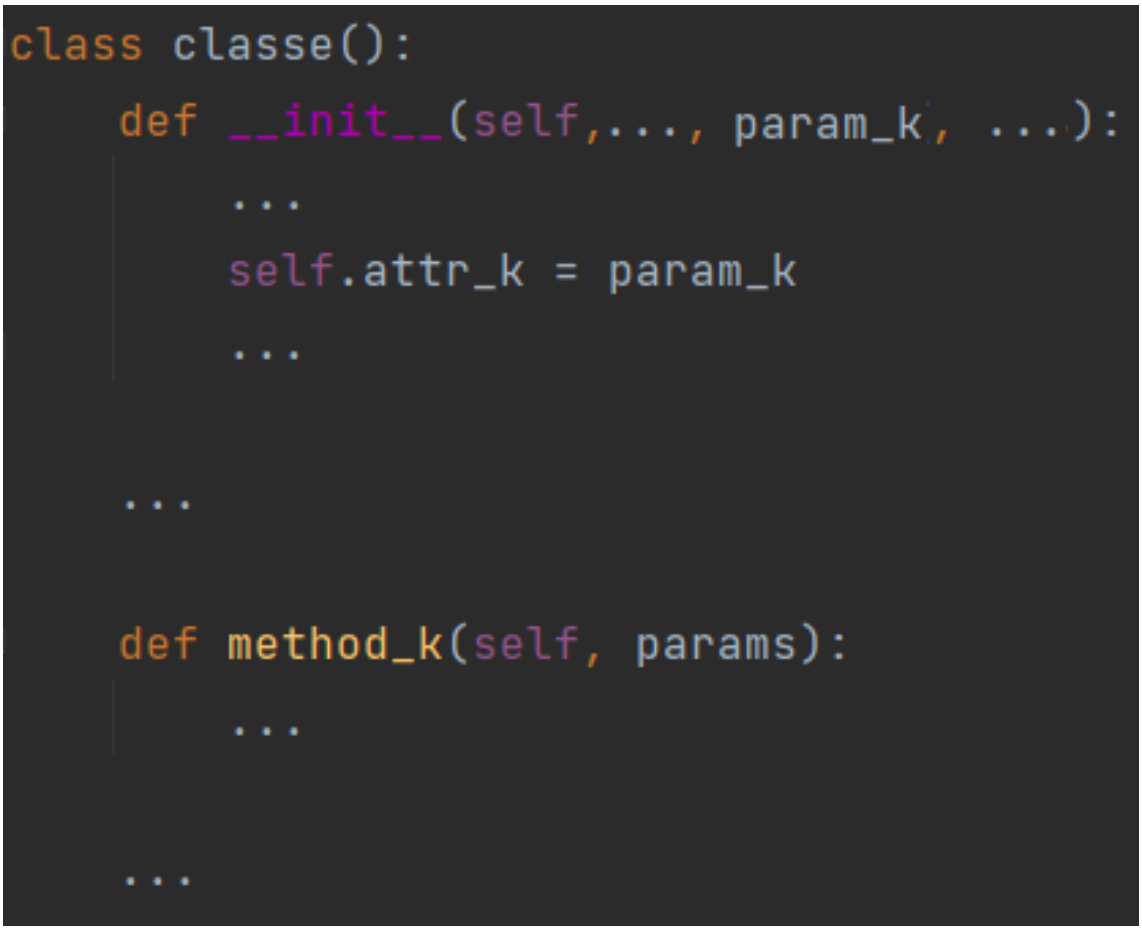


Figura 2.6: Rappresentazione di una generica classe, essa è un'astrazione di un concetto. Al proprio interno definisce il suo stato in un determinato istante, memorizzandolo nelle variabili, e il comportamento, ovvero i metodi, dell'entità rappresentata

Una relazione, invece, riguarda il legame logico tra due o più oggetti, essi sono le istanze, ovvero le espressioni concrete, dei modelli precedentemente introdotti.

2.2.1 Interfaccia Principale

Nella Figura 2.7 viene descritta la struttura generale dell'interfaccia, la classe MainGui si occupa di gestire la funzione di login e di creare la finestra principale dell'applicazione,
Il metodo inizializza gli oggetti delle classi *ProductsSectionView*, *OrdersSectionView*, *UsersSectionView* e *TransactionsSectionView*.
In seguito, ad ognuna delle viste create attribuisce delle funzioni aggiuntivi, questi servono a realizzare la comunicazione tra aree diverse.
Un esempio è dato dall'utilizzo dalla funzione reflect method: la vista della sezione degli ordini, attraverso questo metodo, può avvisare quella relativa alla gestione dei prodotti che un determinato prodotto è stato ordinato, in questo modo la classe *ProductsSectionView* potrà far eseguire al proprio controller le azioni necessarie per quel tipo di evento.
Infine, MainGui chiama la funzione setup ui passando ad essa le 4 view precedentemente istanziate.
Questa seconda funzione ha il compito di strutturare gli aspetti grafici; essa crea il layout principale collocando al suo interno gli oggetti accettati come parametri in fase di definizione.
In ciascuna delle sottoviste assumono un ruolo fondamentale i metodi *SetupSection* e *InitSection*.

Il primo si occupa della creazione dei layout e dei widget, il secondo, invece, inizializza le variabili della classe, istanzia il controller e chiama il metodo per definire gli eventi generati dall'interazione con gli attori.
Le altre funzioni gestiscono la comunicazione con i rispettivi controller e, di conseguenza, con i modelli di dati associati.

2.2.2 Sezione Prodotti

Il diagramma delle classi relativo ai prodotti, rappresentato nelle Figure 2.8 e 2.9, descrive la logica dell'omonima sezione.
Le classi *ProductSectionView*, *ProductSectionController* e *ProductSectionModel* definiscono gli aspetti logici e grafici riguardanti la gestione della lista dei prodotti.
Il compito di definire ed implementare le operazioni su un singolo elemento, invece, è affidata alle classi *EditProdView*, *EditProdController* e *EditProdModel*.
Dividire la gestione dei dati, relativi ai prodotti, tra due gruppi di MVC ha come vantaggio quello di ottenere una maggiore distribuzione del codice, il quale, di conseguenza, risulta più facile da controllare, riutilizzare ed, eventualmente, modificare

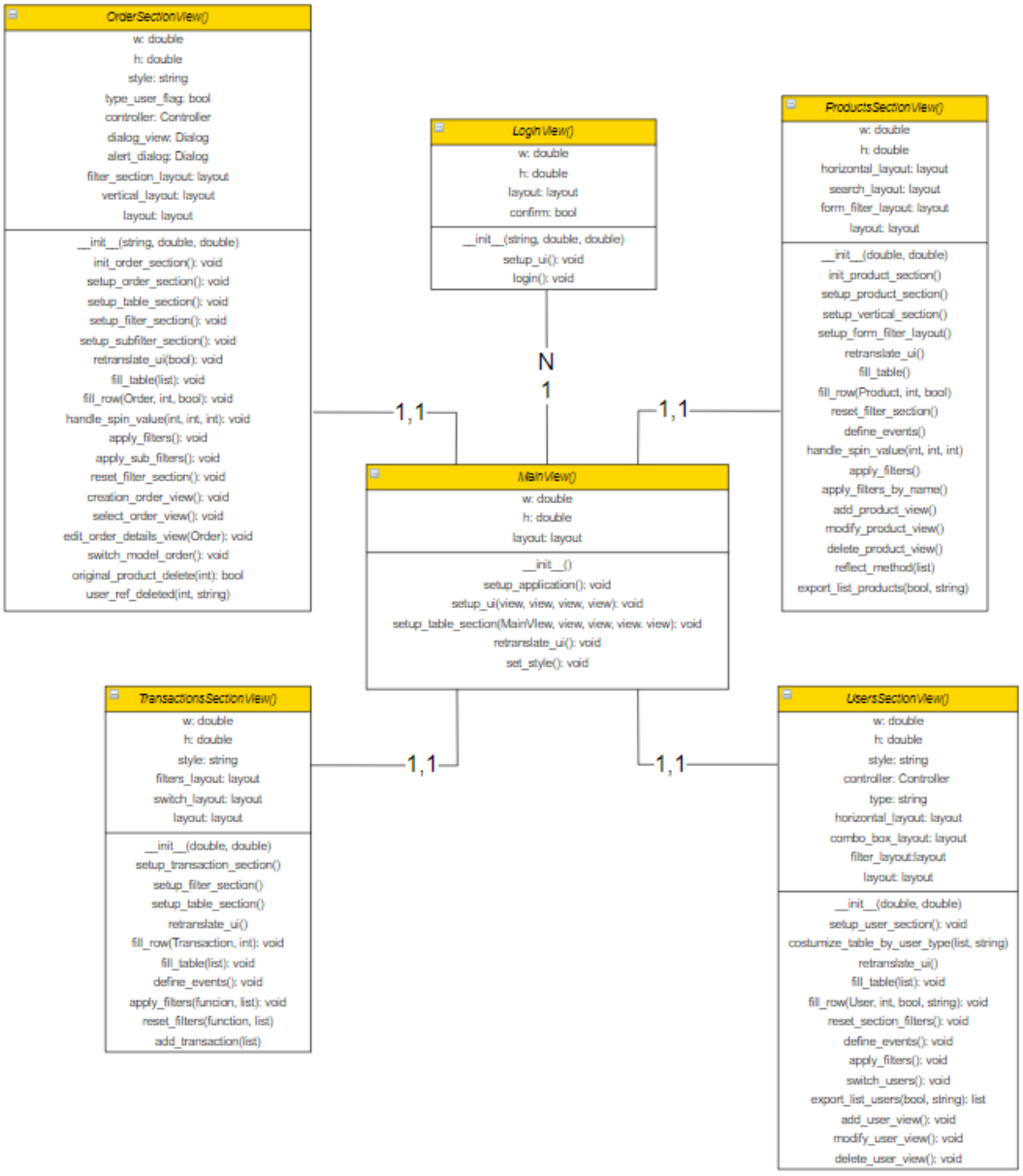


Figura 2.7: Diagramma delle Classi relativo all'Interfaccia Principale

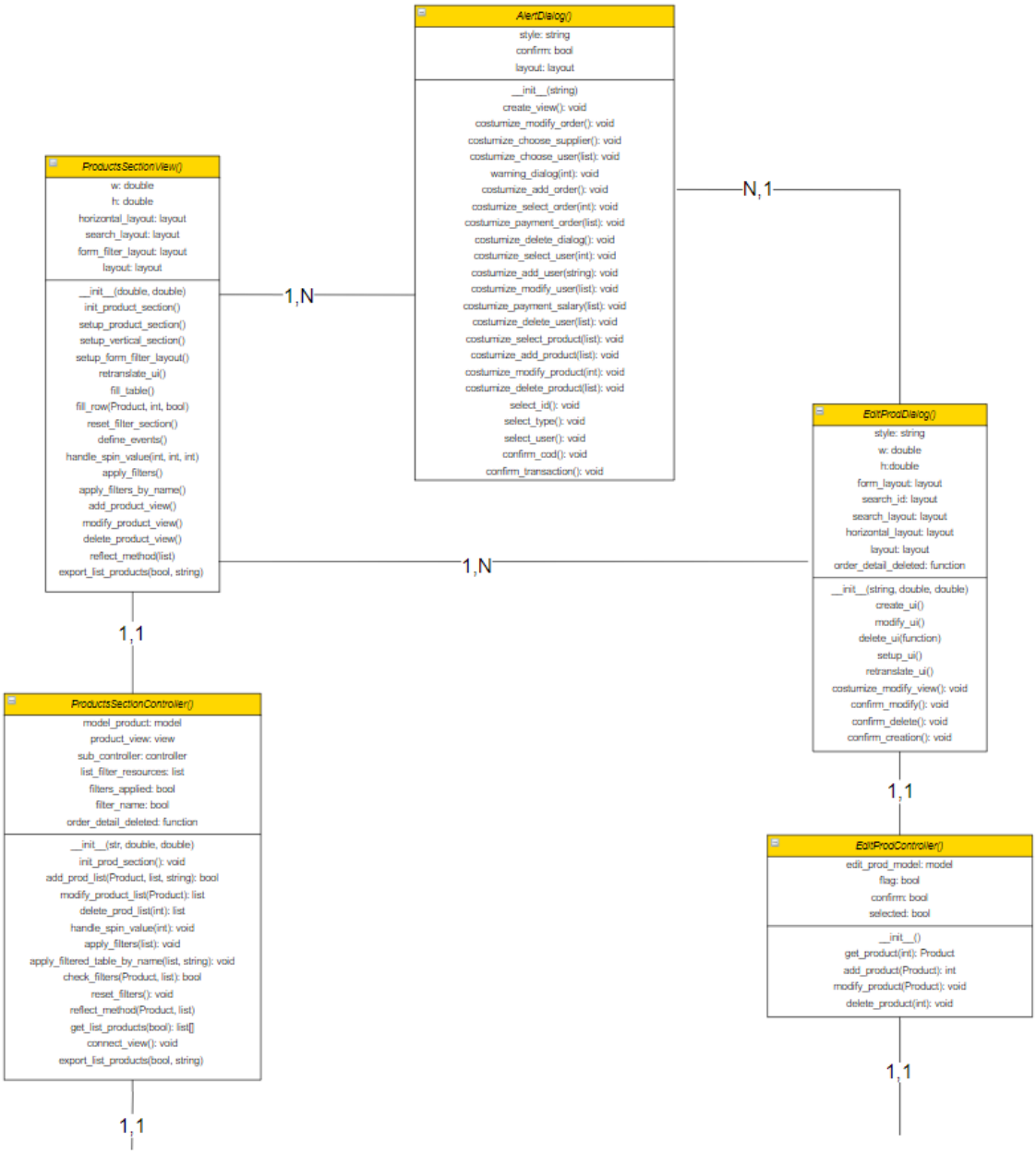


Figura 2.8: Diagramma delle Classi relativo all'area Prodotti

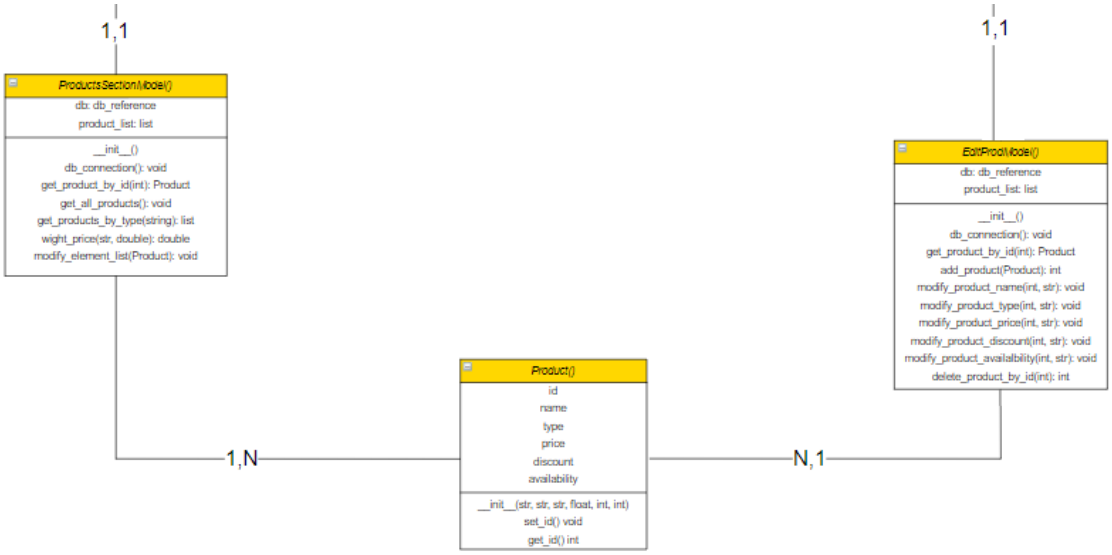


Figura 2.9: Diagramma delle Classi relativo all'area Prodotti

2.2.3 Sezione Ordini

L'area relativa alla gestione agli ordini è, sicuramente, la più complessa tra le quattro che compongono l'architettura.
Le classi descritte nelle Figure 2.12 e 2.13 dovranno occuparsi sia degli ordini dei clienti che di queli fornitori, inoltre dovranno gestire i dettagli relativi a ciascuno di essi.
Per quanto riguarda la gestione degli ordini (ci si riferisce alla Figura 2.11), la vista *InitSection*, deve rappresentare le informazioni di due classi di dati (gli ordini dei clienti e quelli dei fornitori) concettualmente diverse, le quali, tuttavia, necessitano di funzionalità analoghe.
La soluzione consiste nel connettere la View in questione con due diversi controller e, di conseguenza, con altrettanti modelli di dati.
Questa funzionalità è realizzata attraverso il metodo switch order controller: esso cambia il valore dell'attributo controller, assegnandogli, a seconda della scelta dell'utente, un'istanza della classe *OrderCostumersSectionController* oppure di *OrderSuppliersSectionController*.
Inoltre, la vista contiene l'attributo type user flag, che, a seconda del valore assunto (vero o falso), indica se si sta lavorando con gli ordini dei clienti oppure con quelli dei fornitori.
Le classi adibite alla gestione dei dati relativi ai dettagli degli ordini (ci si riferisce alla Figura 2.12), sono state organizzate seguendo una logica opposta rispetto alle precedenti.
Le viste *NewOrderDialog* e *ModifyOrderDialog*, sebbene rappresentino la stessa tipologia di dati, necessitano di funzionalità diverse, esse, inoltre, pur essendo

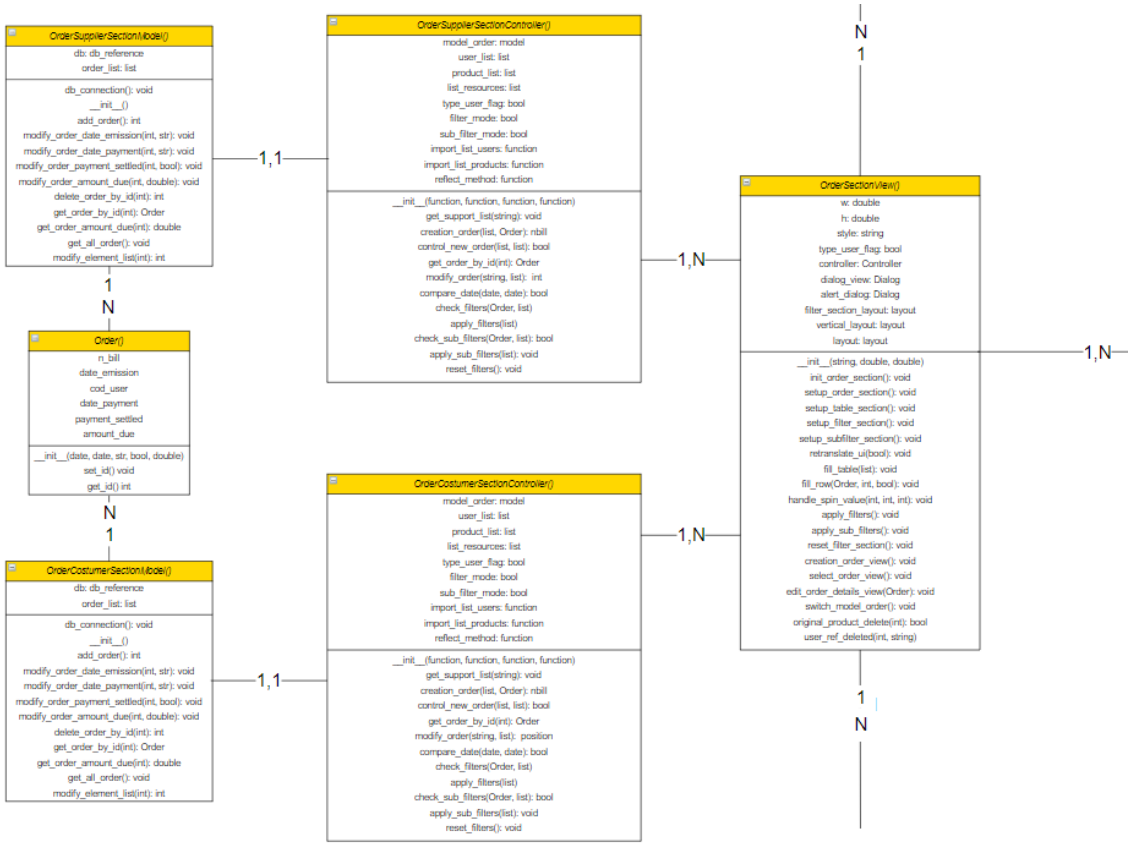


Figura 2.10: Diagramma delle Classi relativo all'area Ordini (ordini clienti e fornitori)

entrambe instanziate dai metodi all'interno di *OrdersSectionView*, non possono essere mandate in esecuzione contemporaneamente.
Quest'ultimo aspetto ci assicura che le due classi non possano richiedere simultaneamente la lettura o la scrittura degli stessi dati.
Entrambe le viste, dunque, si relazioneranno con il controller *OrderDetailsController*.

2.2.4 Sezione Utenti

Questa sezione contiene le informazioni relative ad utenti, dipendenti e fornitori.
La logica organizzativa, descritta nella Figura 2.14, è simile a quella della sezione degli ordini.
La view *UserSectionView* si relaziona con tre diversi controller: *CostumersSectionController*, *EmployeesSectionController* e *SuppliersSectionController*, ciascuno dei quali gestisce il modello relativo alla propria tipologia di utenti.
A seconda delle esigenze del titolare, il controllo degli input e dei dati sarà affidato ad una di queste tre classi.

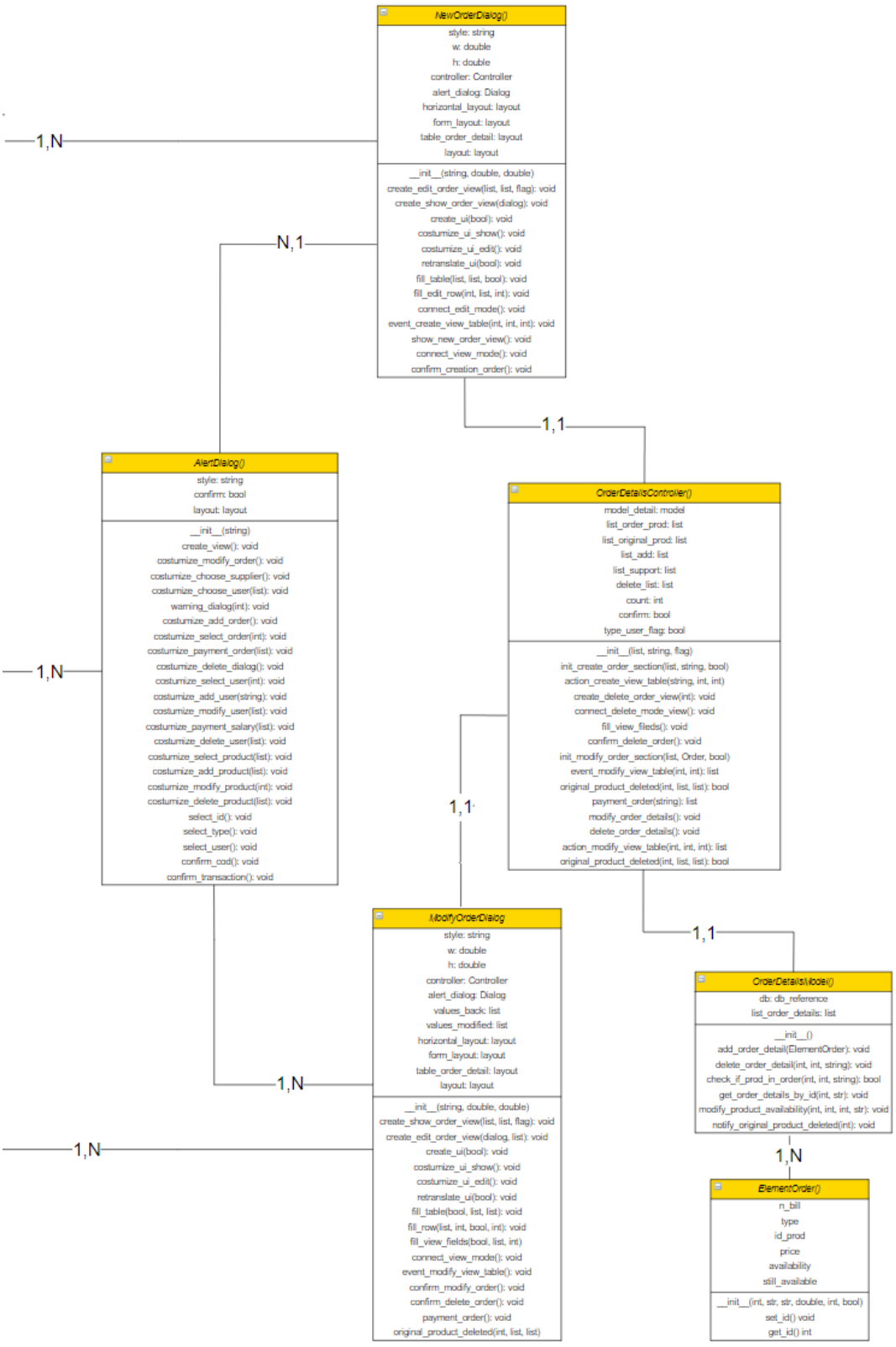


Figura 2.11: Diagramma delle Classi relativo all'area Ordini (dettagli ordine)

Sebbene le operazioni siano molto simili a quelle eseguite sui prodotti, la gestione dei singoli utenti e delle liste degli stessi verrà affidata ad un'unica combinazione di model, view e controller.
Per questo motivo, la vista *EditUserDialog*, dove sarà possibile effettuare le operazioni sugli utenti, invierà gli input ricevuti a *UserSectionView* e di conseguenza al controller e al model che rappresentano quella determinata tipologia.

2.2.5 Sezione Transazioni

L'ultima sezione, relativa al registro dei pagamenti, è quella che presenta la struttura più semplice. L'architettura, mostrata nella Figura 2.14, è formata da soli 3 componenti: *TransactionsSectionView*, *TransactionsSectionController*, *TransactionsSectionModel*

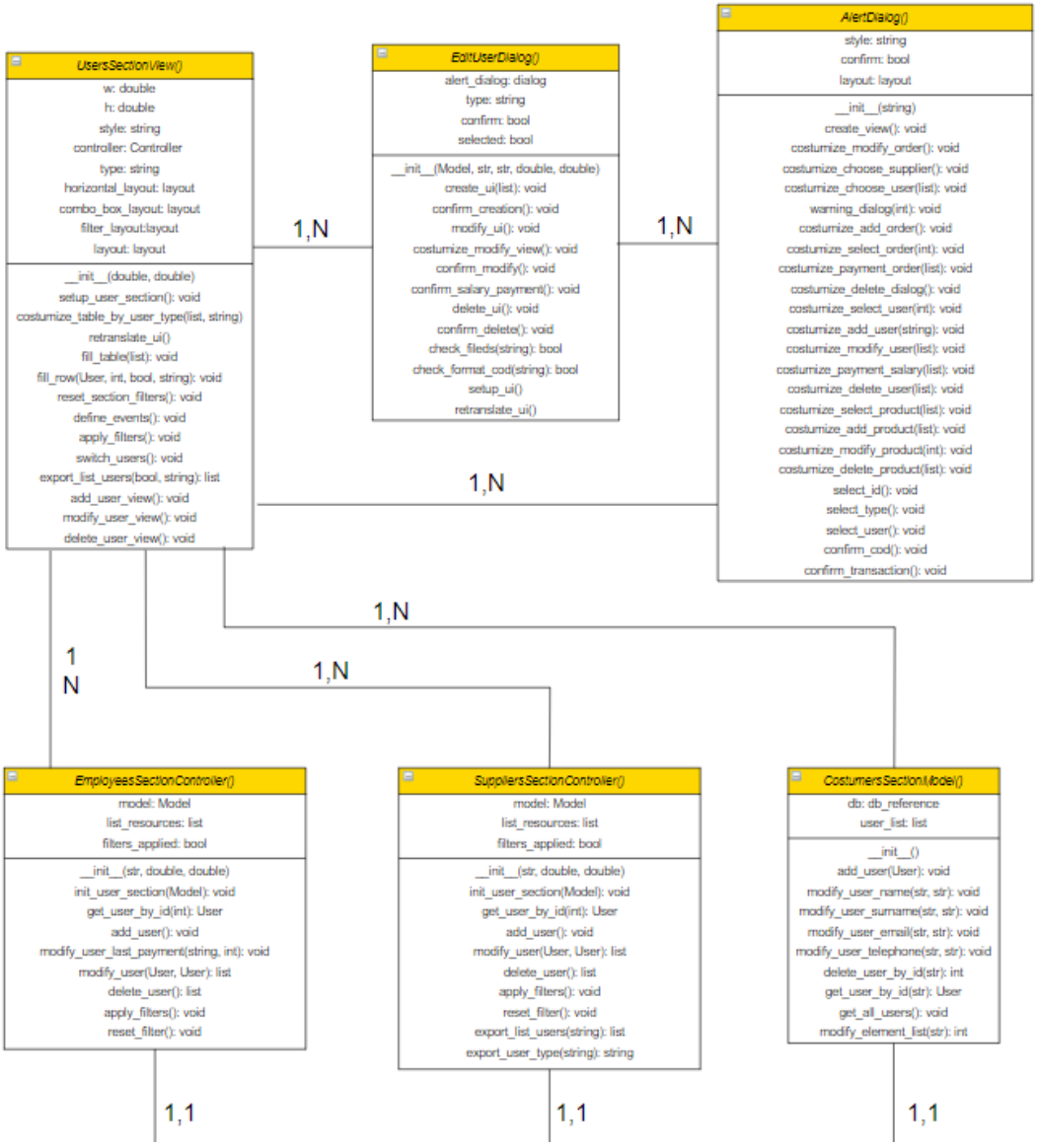


Figura 2.12: Diagramma delle Classi relativo all'area Utenti

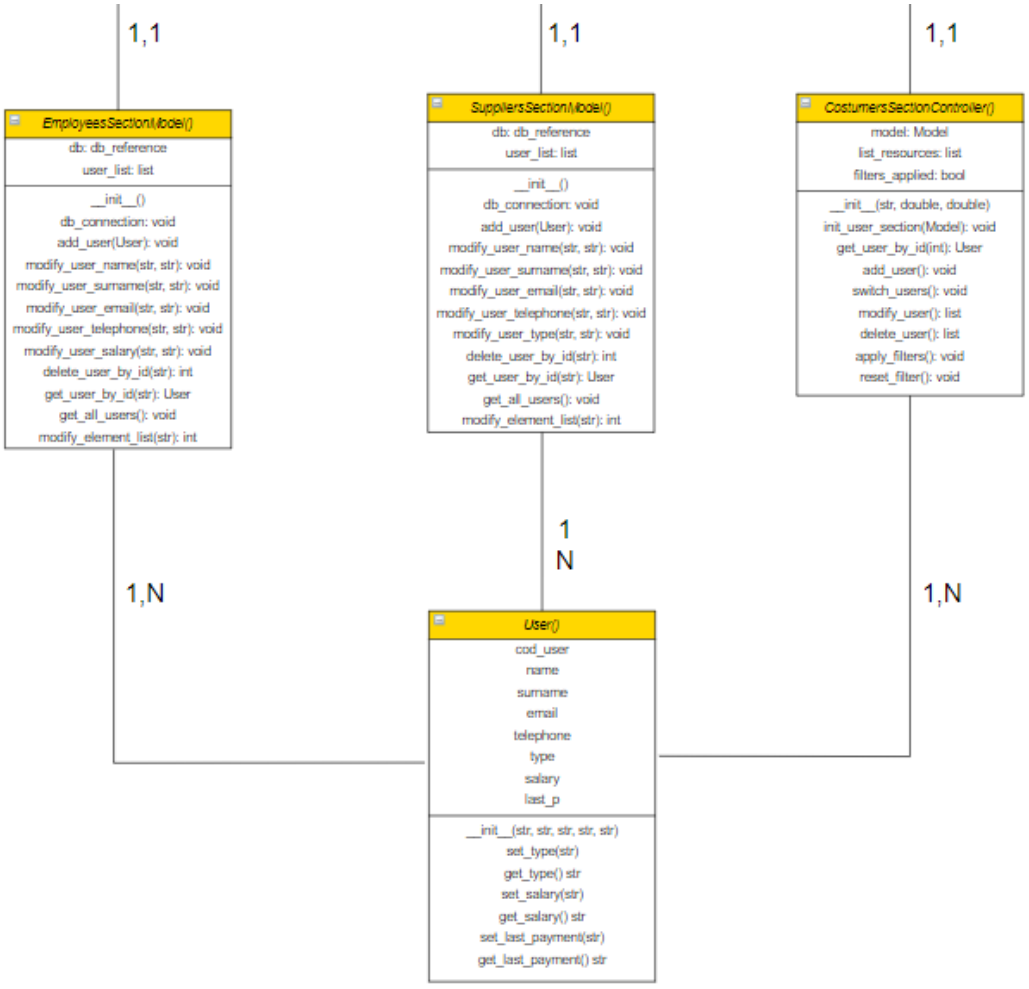


Figura 2.13: Diagramma delle Classi relativo all'area Utenti

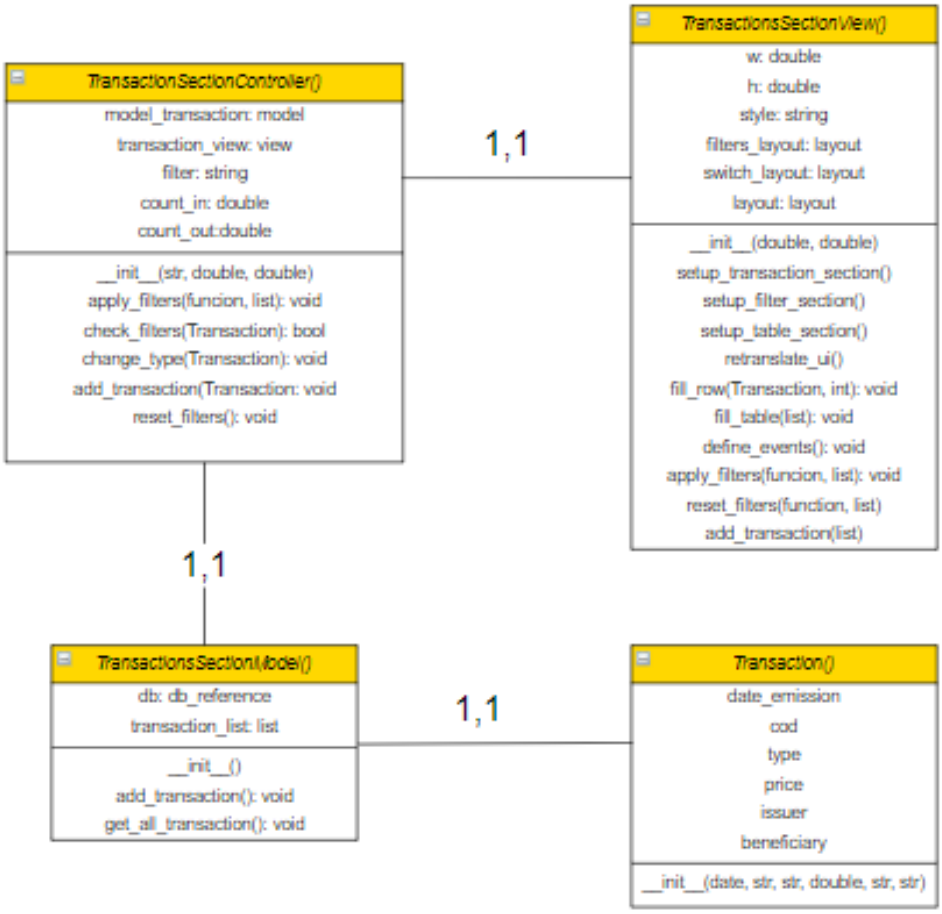


Figura 2.14: Diagramma delle Classi relativo all'area Transazioni

Capitolo 3

Progettazione delle applicazioni e dell'interfaccia

Nel precedente capitolo è stata acquisita una conoscenza dell'architettura in termini di unità di realizzazione, sono stati descritti i vari componenti del sistema, identificando, per ciascuno, la struttura e le relazioni con gli altri elementi. La fase successiva, analizzata nel presente capitolo, dovrà fornire una descrizione in termini di esecuzione. Si mostrerà quali dovranno essere i vari scenari in cui, a seconda degli input del titolare, si potrà trovare il sistema, quali output dovranno essere mostrati dal programma e qual è la logica che dovrà essere utilizzata in fase di implementazione.

3.1 Soluzioni Progettuali

Progettazione delle Interfacce

La realizzazione del front end del programma ha come obiettivo principale quello di implementare un'interfaccia intutiva e ben organizzata.

I servizi offerti e le informazioni mostrate dovranno essere ben distribuite all'interno dell'interfaccia; in questo modo, il titolare potrà facilmente accedere ai dati di interesse.

Sarà possibile, attraverso una barra di selezione, navigare tra quattro schermate, ciascuna delle quali mostra, mediante l'utilizzo di una tabella, le informazioni realtive ad una specifica area dell'attività.

Per quanto riguarda le varie operazioni sui dati, queste dovranno essere eseguite su delle finestre aggiuntive. Queste dovranno essere visibili solo su richiesta e , una volta mandate in esecuzione, eseguire le proprie funzionalita in maniera totalmente autonoma rispetto all'interfaccia principale.

Progettazione Comportamentale del software

I diagrammi utilizzati, nelle sezioni successive, forniranno una vista più dettagliata sulle soluzioni realizzative delle varie funzionalità che dovranno essere implementate.

Il primo diagramma è quello delle attività; esso descrive la sequenza di azioni che devono essere realizzate all'interno di un determinato scenario. Il flusso, tra i

diversi stati del sistema, è rappresentato tramite delle frecce orientate, che indicano la sequenza temporale con cui devono essere effettuate le azioni di transizione.

Il diagramma delle sequenze, invece, analizza la relazioni che intercorrono, in termini di scambio di messaggi, tra attori e oggetti del sistema in un derterminato scenario. Per ogni elemento viene indicato, in termini di realizzazione della funzionalità, il suo ciclo di esecuzione e quali sono i dati elaborati, ricevuti e inviati.

3.2 Progettazione della sezione relativa ai Prodotti

L'area relativa ai prodotti sarà strutturata come è mostrato nella Figura 3.1.

La descrizione si può effettuare individuando le tre macrosezioni in cui è suddivisa la schermata:

- *Sezione Filtri*: si trova nella parte superiore dell'interfaccia; essa contiene sia un'area per impostare, mediante dei campi di inserimento, i parametri relativi a prezzo, quantità e sconto, sia una barra di ricerca, dove si potrà digitare il nome del prodotto desiderato.
- *Tabella Prodotti*: disposta nella zona centrale, essa mostrerà le informazioni relative ai prodotti correntemente visualizzati.
- *Pulsanti degli eventi*: questi sono collocati nella parte inferiore della schermata, essi permettono di accedere alle interfacce dedicate alla gestione dei singoli prodotti.

Nelle successive sezioni si analizzeranno le funzionalità accessibili all'interno dell'interfaccia, descrivendo, per ciascuna di esse, le caratteristiche grafiche, la sequenza di azioni associate e la logica implementativa.

3.2.1 Ricerca dei Prodotti

Realizzazione grafica e logica degli eventi

Il titolare potrà impostare filtri su prezzo, quantità e sconto, come mostrato nella Figura 3.2, oppure cercare i prodotti in base al loro nome, come evidenziato nella Figura 3.3.

Attraverso gli appositi tasti di conferma verranno applicati i filtri selezionati e, successivamente, a seconda dei risultati della ricerca, la tabella mostrerà i prodotti trovati.

La Figura 3.4 descrive il flusso di attività associato alla funzionalità di ricerca:

- Il titolare imposta dei parametri per il filtraggio e conferma la ricerca.
- Nel caso ci siano prodotti che soddisfano i criteri selezionati, questi vengono visualizzati nella tabella.

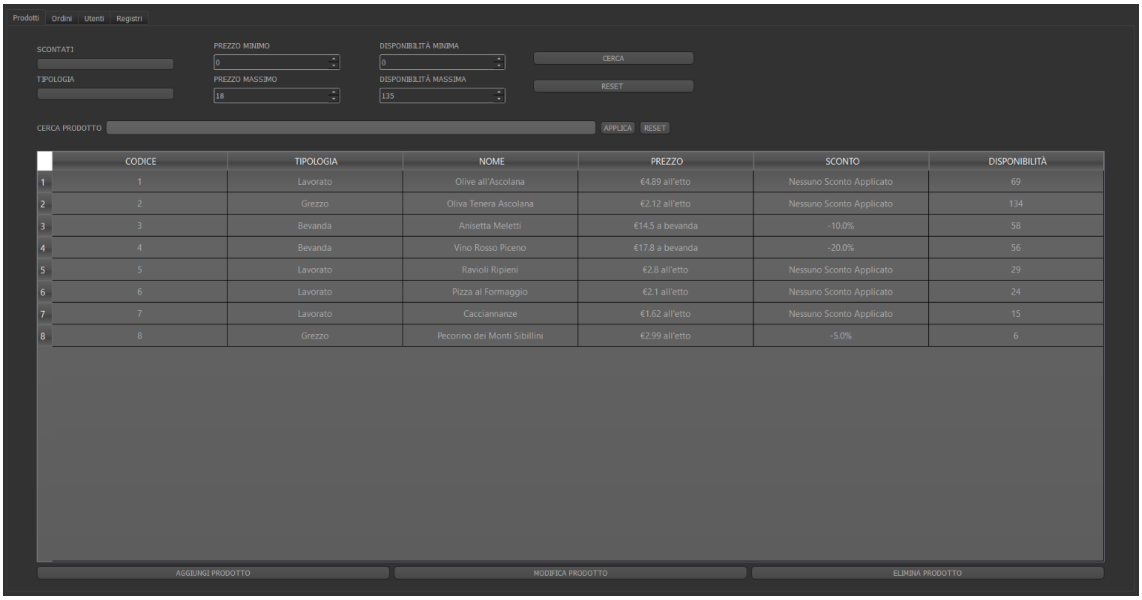


Figura 3.1: Interfaccia della sezione relativa ai Prodotti

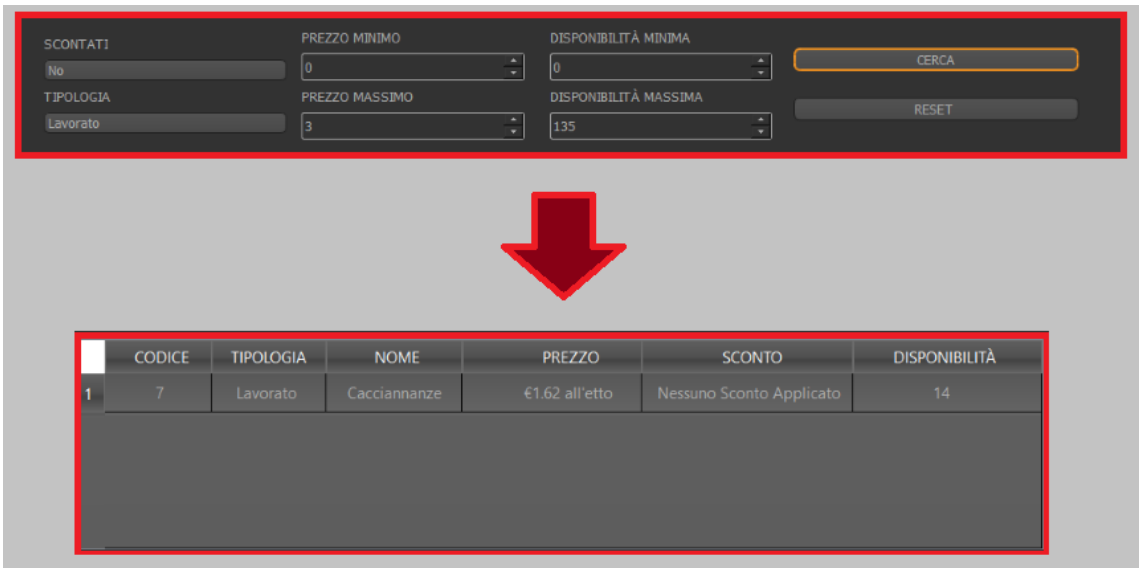


Figura 3.2: Esempio del filtraggio dei prodotti

Figura 3.3: Esempio della ricerca tra i prodotti in base al nome

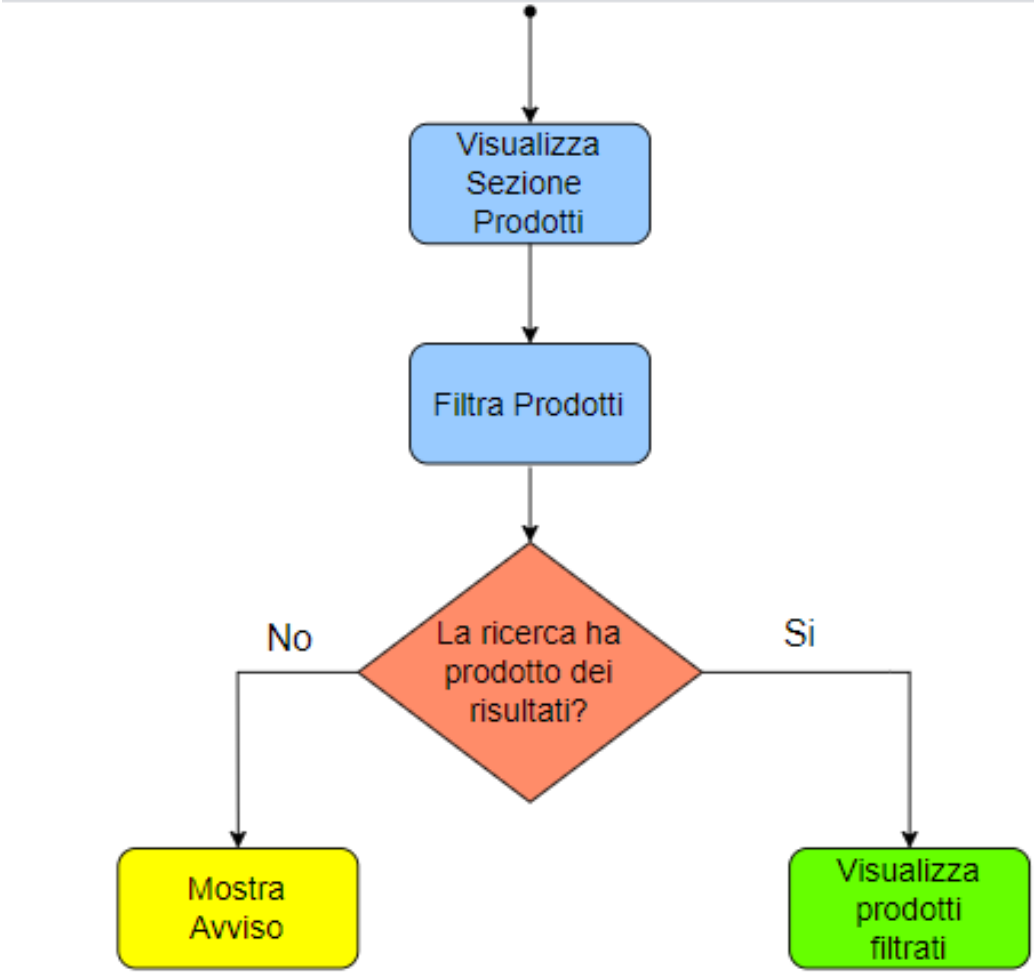


Figura 3.4: Sequenza di azioni relativa al filtraggio della lista prodotti

- Nel caso in cui la ricerca non produca alcun risultato, il sistema avvisa il titolare mediante una finestra di errore.

3.2.2 Aggiunta di un nuovo Prodotto

Logica degli eventi e descrizione grafica

Il diagramma nella Figura 3.5 rappresenta il flusso di attività relativo all'inserimento di un nuovo prodotto nel sistema.

Esso può essere descritto come segue:

- Il titolare, che si trova nell'area Prodotti, seleziona la funzionalità per aggiungere un nuovo prodotto.
- Il sistema visualizza l'interfaccia dedicata all'inserimento delle informazioni relative all'elemento da aggiungere nel sistema.

- Il titolare inserisce, negli appositi campi, i dati relativi al nuovo prodotto e conferma l'operazione.
- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo prodotto dovrà essere aggiunto nel sistema e mostrato nella tabella.

La Figura 3.6 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.7 mostra quali sono le interazioni fra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità.

La classe *ProductsSectionView()* delega le istanze di *EditProdDialog()* e *AlertDialog()*, inizializzate rispettivamente mediante i metodi *CreateUi()* e *CostumizeAddProduct()* alla gestione delle interfacce per la creazione del prodotto e delle finestre di validazione dell'operazione.

Per quanto riguarda la registrazione del nuovo elemento del sistema, essa verrà gestita totalmente da *EditProdController()* e *EditProdModel()*.

Qualora l'operazione di aggiunta vada a buon fine, il nuovo prodotto, associato ad un parametro di conferma, verrà restituito a *ProductSectionView()*.

ProductSectionController() aggiunge il nuovo prodotto alla lista e controlla, attraverso il metodo *CheckFields()*, se quest'ultimo debba essere aggiunto alla tabella, nel caso siano stati applicati dei filtri.

3.2.3 Modifica di un Prodotto presente le sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.8, rappresenta il flusso di attività relativo alla modifica di un prodotto presente nel sistema.

Esso può essere descritto come segue:

- Il titolare, che si trova nell'area Prodotti, seleziona la funzionalità per la modifica di un prodotto presente nel sistema.
- Il sistema visualizza l'interfaccia di modifica dove, inizialmente, verrà richiesto di inserire il codice appartentente al prodotto che si vuole modificare.
- Il sistema, se trova un elemento associato al codice inserito, valorizza i campi dell'interfaccia con le informazioni risultanti.
- Il titolare, una volta modificati i dati, conferma l'operazione.

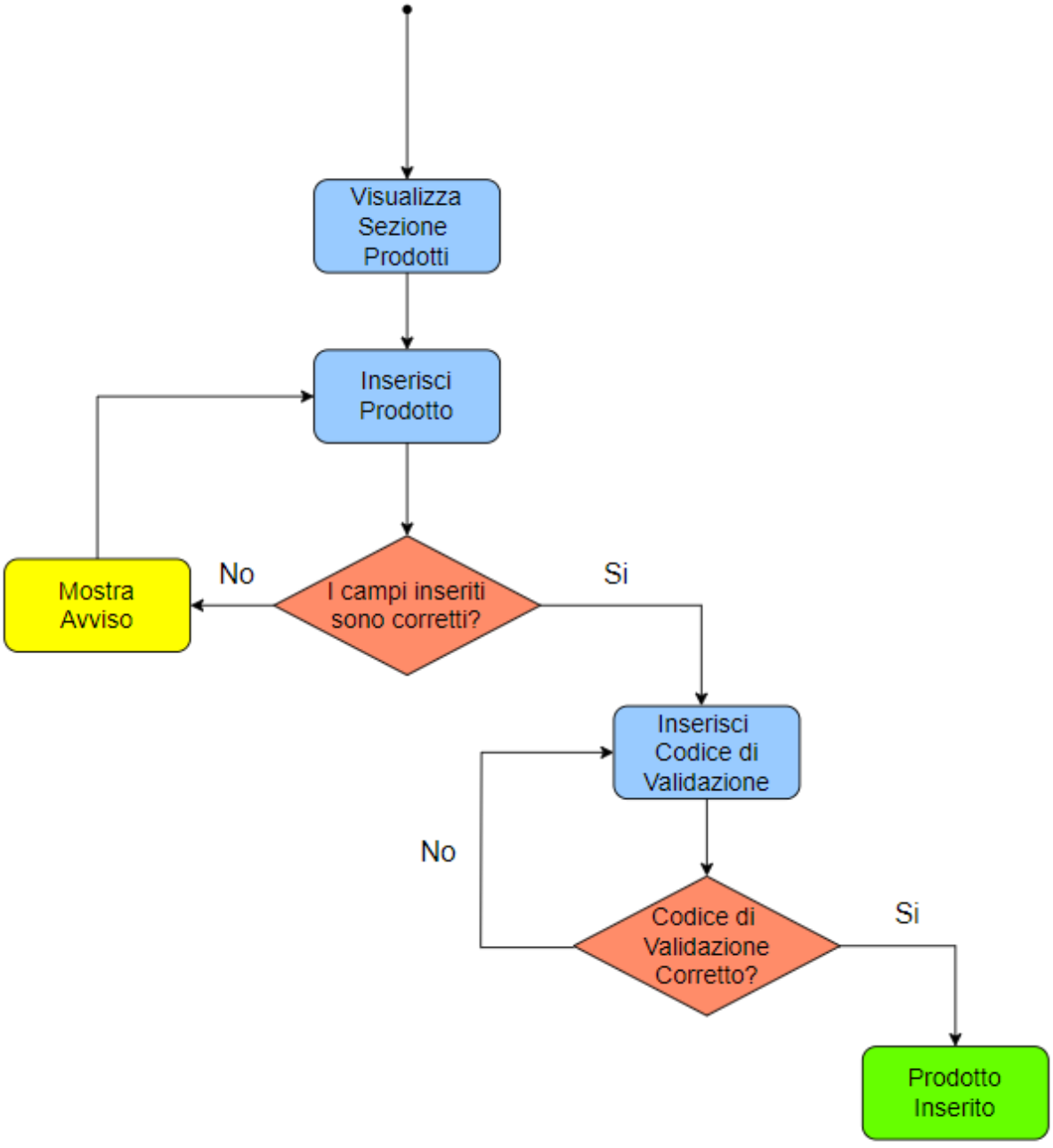


Figura 3.5: Sequenza di azioni relativa all'aggiunta di un nuovo prodotto

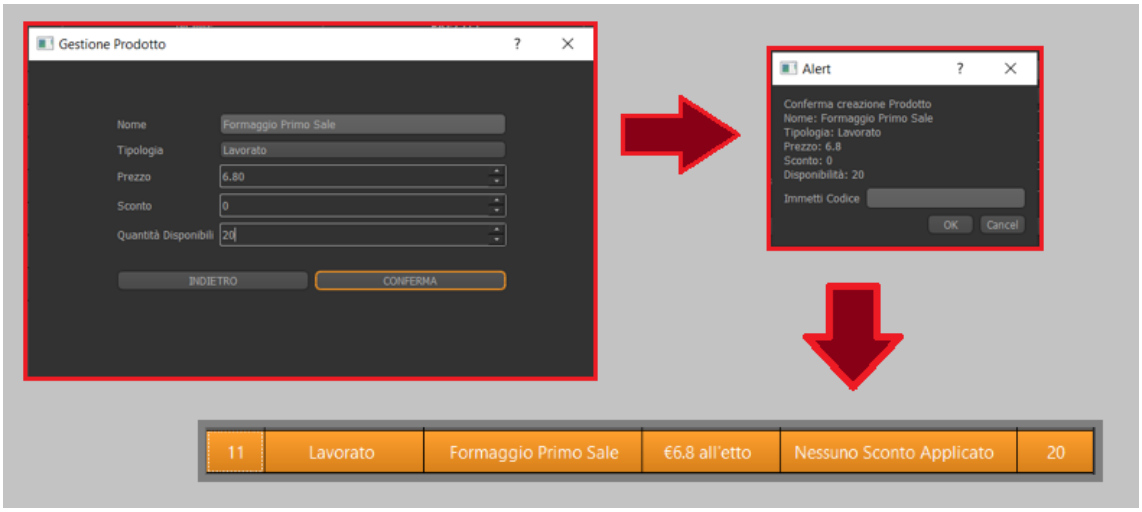


Figura 3.6: Creazione di un nuovo prodotto

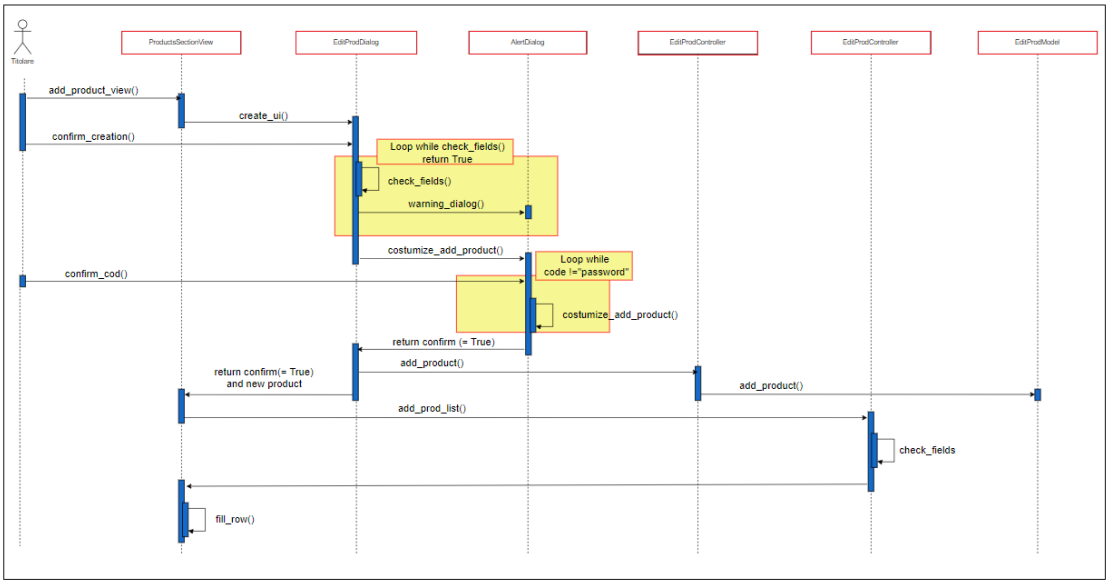


Figura 3.7: Diagramma delle sequenze relativo all'aggiunta di un nuovo Prodotto

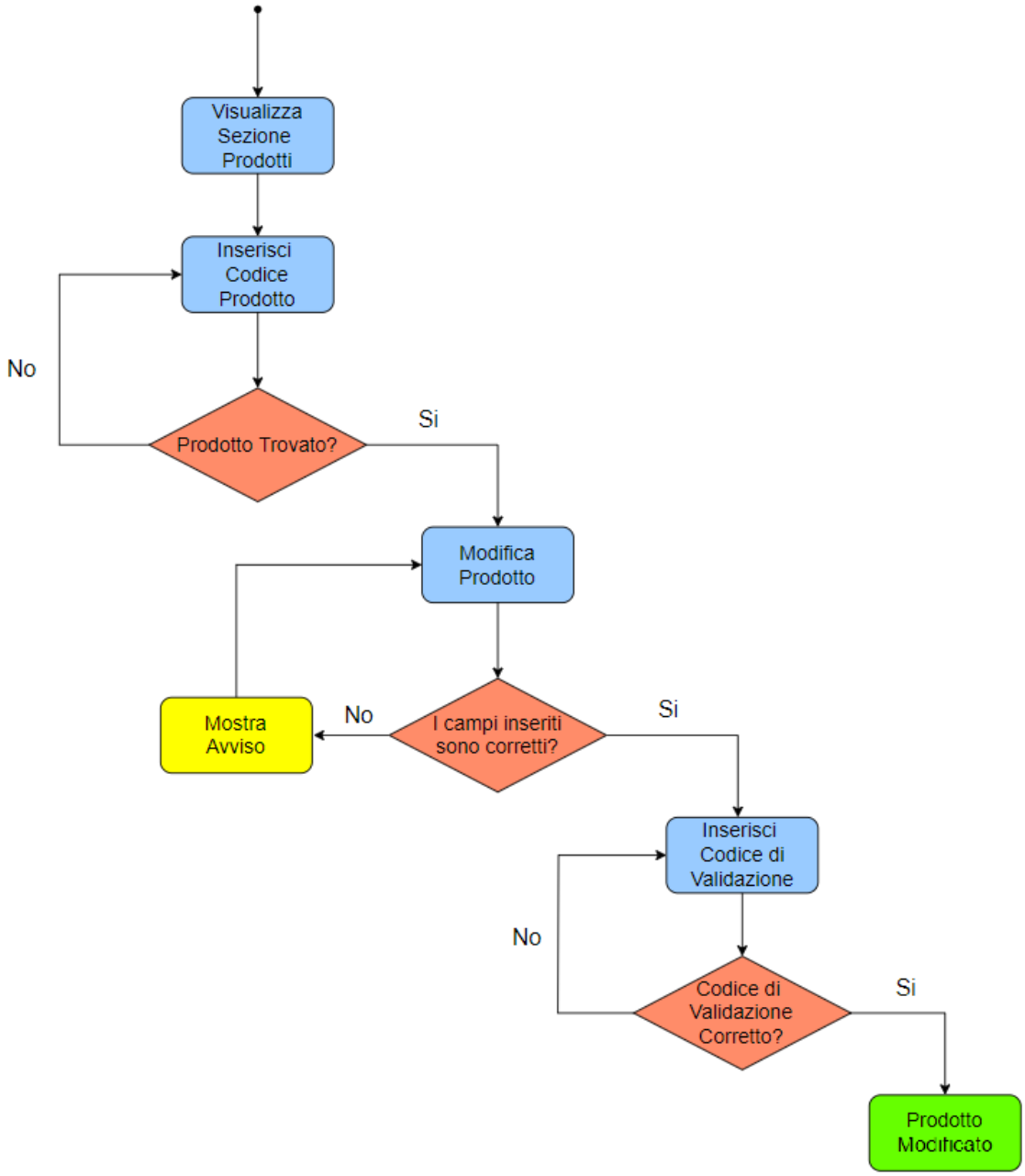


Figura 3.8: Sequenza di azioni relativa alla modifica di un prodotto

- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo prodotto dovrà essere aggiornato nel sistema e mostrato correttamente nella tabella.

La Figura 3.9 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

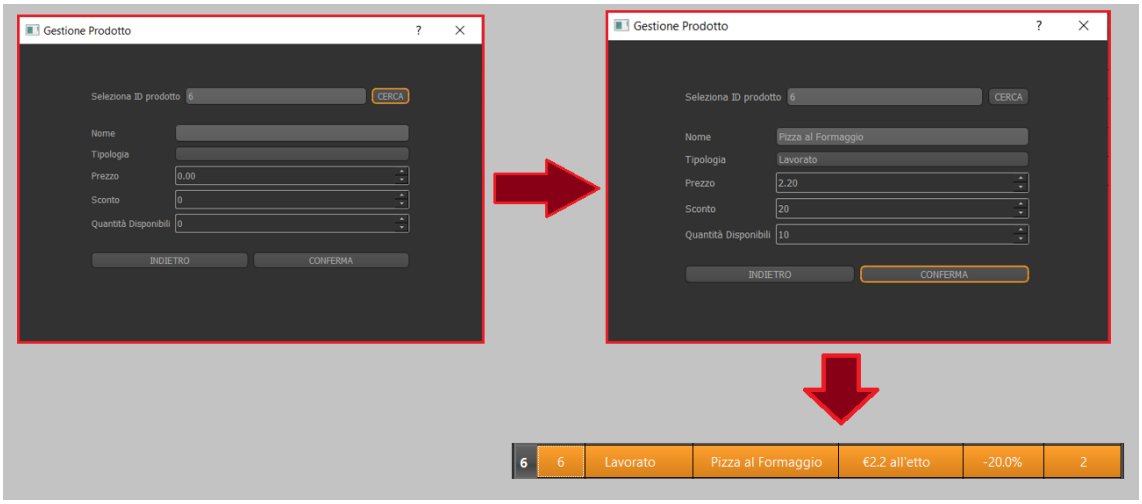


Figura 3.9: Modifica di un Prodotto

Diagrammi delle sequenze associati

La Figura 3.10 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità.

Gli input dell'utente, nel verificarsi del presente scenario, sono accettati da un'istanza di *EditProdDialog()*, inizializzata mediante il metodo *ModifyUi()*.

EditProdDialog() e *EditProdController()* si occupano, invece, di recuperare le informazioni del prodotto da modificare e, successivamente, dell'aggiornamento nel sistema delle stesse.

Le interfacce di errore e di validazione saranno mandate in esecuzione richiamando le funzionalità all'interno di *AlertDialog()*.

Infine, le istanze di *ProductSectionView()* e *ProductSectionController()*, dopo aver ricevuto le informazioni relative all'esito dell'operazione svolta, applicano i controlli per stabilire se sia necessario un aggiornamento dell'interfaccia.

3.2.4 Eliminazione di un prodotto presente le sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.11, rappresenta il flusso di attività relativo all'eliminazione di un prodotto presente nel sistema; esso può essere descritto spiegato di seguito:

- Il titolare, che si trova nell'area relativa ai prodotti, seleziona la funzionalità per l'eliminazione di un prodotto presente nel sistema.
- Il sistema visualizza un'interfaccia dove verrà richiesto di inserire il codice appartenente al prodotto che si vuole eliminare.
- Il titolare conferma l'eliminazione.

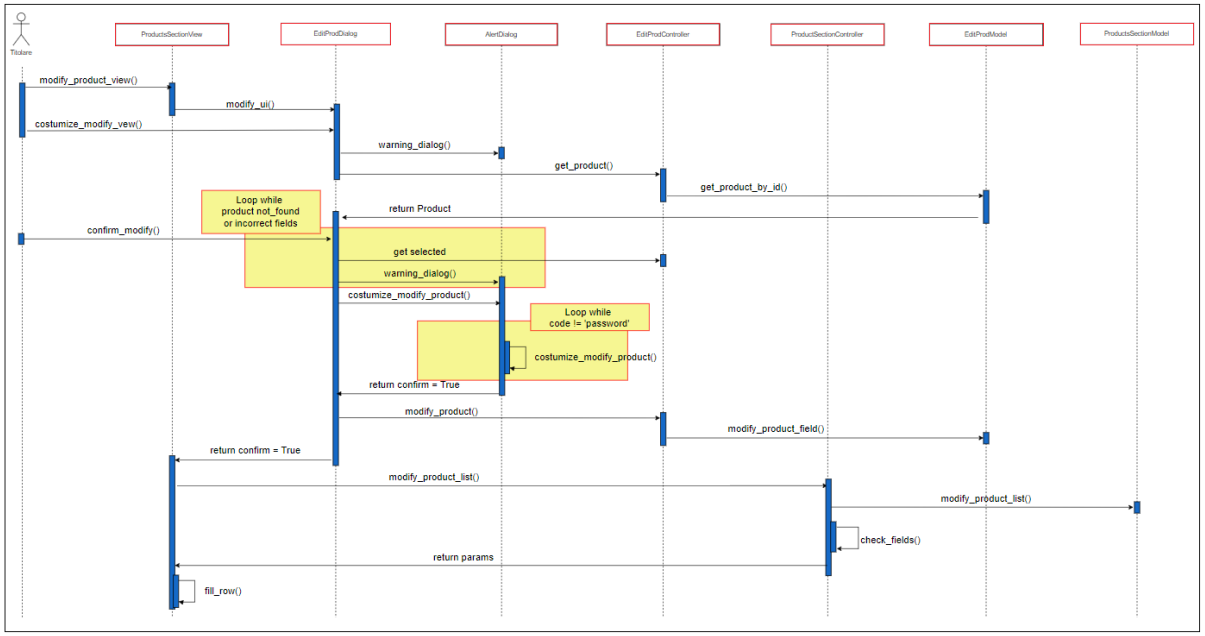


Figura 3.10: Diagramma delle sequenze relativo alla modifca di un Prodotto

- Il sistema, se trova un elemento associato al codice inserito, visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo prodotto dovrà essere eliminato dal sistema e rimosso correttamente nella tabella.

La Figura 3.12 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.13 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella funzionalità di eliminazione di un prodotto. I ruoli svolti dalle istanze, nella realizzazione di questo scenario, sono analoghi a quelli descritti nel caso dell'aggiunta e della modifica di un prodotto.

L'unica differenza, al livello di logica sequenziale, è il controllo eseguito mediante il metodo *OriginalProductDeleted()*. Esso verifica se il prodotto sia associato ad almeno un ordine non saldato. Qualora venga verificata la precedente condizione, non sarà possibile eseguire la cancellazione

3.3 Progettazione della sezione relativa agli ordini

L'area dedicata alla gestione degli ordini sarà organizzata come descritto nella Figura 3.14.

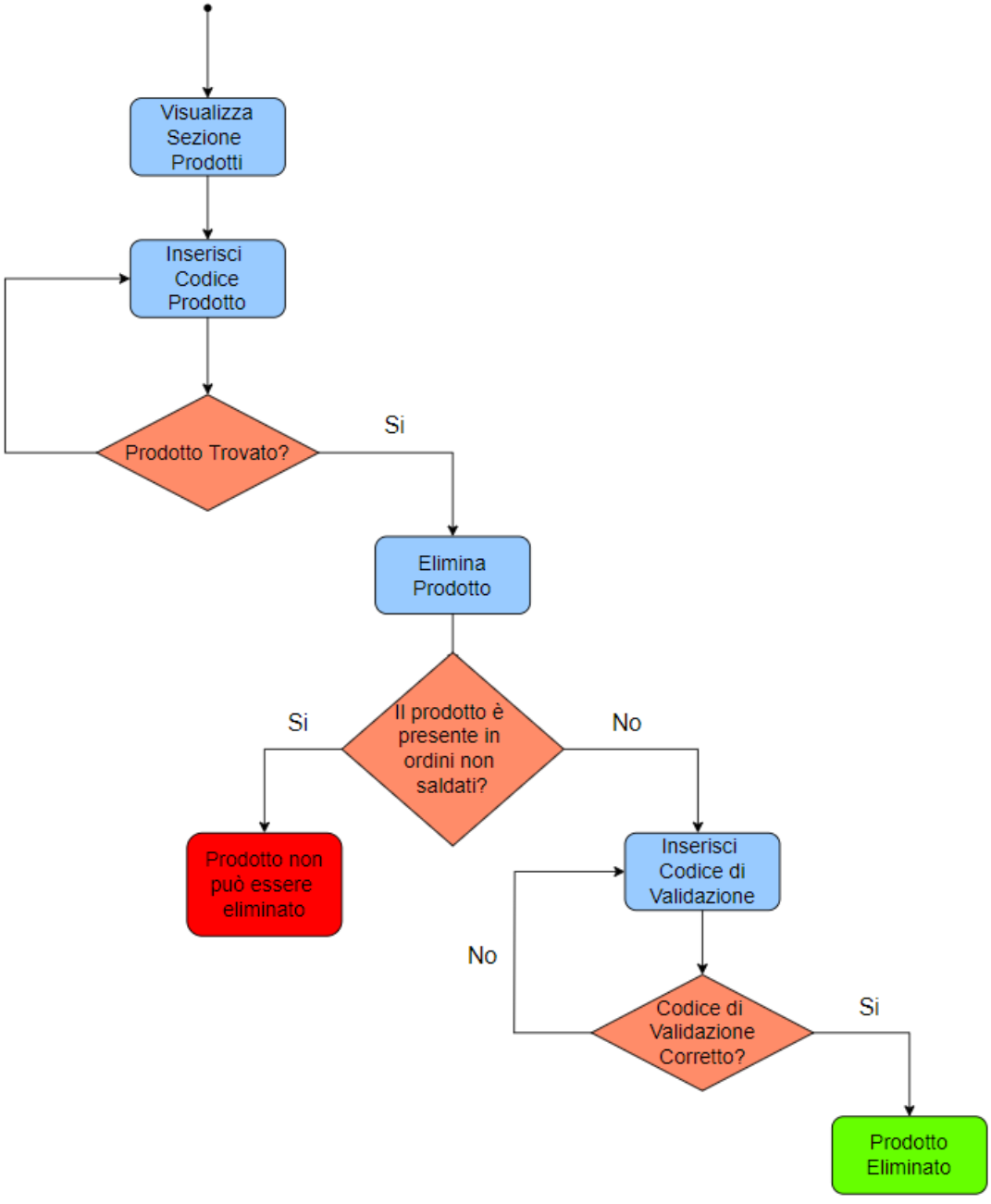


Figura 3.11: Sequenza di azioni relativa all'eliminazione di un prodotto

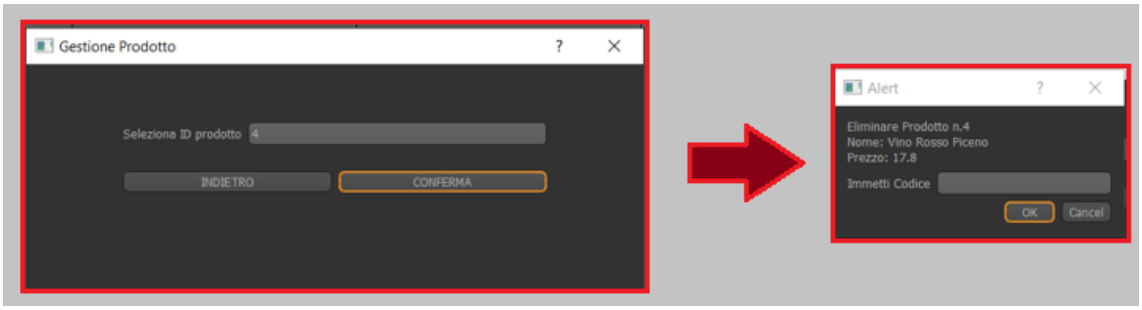


Figura 3.12: Eliminazione di un Prodotto

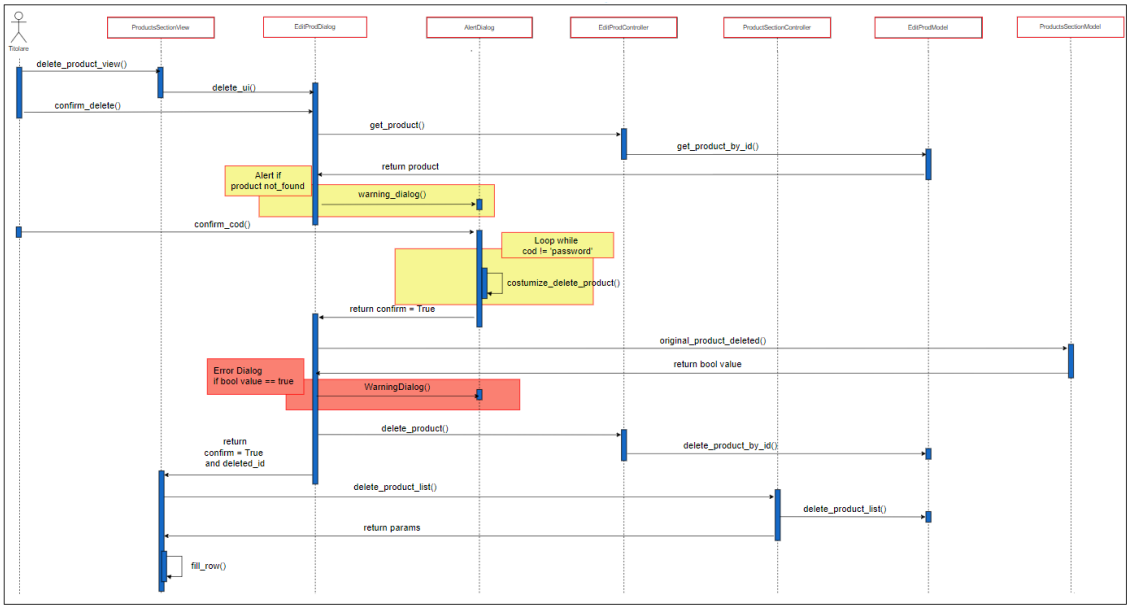


Figura 3.13: Diagramma delle sequenze relativo all'eliminazione di un Prodotto

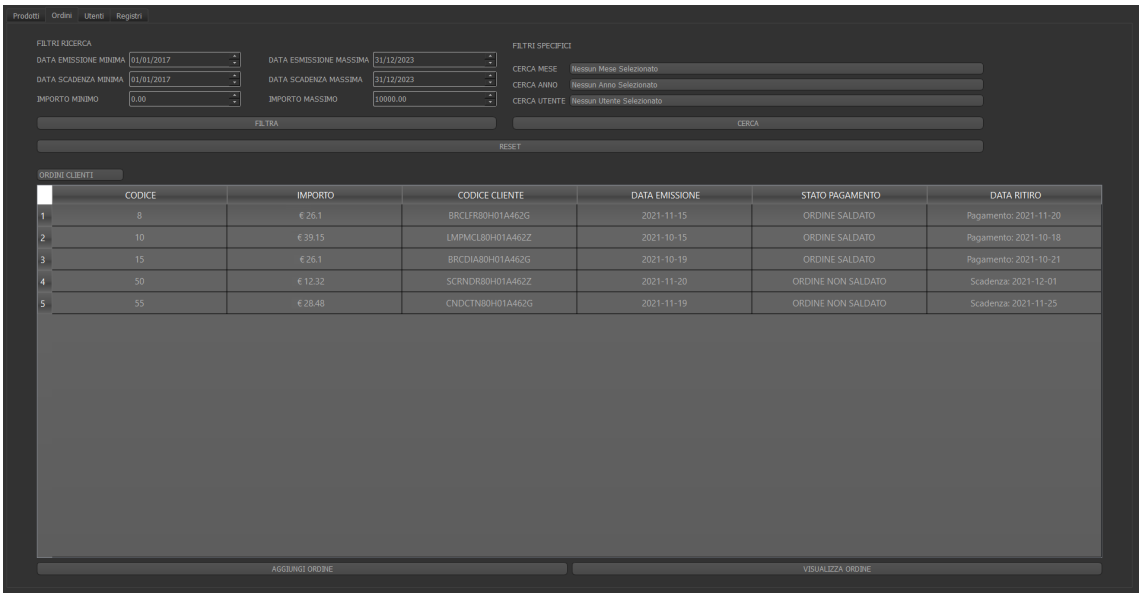


Figura 3.14: Sezione degli ordini

La descrizione si può effetturare individuando le tre macrosezioni in cui è suddivisa la schermata:

- *Sezione Filtri*: si trova nella parte superiore dell'interfaccia; essa, a sua volta, si divide in due aree, le quali permettono di selezionare, attraverso dei campi per l'inserimento, i parametri di ricerca. La prima sottosezione riguarda i filtri relativi a data di emissione, data di scadenza del pagamento e importo; la seconda, invece permette di effettuare una ricerca più specifica in base ad anno e mese di emissione oppure all'utente associato.
- *Tabella Ordini*: disposta nella zona centrale, essa mostrerà le informazioni relative agli ordini correntemente visualizzati.
- *Pulsanti degli eventi*: sono collocati nella parte inferiore della schermata, essi permettono di accedere alle interfacce dedicate alla gestione dei dettagli dei singoli ordini.

Inoltre è presente un pulsante di switch per cambiare la tipologia di ordini visualizzati (Clienti o Fornitori).

Nelle prossime sezione si analizzeranno le funzionalità accessibili all'interno dell'interfaccia, descrivendo, per ciascuna, le caratteristiche grafiche, la sequenza di azioni associate e la logica implementativa.

Valgono, inoltre, per questa area del programma, le stesse premesse fatte nei precedenti capitoli: l'implementazione di alcune funzionalità risulta analoga sia per gli ordini associati ai clienti, sia per quelli riguardanti i fornitori.

Perciò, nella descrizione dei diagrammi delle sequenze, relative alle funzioni in questione, per non fornire informazioni ridondanti, si è deciso di generalizzare utilizzando il concetto di "ordine".

Tuttavia, dove è necessario, verrà specificata quale classe di ordini si sta analizzando.

3.3.1 Ricerca degli ordini

Realizzazione grafica e logica degli eventi

Il titolare potrà impostare i filtri precedentemente elencati e, mediante gli appositi tasti, di applicarli alla tabella.

La Figura 3.15 descrive il flusso di attività associato alla funzionalità di ricerca; in particolare:

- Il titolare imposta dei parametri per il filtraggio e conferma la ricerca.
- Nel caso ci siano ordini che soddisfano i criteri selezionati, questi vengono visualizzati nella schermata.
- Nel caso in cui la ricerca non produca alcun risultato, il sistema avvisa il titolare mediante una finestra di errore.

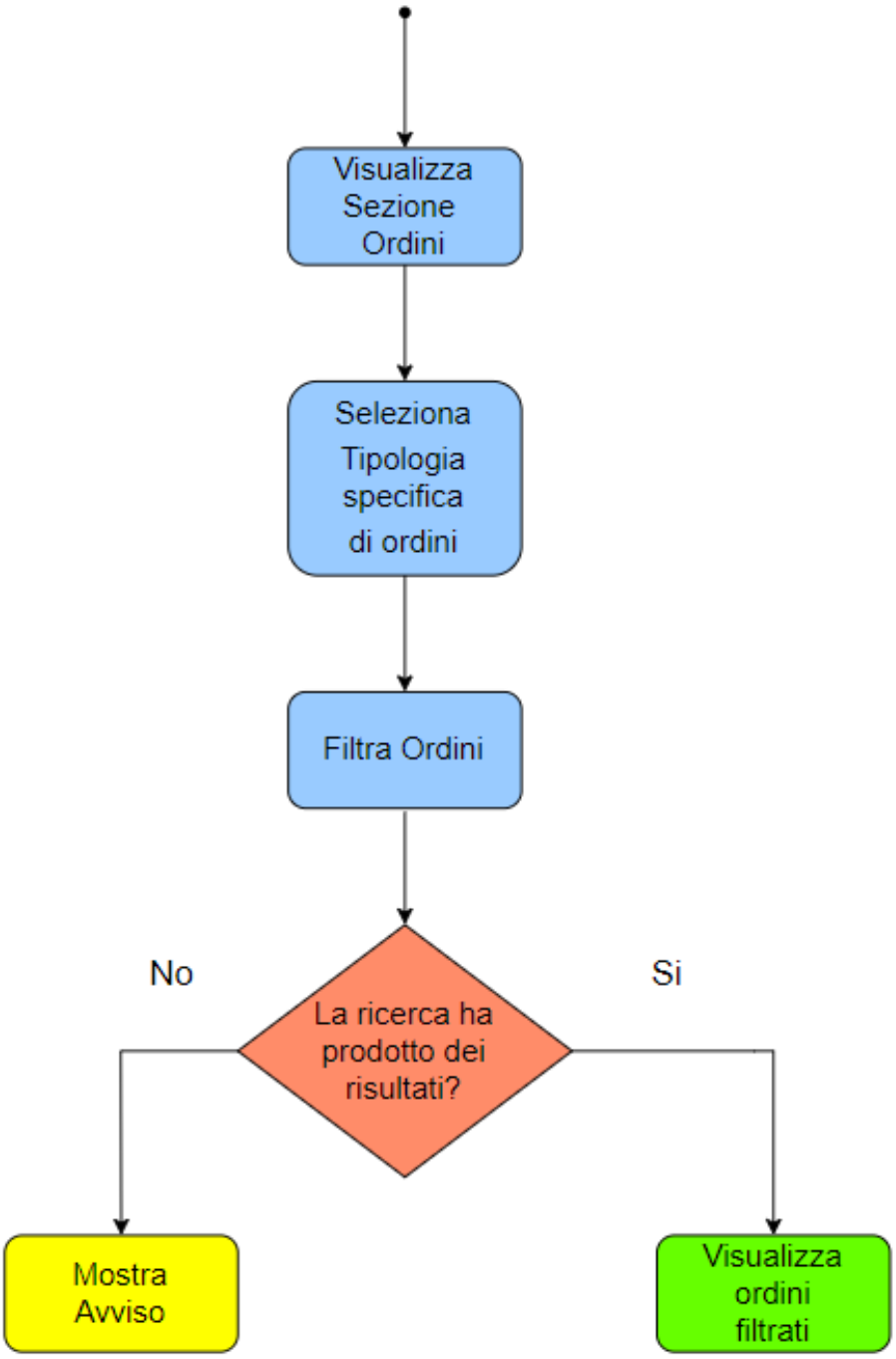


Figura 3.15: Sequenza di azioni relativa al filtraggio della lista degli ordini

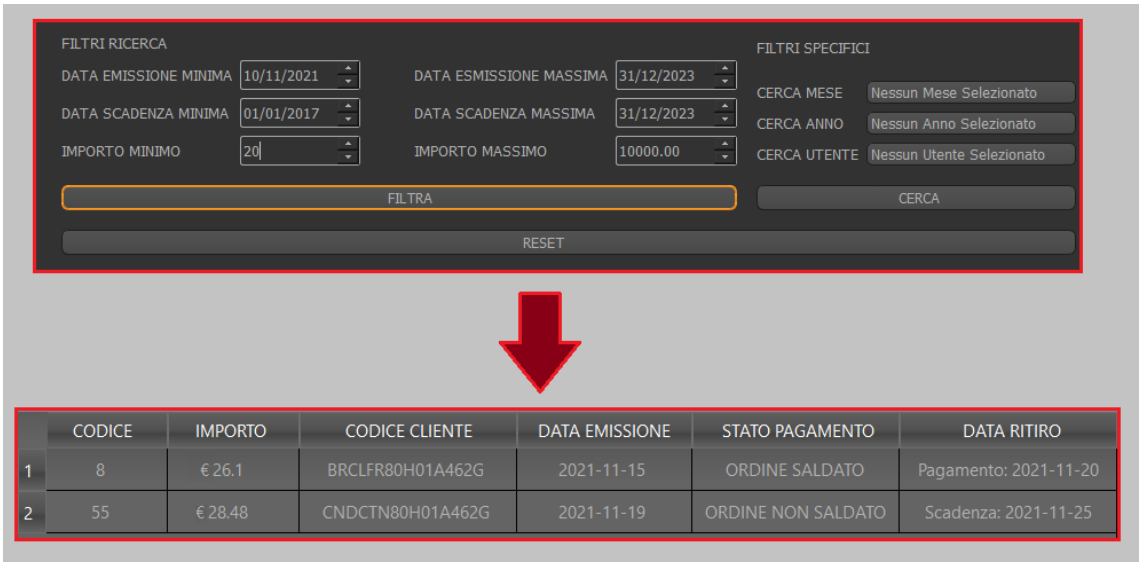


Figura 3.16: Esempio del filtraggio degli ordini

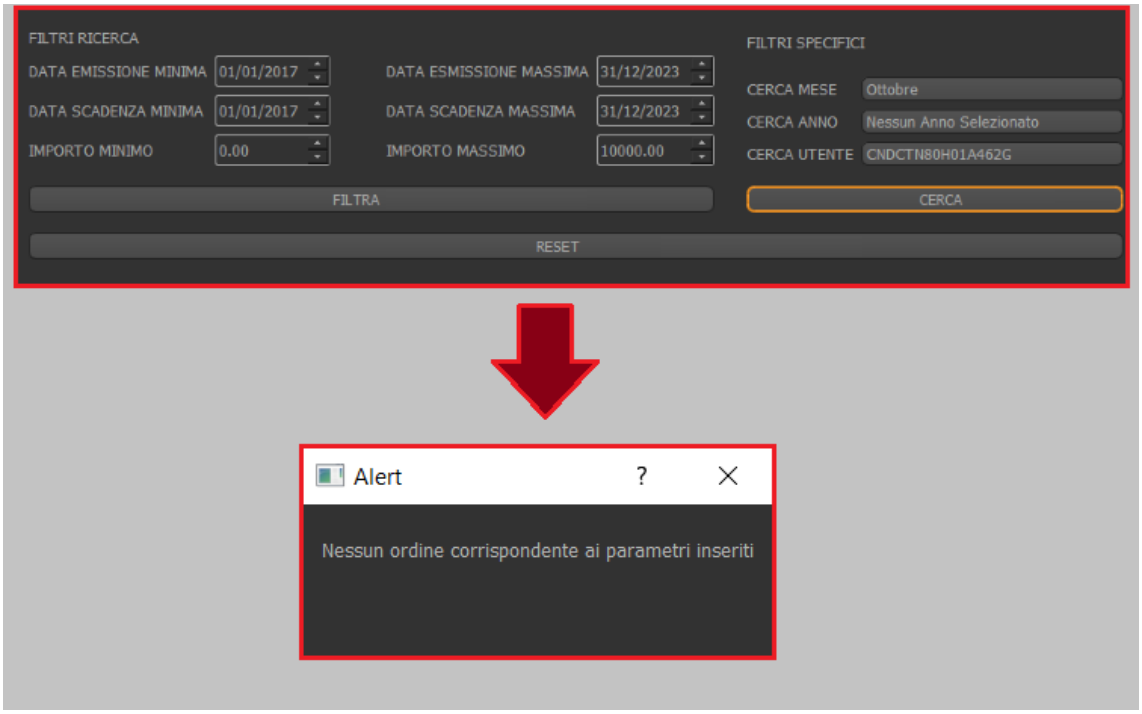


Figura 3.17: Esempio dell'utilizzo dei filtri specifici (nessuno ordine trovato)

Le Figure 3.16 e 3.17 descrivono come dovranno essere realizzate graficamente le varie fasi di questo scenario.

3.3.2 Aggiunta di un nuovo ordine

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.16, rappresenta il flusso di attività relativo all'inserimento di un nuovo ordine nel sistema.

Il flusso di azioni rappresentato può essere descritto come segue:

- Il titolare, che si trova nell'area ordini, in particolare nella sezione dedicata alla tipologia desiderata, seleziona la funzionalità per l'aggiunta di un nuovo ordine.
- Il sistema visualizza, in modo sequenziale, l'interfaccia per selezionare la tipologia dei prodotti da ordinare (soltanto nel caso in cui sia un ordine relativo ad un fornitore) e l'utente associato.
- Il sistema visualizza l'interfaccia per l'inserimento dei prodotti.
- Il titolare seleziona tutti i dettagli dell'ordine e, successivamente, conferma la creazione.
- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo ordine dovrà essere aggiunto nel sistema e mostrato nella tabella.

Le Figure 3.19 e 3.20 descrivono come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

I diagrammi nelle Figure 3.21 e 3.22 rappresentano le interazioni tra gli oggetti del sistema coinvolti nella creazione di un nuovo ordine associato ad un cliente; le Figure 3.23 e 3.24 descrivono il caso in cui l'ordine si riferisce ad un fornitore.

Oltre alla registrazione del nuovo ordine creato, realizzata da *OrderSectionController()* e *OrderSectionModel()*, lo scenario in analisi prevede diverse interazioni con il database.

Il sistema dovrà, innanzitutto, recuperare sia la lista degli utenti, per permettere al titolare di scegliere quello a cui associare l'ordine, sia la lista dei prodotti. Quest'ultima, in particolare, nel caso in cui l'ordine sia associato ad un fornitore, dovrà contenere soltanto i prodotti della tipologia dell'utente selezionato.

L'interfaccia per selezionare tutti i dettagli relativi all'ordine, definita da *NewOrderDialog()*, verrà gestita, a livello logico, da *OrderDetailsController()*. Questa classe, inoltre, dovrà sia indicare a *OrderDetailsModel()* quali sono i prodotti ordinati da registrare, sia restituire le quantità degli stessi a *OrderSectionController()*. Qualora l'ordine sia associato ad un cliente, le modifiche dovranno essere comunicate alla sezione prodotti mediante la funzione *ReflectMethod()*.

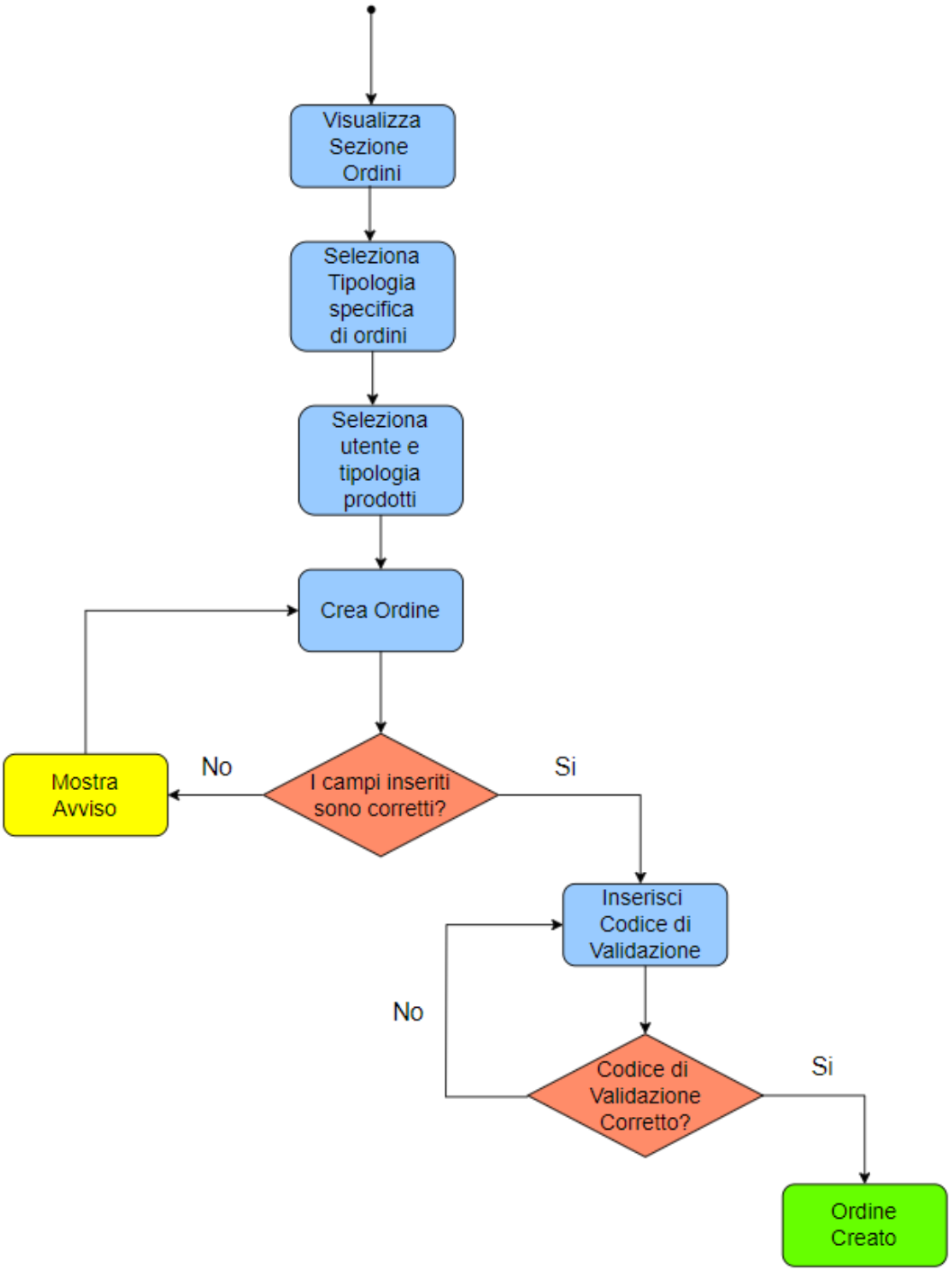


Figura 3.18: Sequenza di azioni relativa all'aggiunta di un nuovo ordine

Figura 3.19: Selezione dell'utente associato all'ordine

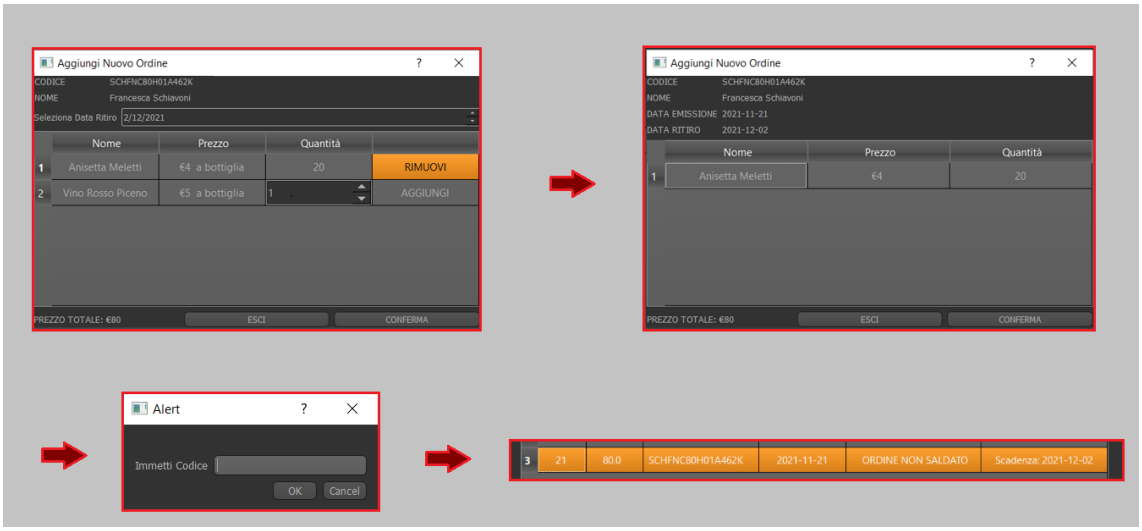


Figura 3.20: Creazione del nuovo ordine

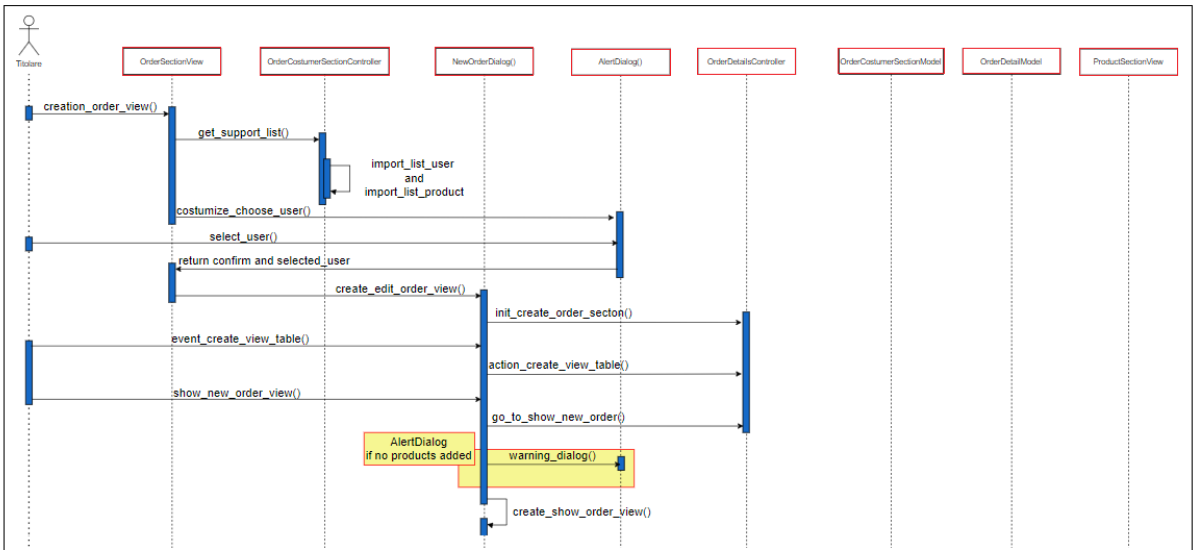


Figura 3.21: Diagramma delle sequenze relativo alla creazione di un nuovo ordine di un cliente

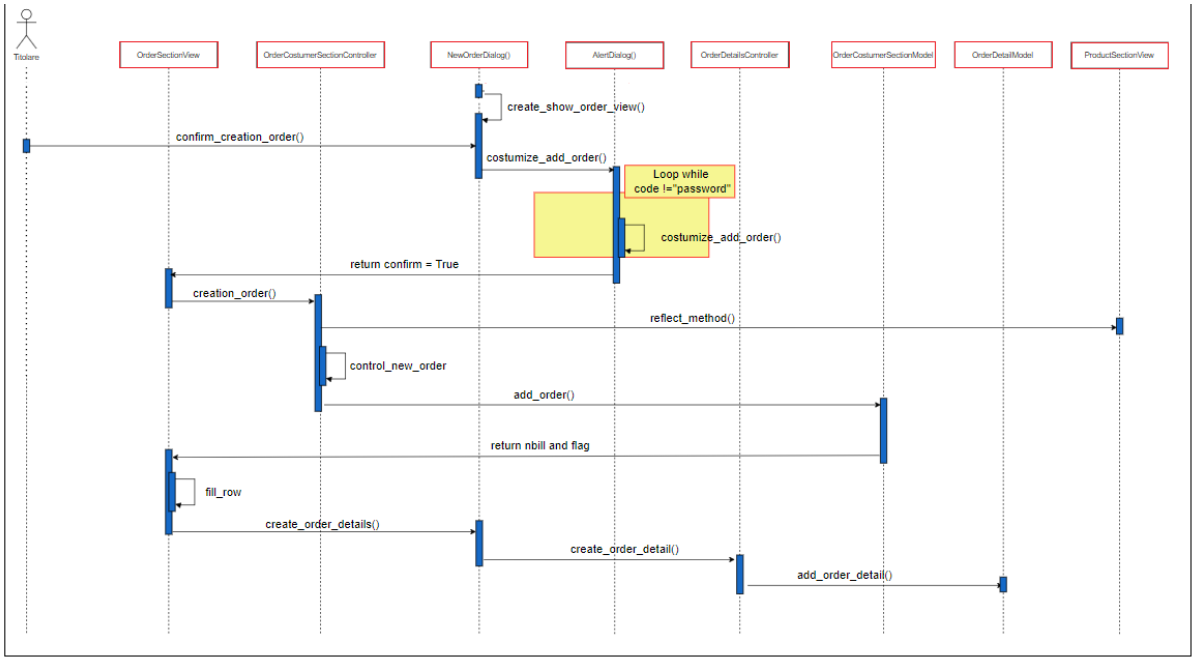


Figura 3.22: Diagramma delle sequenze relativo alla registrazione di un nuovo ordine di un cliente

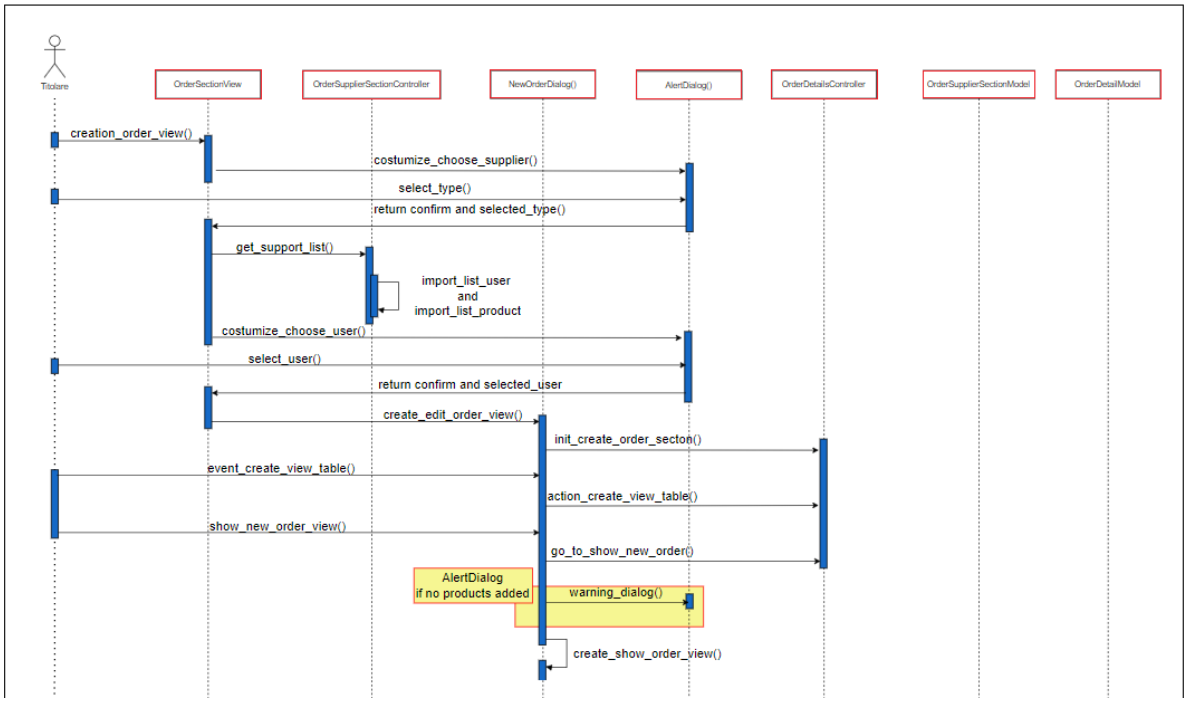


Figura 3.23: Diagramma delle sequenze relativo alla creazione di un nuovo ordine associato ad un fornitore

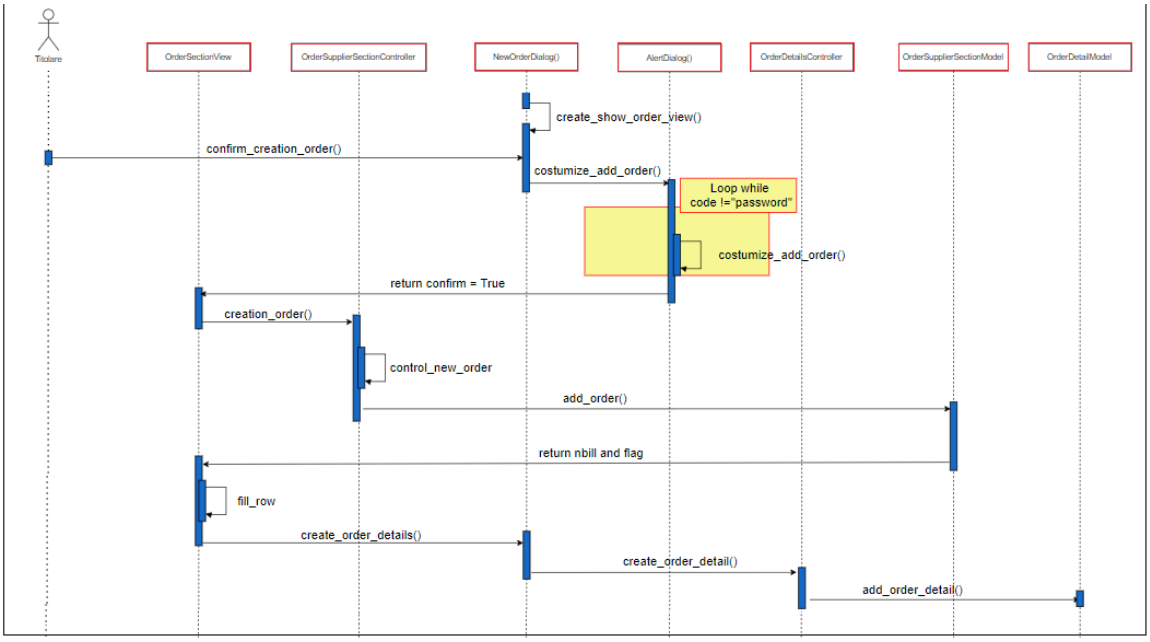


Figura 3.24: Diagramma delle sequenze relativo alla registrazione di un nuovo ordine associato ad un fornitore

3.3.3 Visualizzazione di un ordine presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.25, rappresenta il flusso di attività relativo alla visualizzazione dei dettagli di un ordine presente nel sistema.

Il flusso di azioni rappresentato può essere descritto come segue:

- Il titolare, che si trova nell'area ordini, in particolare nella sezione dedicata alla tipologia desiderata, seleziona la funzionalità per visualizzare un ordine.
- Il sistema visualizza un'interfaccia dove verrà richiesto di inserire il codice appartenente all'ordine che si vuole modificare.
- Il sistema, se trova un elemento associato al codice inserito, manda in esecuzione un'interfaccia con le informazioni relative al prodotto.

La Figura 3.26 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.27 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità.

OrdersSectionView(), attraverso il metodo *CostumizeSelectOrder()*, manda in esecuzione un'istanza di *AlertDialog()* per selezionare il codice dell'ordine da visualizzare. L'ordine verrà cercato e, successivamente, recuperato da *OrdersSectionController()* e *OrdersSectionModel()*.

OrderDetailsController(), infine, preleva i prodotti associati all'ordine dalla classe *OrderDetailsModel()*; questi saranno visibili nell'interfaccia gestita da *ModifyOrderDialog()* nel metodo **InitShowOrderView()**

3.3.4 Modifica di un ordine presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.28, rappresenta il flusso di attività relativo alla modifica di un ordine presente nel sistema.

Il flusso di azioni rappresentato può essere descritto come segue:

- Il titolare, che sta visualizzando uno degli ordini, seleziona il tasto 'modifica'
- Se l'ordine è già stato saldato, il sistema avverte il titolare che non è più possibile modificare l'ordine .
- Il sistema visualizza l'interfaccia per l'inserimento di prodotti e la modifica di quelli già presenti.

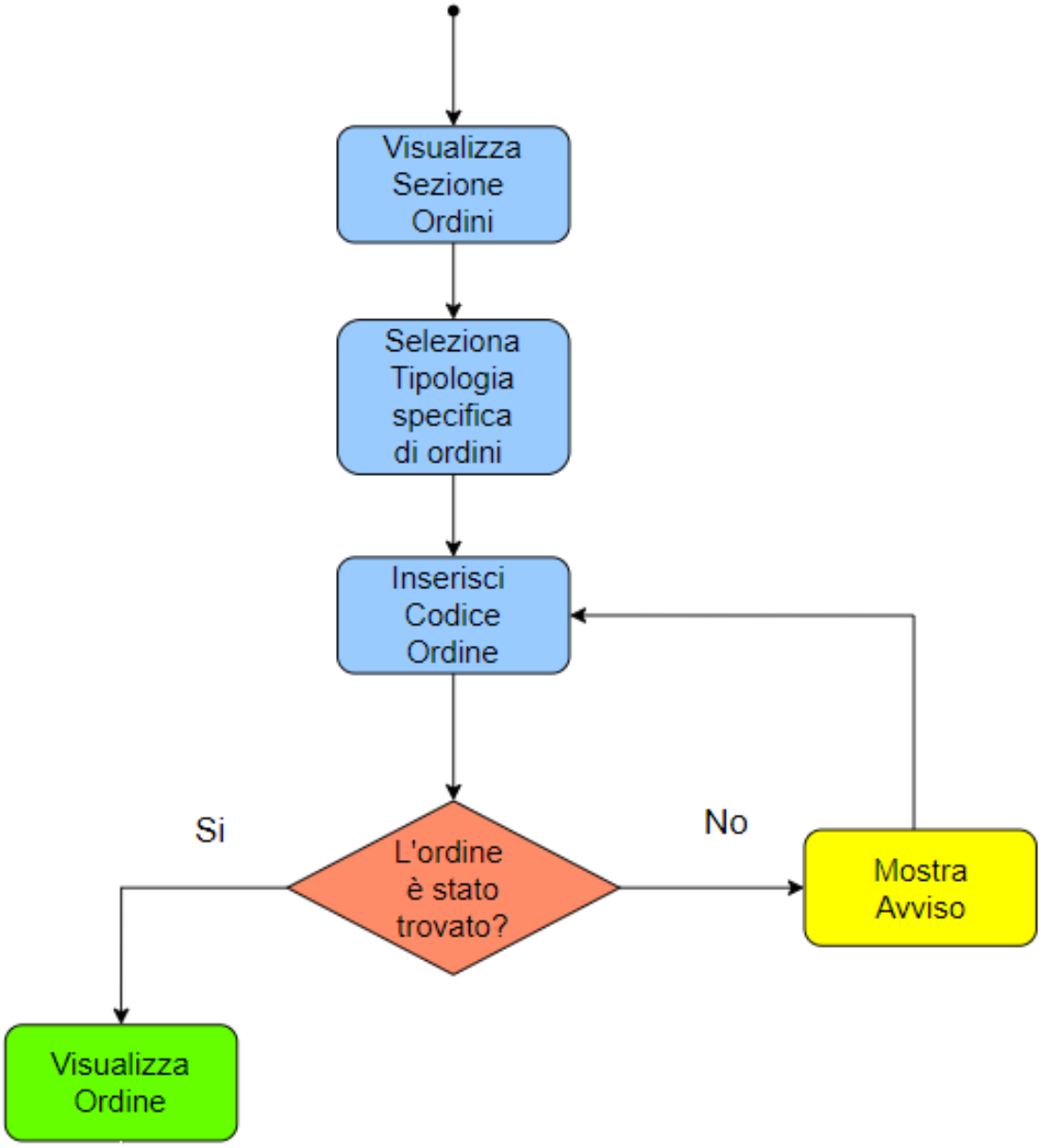


Figura 3.25: Sequenza di azioni relativa alla selezione di un ordine e visualizzazione della lista dei prodotti associata

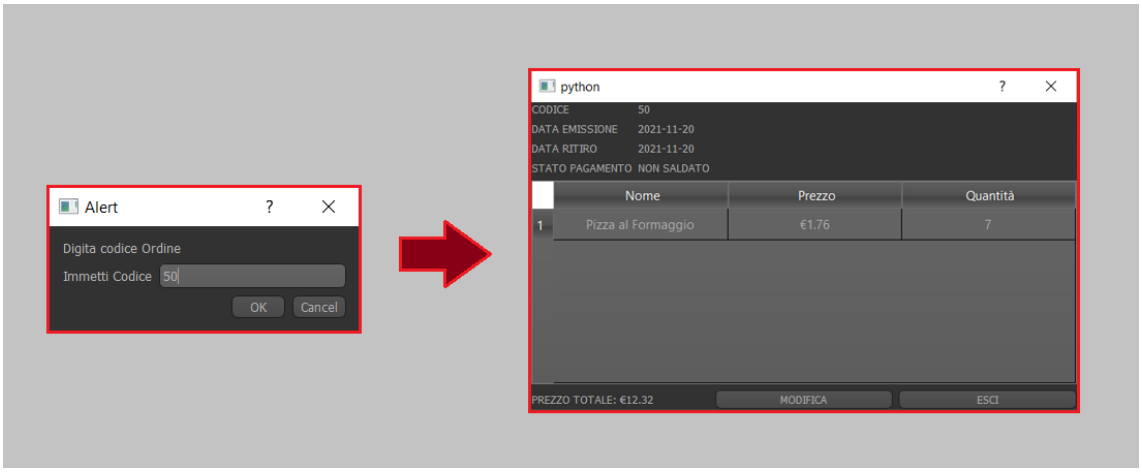


Figura 3.26: Visualizzazione dei dettagli dell'ordine

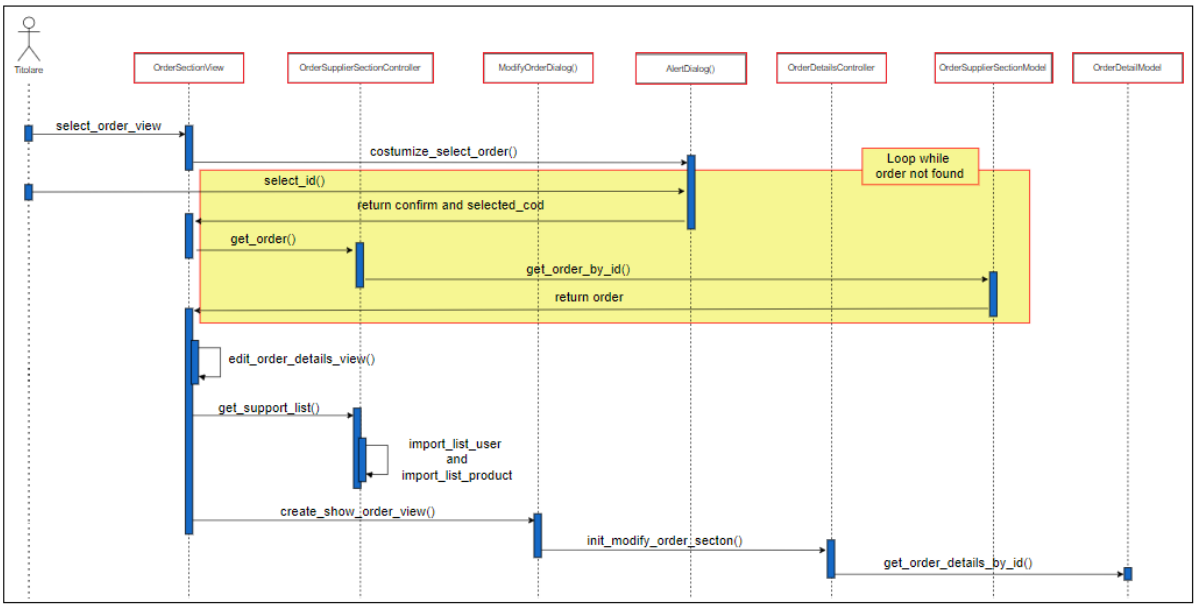


Figura 3.27: Diagramma delle sequenze relativo alla visualizzazione dei dettagli di un ordine esistente

- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, i nuovi dati relativi all'ordine dovranno essere aggiornati nel sistema e mostrati nella tabella.

La Figura 3.29 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.30 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella modifica di un ordine associato ad un utente, nel caso degli ordini relativi ai fornitori, il diagramma di riferimento è quello nella Figura 3.31.

Il titolare può accedere alla funzionalità di modifica soltanto nel caso in cui stia visualizzando i dettagli relativi all'ordine. Di conseguenza, il sistema si trova nello stato in cui sia i prodotti all'interno del magazzino, sia quelli associati all'ordine sono stati già prelevati dal database.

Viene mandata in esecuzione una nuova istanza di *ModifyOrderDialog()*; l'interfaccia visualizzata, creata mediante il metodo *CreateEditOrderView()*, permetterà di modificare ed eliminare i prodotti già presenti nell'ordine, oppure di aggiungerne nuovi.

Analogamente al caso della funzionalità di creazione, *OrdersSectionController()* e *OrdersSectionModel()* si dovranno occupare dell'aggiornamento delle informazioni relative all'ordine ed, eventualmente, notificare alla sezione prodotti la modifica dei prodotti. *OrdersDetailsController()* e *OrdersDetailsModel* gestiranno la modifica della lista dei prodotti associati.

3.3.5 Eliminazione di un ordine presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.32, rappresenta il flusso di attività relativo all'inserimento di un nuovo ordine nel sistema.

Il flusso di azioni rappresentato può essere descritto come segue:

- Il titolare, che sta visualizzando uno degli ordini, seleziona il tasto "Modifica"
- Se l'ordine è già stato saldato, il sistema avverte il titolare che non è più possibile modificare l'ordine
- Il sistema visualizza l'interfaccia per l'inserimento di prodotti e la modifica di quelli già presenti.
- Il titolare preme il tasto per "Elimina".
- Viene mostrata un'interfaccia per l'inserimento di un codice di validazione.

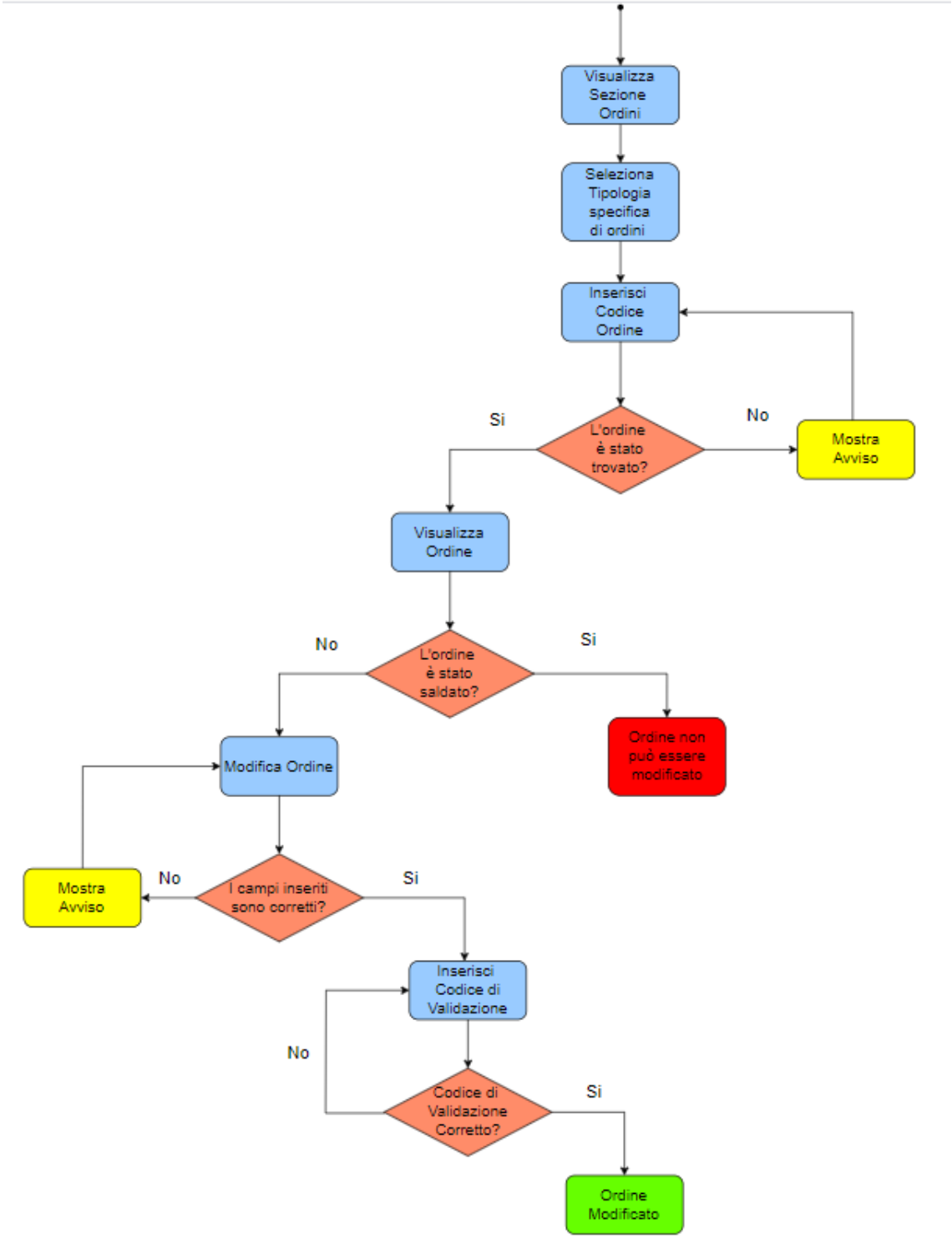


Figura 3.28: Visualizzazione dettagli ordine

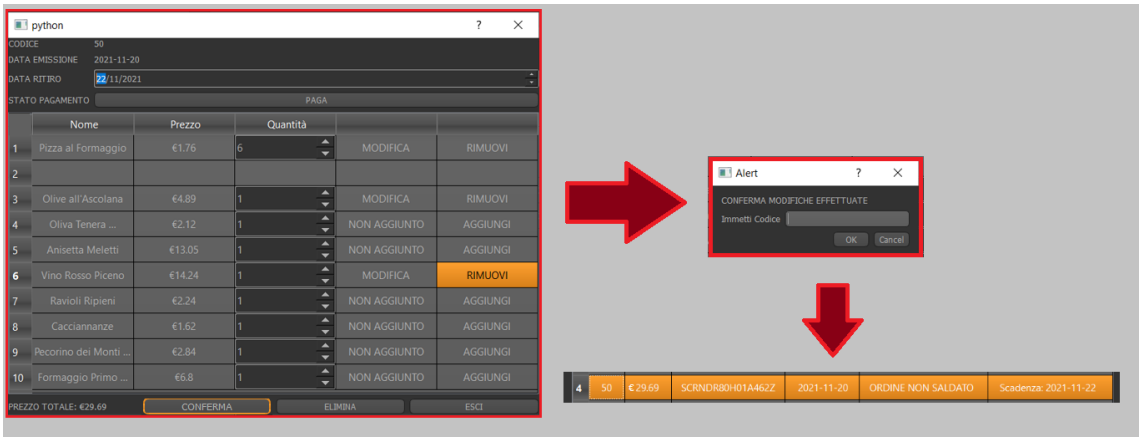


Figura 3.29: Modifica di un un ordine

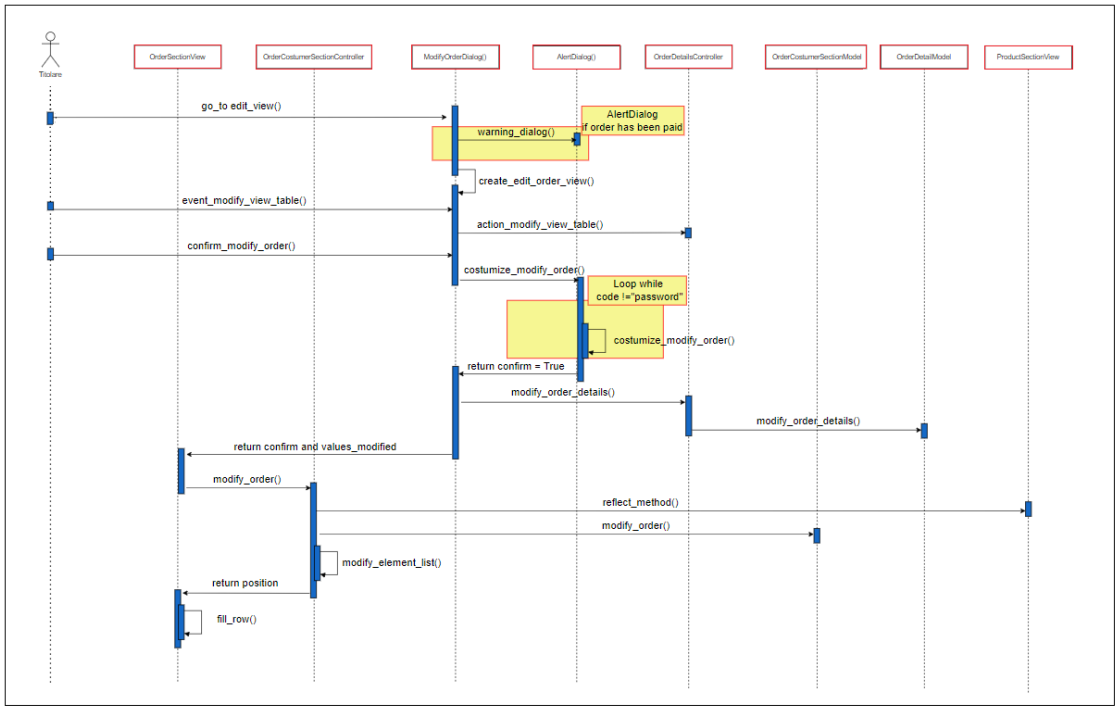


Figura 3.30: Diagramma delle sequenze relativo alla modifica di un ordine appartenente ad un cliente

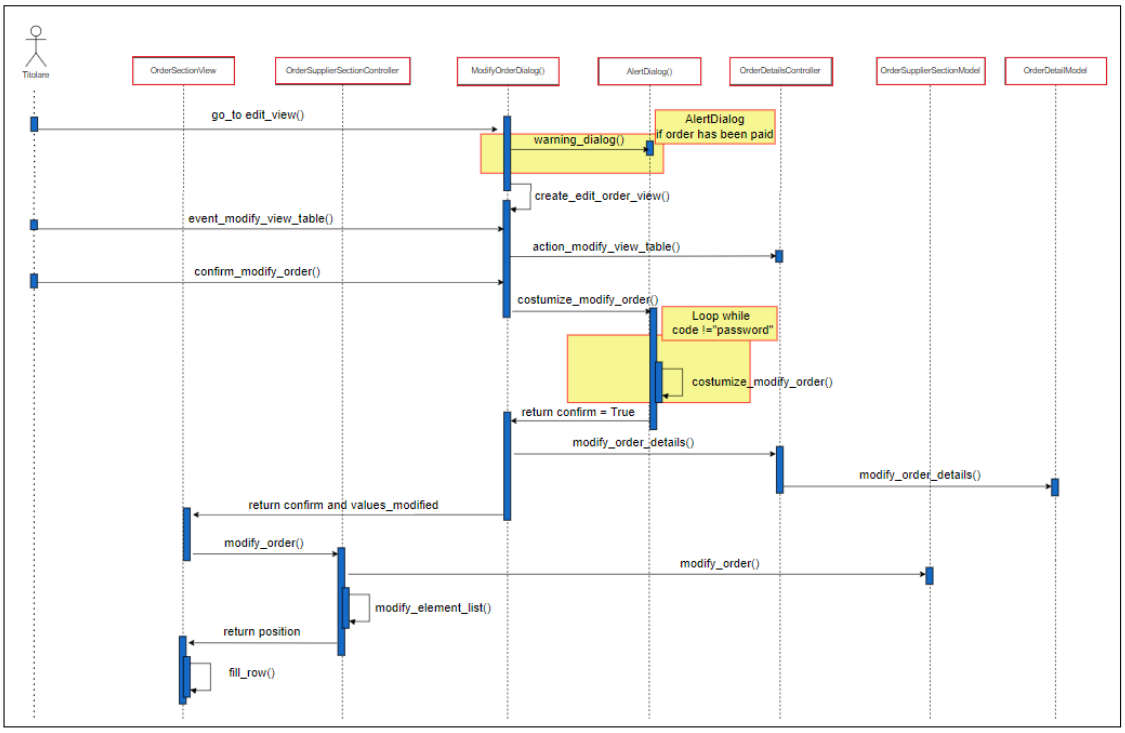


Figura 3.31: Diagramma delle sequenze relativo alla modifica di un ordine appartenente ad un fornitore

- In caso di validazione, i nuovi dati relativi all'ordine dovranno essere eliminati dal sistema e rimossi dalla tabella.

La Figura 3.33 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.34 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nell'eliminazione di un ordine associato ad un cliente, nel caso degli ordini relativi ai fornitori; il diagramma di riferimento è quello nella Figura 3.35.

Anche questa funzionalità è accessibile soltanto dalla schermata adibita alla visualizzazione dell'ordine; valgono, quindi, le stesse considerazioni fatte nel caso della modifica.

L'unica differenza sostanziale, oltre alle operazioni che verranno eseguite in caso di conferma, è la tipologia di interfaccia visualizzata; verrà mostrata soltanto la schermata per l'inserimento del codice di validazione.

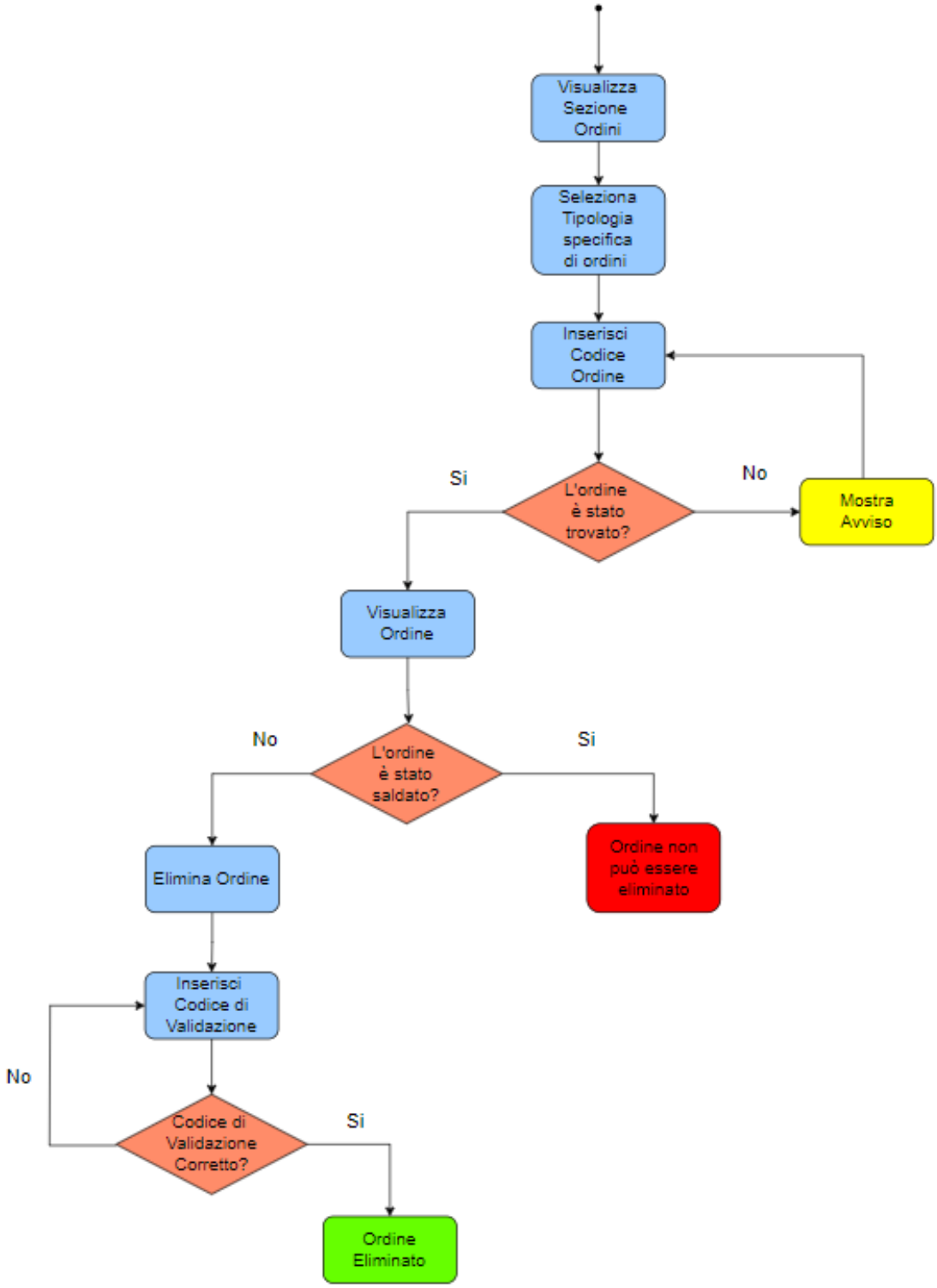


Figura 3.32: sequenza di azioni relativa all'eliminazione di un ordine presente nel sistema

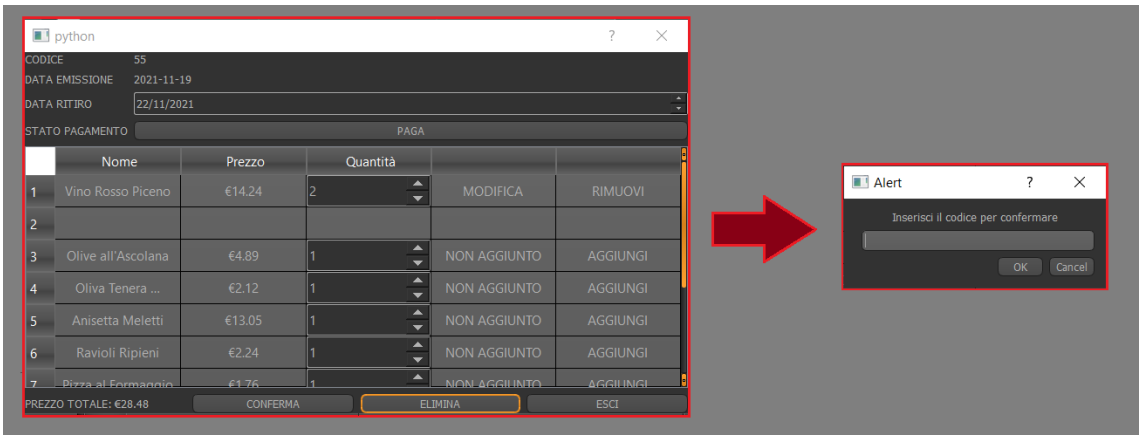


Figura 3.33: Eliminazione di un ordine presente nel sistema

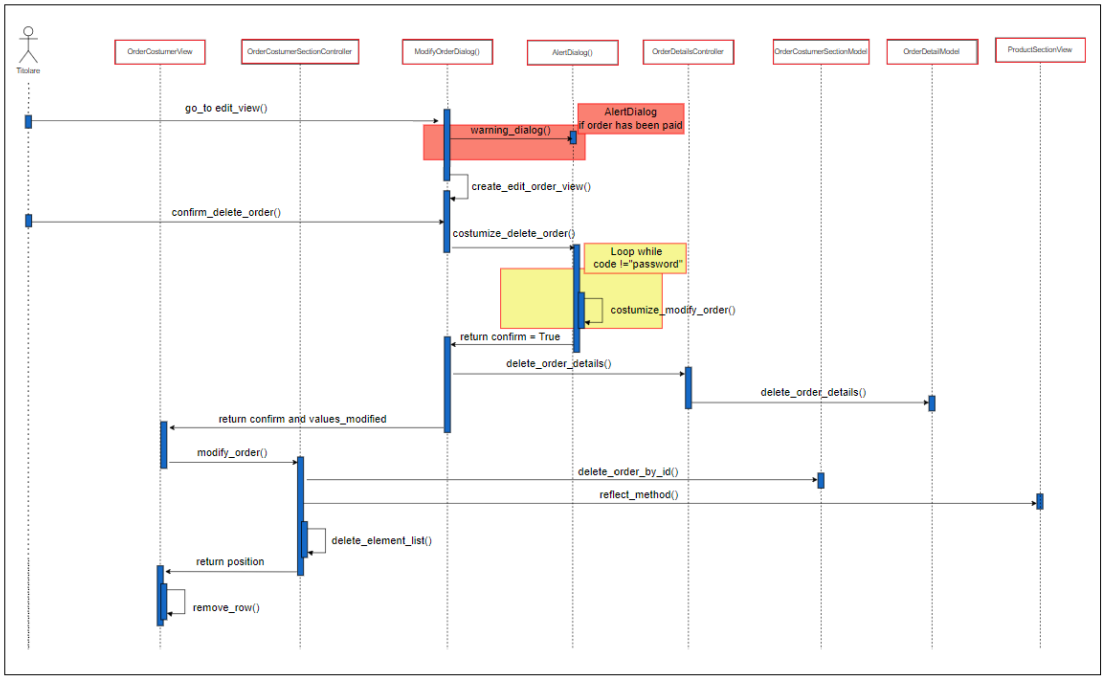


Figura 3.34: Diagramma delle sequenze relativo all'eliminazione di un ordine appartenente ad un cliente

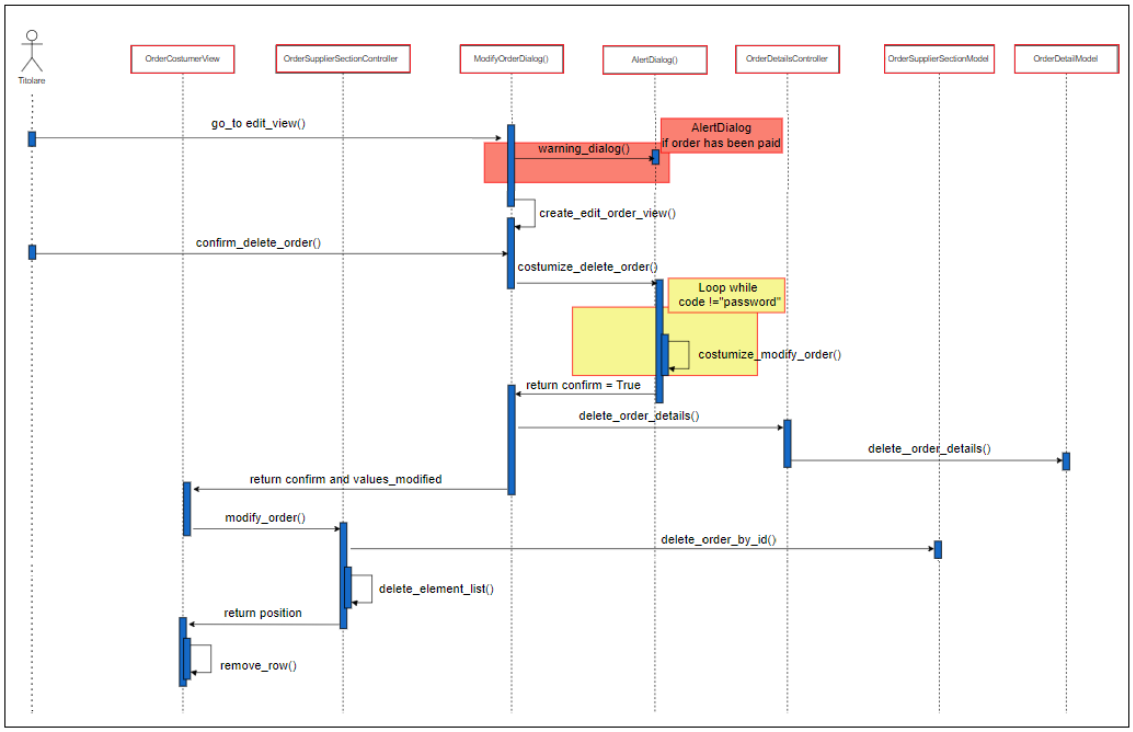


Figura 3.35: Diagramma delle sequenze relativo all'eliminazione di un ordine appartenente ad un fornitore

3.3.6 Pagamento di un ordine presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.36, rappresenta il flusso di attività relativo al pagamento di un ordine presente nel sistema.

- Il titolare, che sta visualizzando uno degli ordini, seleziona il tasto "Modifica"
- Se l'ordine è già stato saldato, il sistema avverte il titolare che non è più possibile modificare l'ordine.
- Il sistema visualizza l'interfaccia per l'inserimento di prodotti e la modifica di quelli già presenti.
- Il titolare preme il tasto 'Paga'.
- Viene mostrata un'interfaccia per l'inserimento di un codice di validazione e della fattura.
- In caso di validazione, dovrà essere registrata la fattura relativa al pagamento avvenuto.

La Figura 3.37 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario. Lo stato in cui si trova il sistema in questo scenario è lo stesso

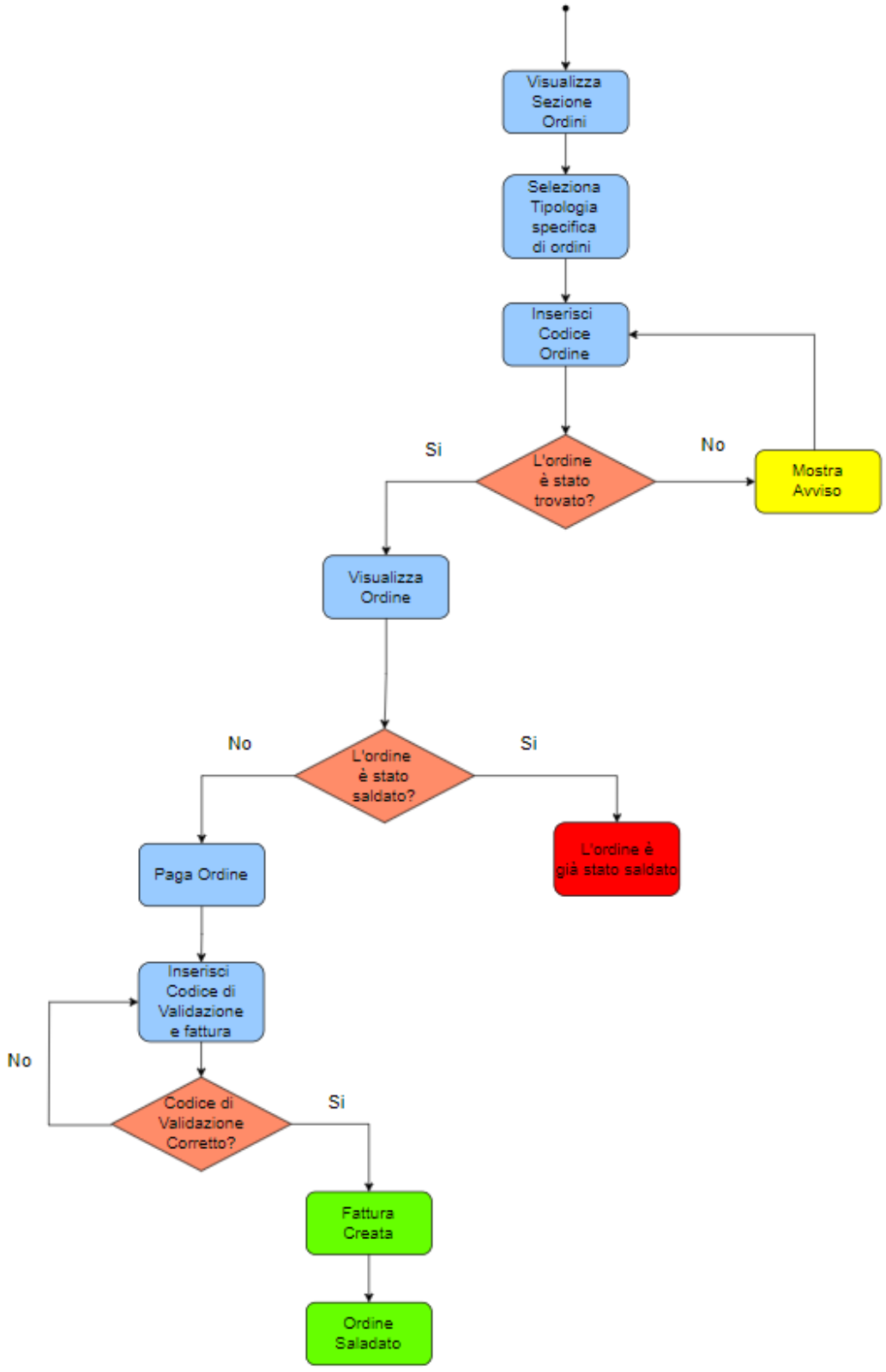


Figura 3.36: sequenza di azioni relativa al pagamento di un ordine presente nel sistema

Figura 3.37: Pagamento di un ordine associato ad un cliente

descritto nei tre precedenti. Tuttavia, le operazioni svolte, in caso di conferma, riguardano soltanto l'aggiornamento del campo relativo allo stato del pagamento. Inoltre, qualora l'ordine sia associato ad un fornitore, verrà notificata la modifica, mediante la funzione *ReflectMethod()* alla sezione dei prodotti,

Diagrammi delle sequenze associati

La Figura 3.38 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nel pagamento di un ordine associato ad un utente, nel caso degli ordini relativi ai fornitori; il diagramma di riferimento è quello mostrato nella Figura 3.39.

Sta volta la conferma dell'operazione implica, oltre alla modifica dell'attributo relativo allo stato del pagamento, la creazione di una nuova transazione.

3.4 Progettazione della sezione relativa agli utenti

L'area utenti sarà organizzata come descritto nella Figura 3.40. La descrizione si può effetturare individuando le tre macrosezioni in cui è suddivisa la schermata:

- *Sezione Filtri*: si trova nella parte superiore dell'interfaccia, essa contiene una barra di ricerca dove si potrà digitare il codice dell'utente desiderato.
- *Tabella Utenti*: disposta nella zona centrale; essa mostrerà le informazioni relative agli utenti correntemente visualizzati.
- *Pulsanti degli eventi*: sono collocati nella parte inferiore della schermata, essi permettono di accedere alle interfacce dedicate alla gestione dei singoli utenti.

Inoltre è presente un pulsante di switch per cambiare la tipologia di utenti visualizzati (Clienti, Dipendenti o Fornitori).

Nelle prossime sezioni si analizzeranno le funzionalità accessibili all'interno dell'interfaccia, descrivendo, per ciascuna di esse, le caratteristiche grafiche, la sequenza di azioni associate e la logica implementativa.

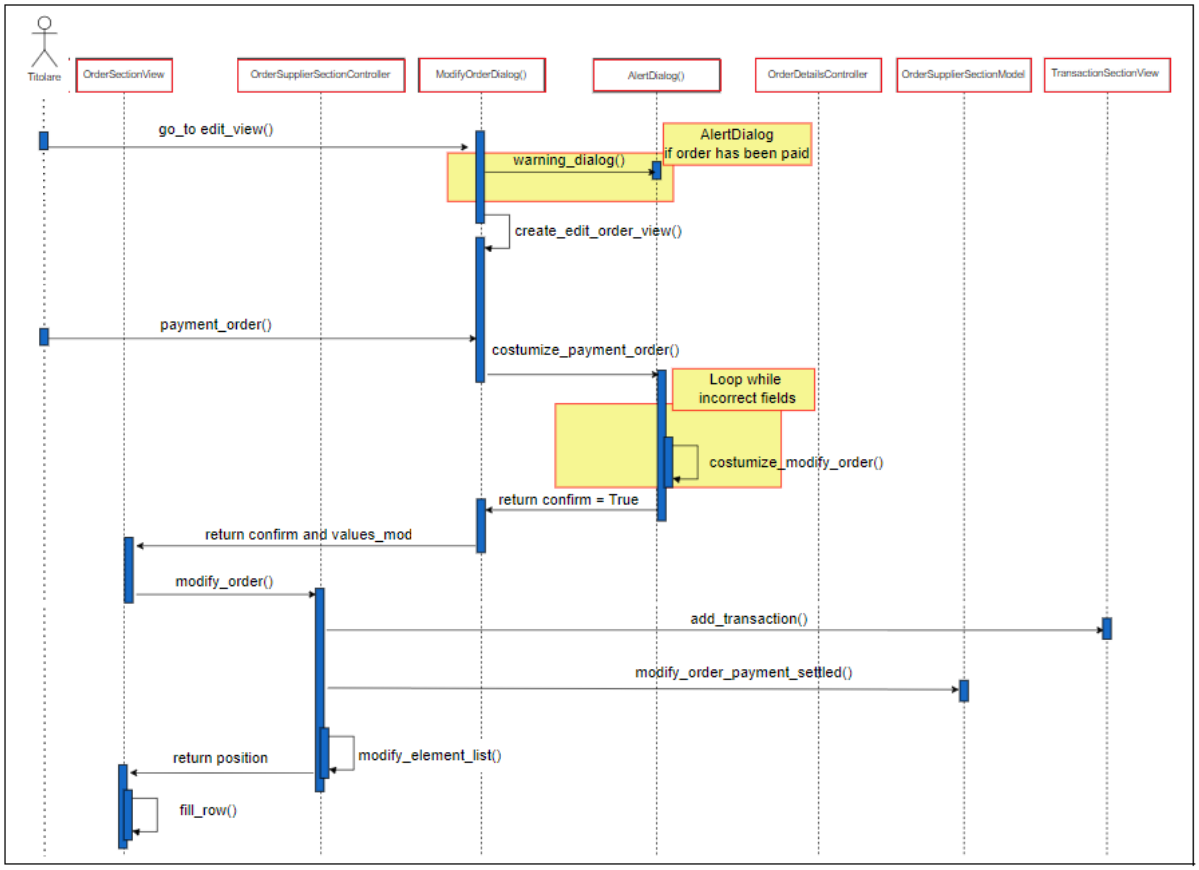


Figura 3.38: Diagramma delle sequenze relativo al pagamento di un ordine appartenente ad un cliente

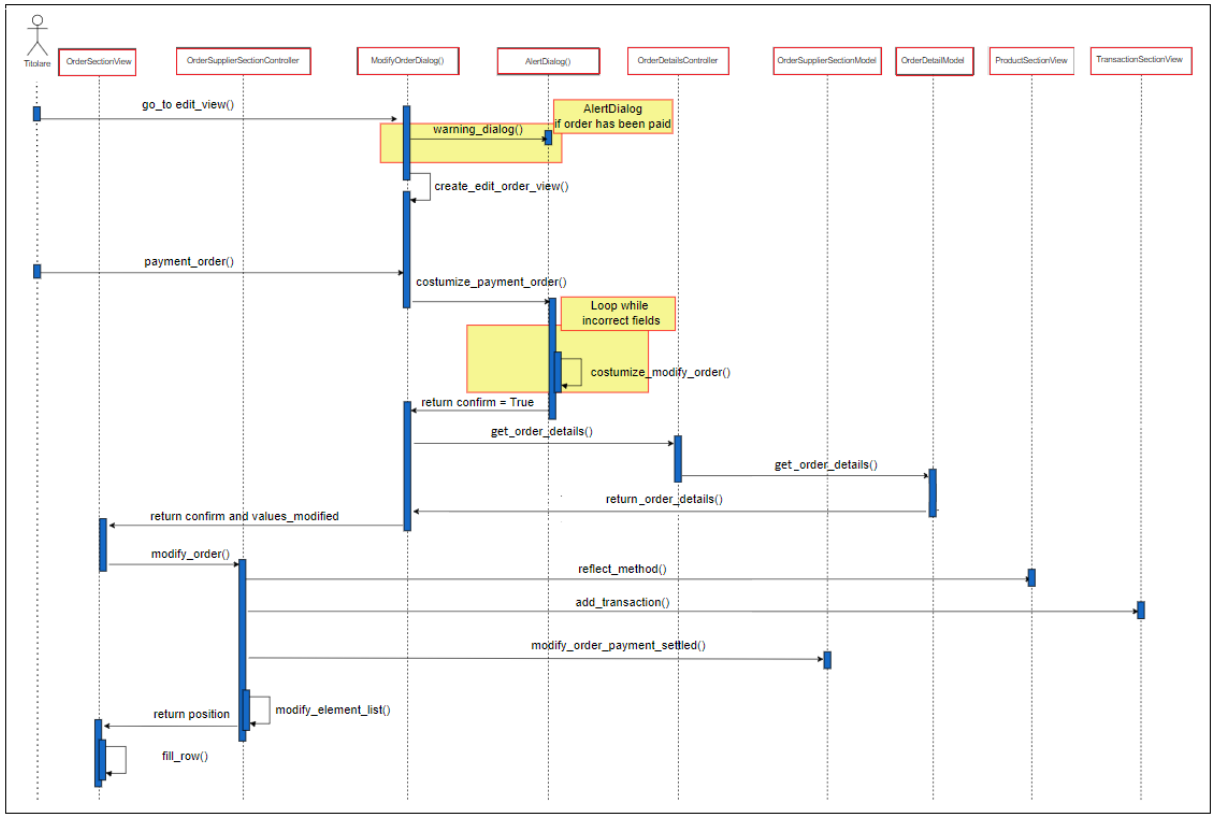


Figura 3.39: Diagramma delle sequenze relativo al pagamento di un ordine appartenente ad un fornitore

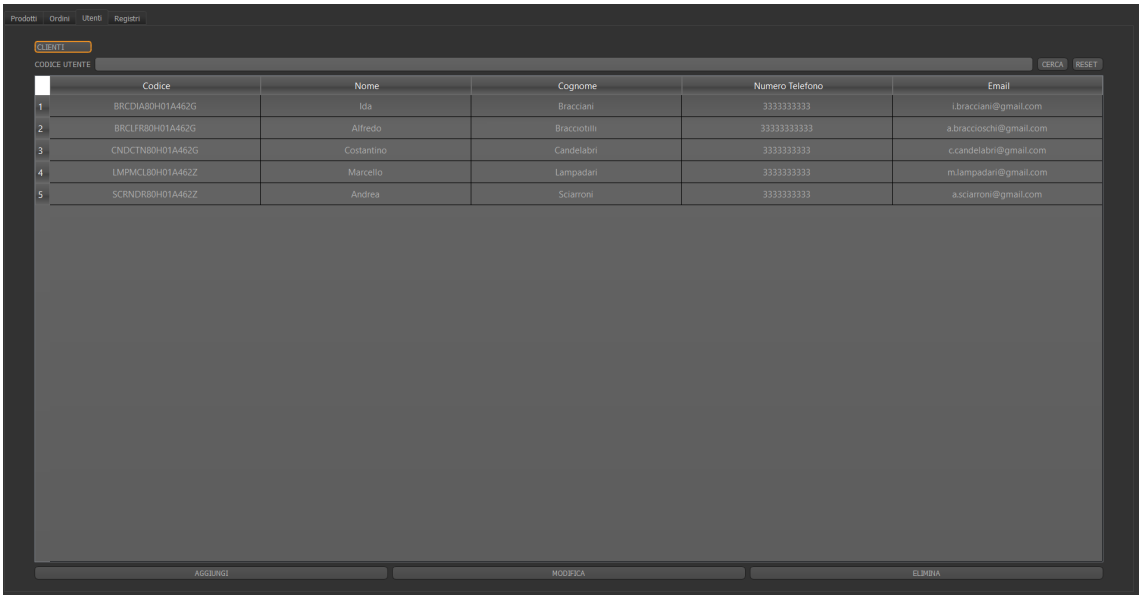


Figura 3.40: Interfaccia della sezione relativa agli Utenti

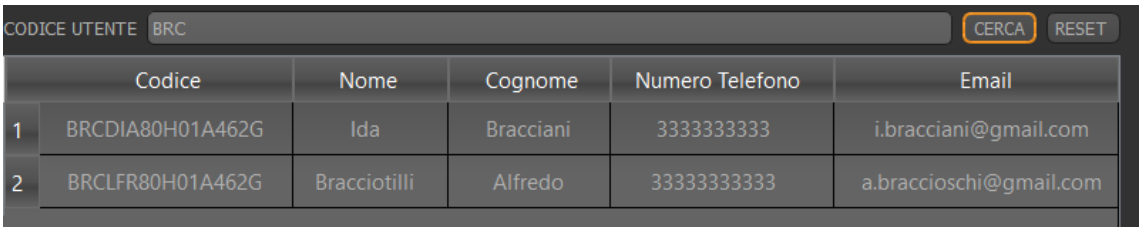


Figura 3.41: Esempio del filtraggio degli utenti

Valgono, inoltre, per questa area del programma, le stesse premesse fatte nei precedenti capitoli: l'implementazione di alcune funzionalità risulta analoga per le tre classi di utenti.

Perciò, nella descrizione dei diagrammi delle sequenze, per non fornire informazioni ridondanti, si è deciso di generalizzare utilizzando il concetto di "Utente".

Tuttavia, dove è necessario, verrà specificata quale classe di utenti si sta analizzando.

3.4.1 Ricerca degli Utenti

Realizzazione grafica e logica degli eventi

Il titolare potrà cercare gli utenti in base al loro codice identificativo (Figura 3.41).

Attraverso gli appositi tasti di conferma verranno applicati i filtri selezionati e, successivamente, a seconda dei risultati della ricerca, la tabella mostrerà gli utenti trovati.

La Figura 3.42 descrive il flusso di attività associato alla funzionalità di ricerca; in particolare:

- Il titolare digita una sequenza di caratteri e conferma la ricerca.
- Nel caso ci siano utenti con codice contenente la sequenza inserita vengono visualizzati nella schermata.
- Nel caso in cui la ricerca non produca alcun risultato, il sistema avvisa il titolare con una finestra di errore.

3.4.2 Aggiunta di un nuovo Utente

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.43, rappresenta il flusso di attività relativo all'inserimento di un nuovo utente nel sistema.

Esso può essere descritto come segue:

- Il titolare, che si trova nell'area Utenti, in particolare nella sezione dedicata alla tipologia desiderata, seleziona la funzionalità per l'aggiunta di un nuovo utente.
- Il sistema visualizza l'interfaccia per l'inserimento dei dati.
- Il titolare inserisce, negli appositi campi, le informazioni relative al nuovo utente e conferma l'operazione.
- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo utente dovrà essere aggiunto nel sistema ed, eventualmente, mostrato nella tabella.

La Figura 3.43 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.45 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità.

Il titolare, premendo il tasto "Aggiungi", richiama il metodo *AddUserView()* di *UsersSectionView()*, che manda in esecuzione un'istanza di *EditUserDialog()*; l'interfaccia risultante, descritta dal metodo *CreateUi()*, permetterà di inserire le informazioni relative al nuovo utente.

Nel caso in non siano presenti errori nei dati inseriti o nella fase di validazione, *UsersSectionView()* richiama il metodo *AddUser()* di *UsersSectionController()*; quest'ultimo invoca la funzionalità della classe *UsersSectionModel()* per aggiungere il nuovo utente nel sistema.

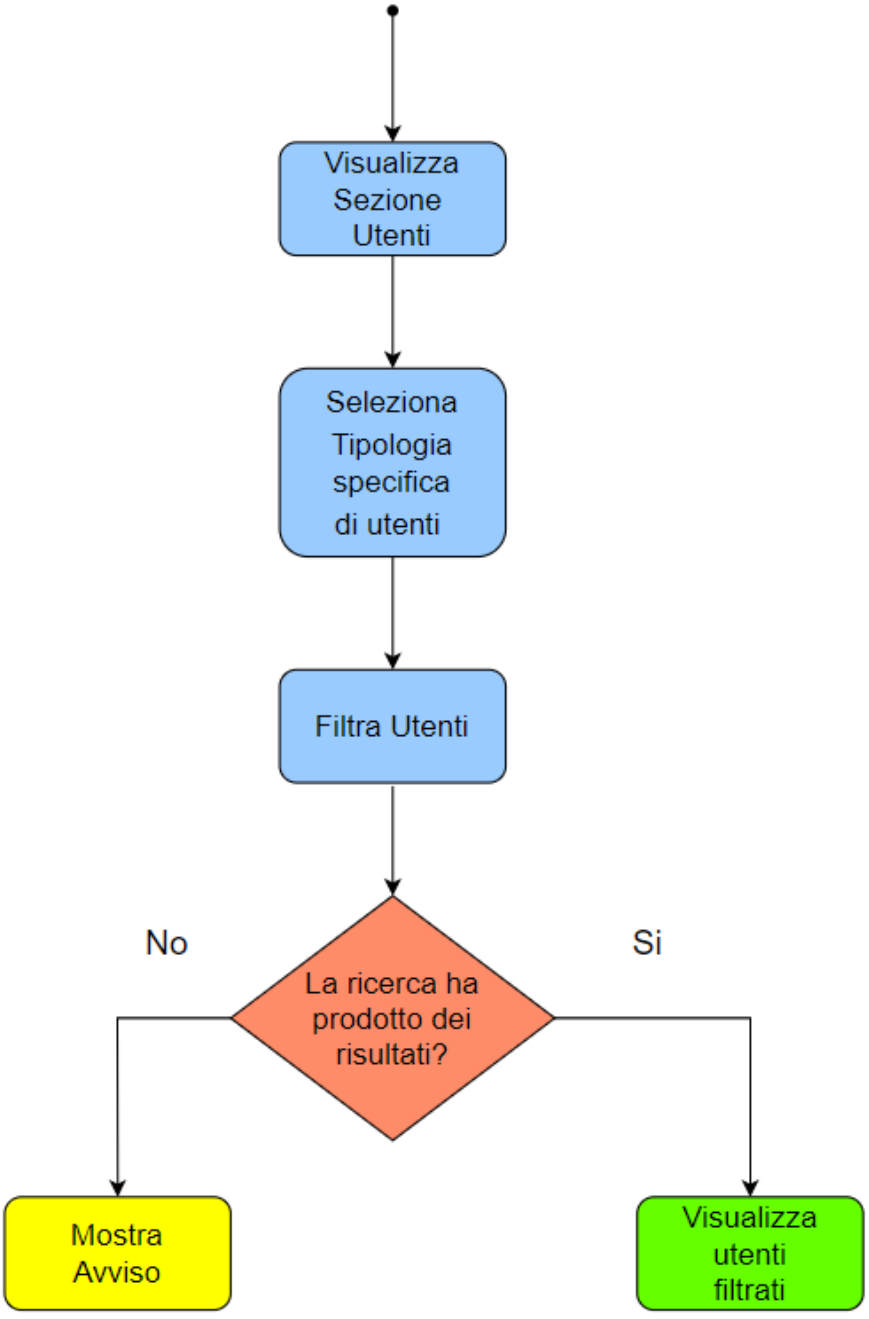


Figura 3.42: Sequenza di azioni relativa al filtraggio della lista utenti

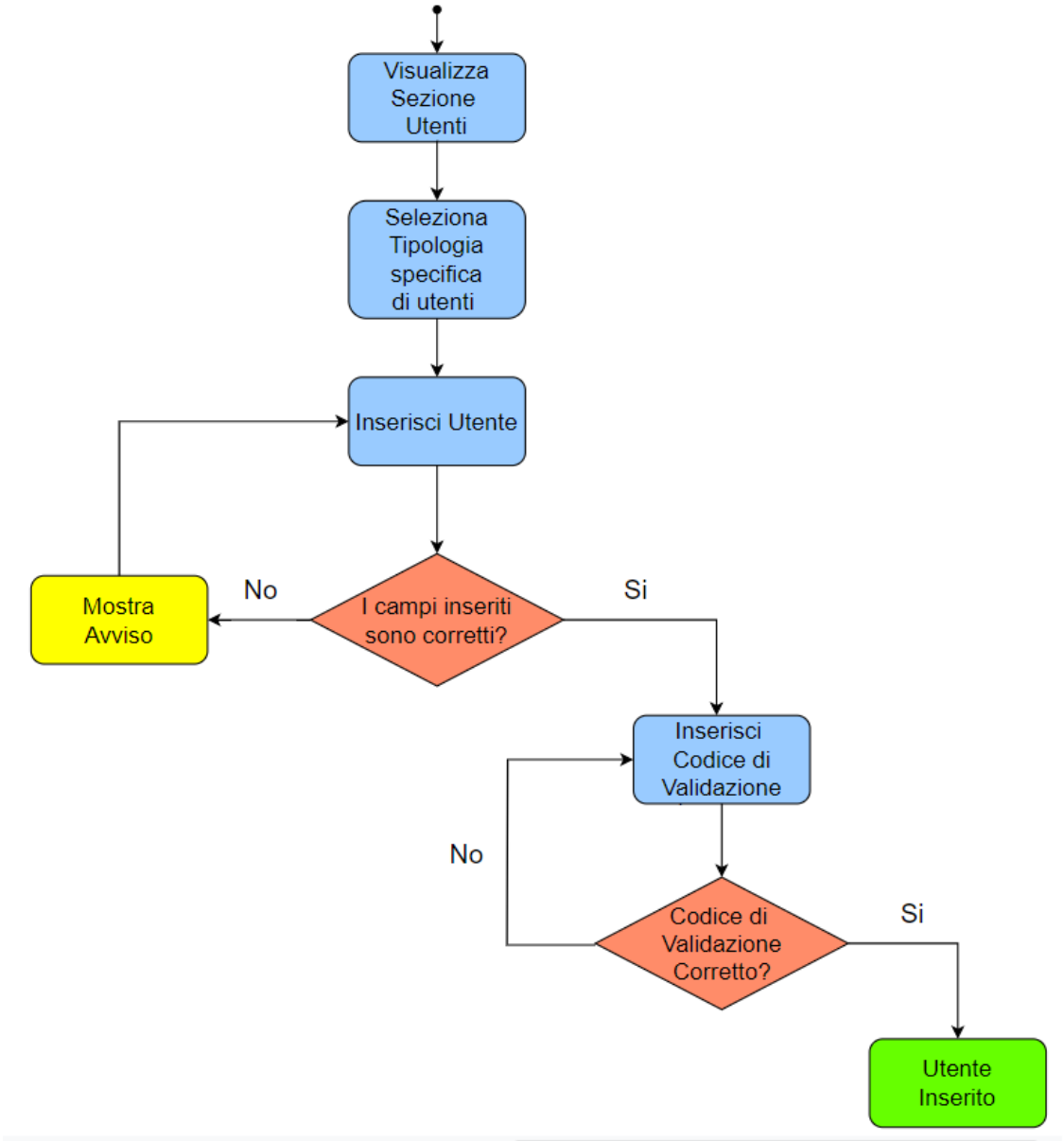


Figura 3.43: Sequenza di azioni relativa all'aggiunta di un nuovo utente

Figura 3.44: Creazione di un nuovo dipendente

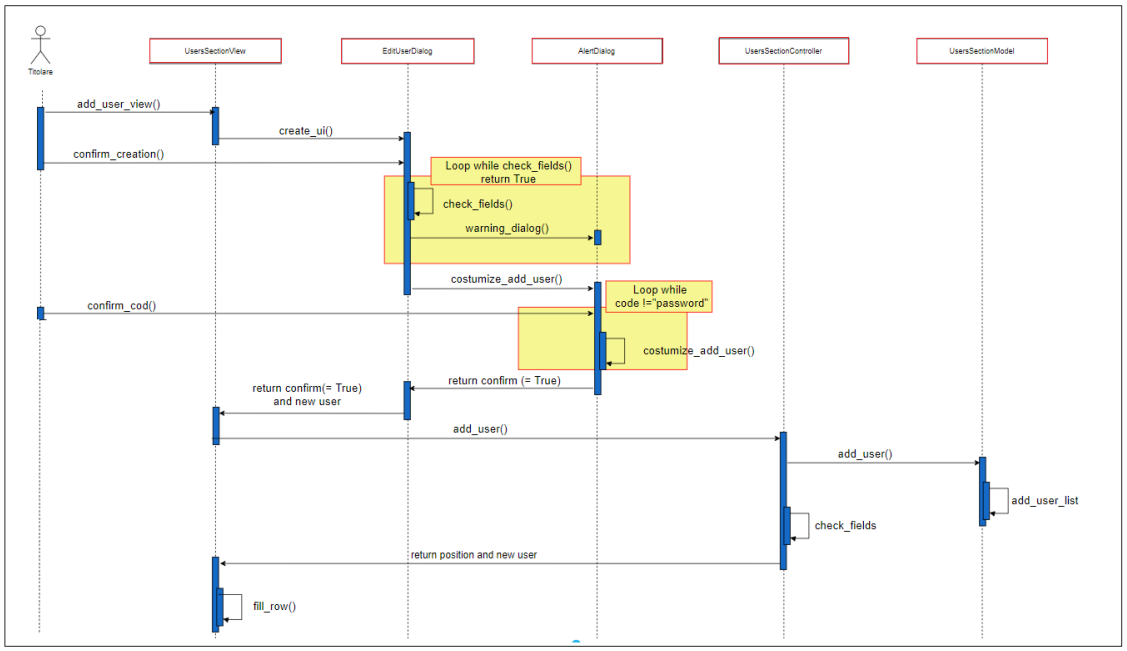


Figura 3.45: Diagramma delle sequenze relativo all'aggiunta di un nuovo Utente

3.4.3 Modifica di un utente presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.46, rappresenta il flusso di attività relativo alla modifica di un utente presente nel sistema.

Esso può essere descritto come segue:

- Il titolare, che si trova nell'area utenti, in particolare sta visualizzando la tipologia di utenti desiderata; egli seleziona la funzionalità per la modifica di un utente presente nel sistema.
- Il sistema visualizza l'interfaccia di modifica dove, inizialmente, verrà richiesto di inserire il codice appartentente al utente che si vuole modificare.
- Il sistema, se trova un elemento associato al codice inserito, valorizza i campi dell'interfaccia con le informazioni relative all' utente.
- Il titolare, una volta modificati i dati, conferma l'operazione.
- Se i parametri inseriti sono corretti, il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo utente dovrà essere aggiornato nel sistema e mostrato correttamente nella tabella.

La Figura 3.47 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

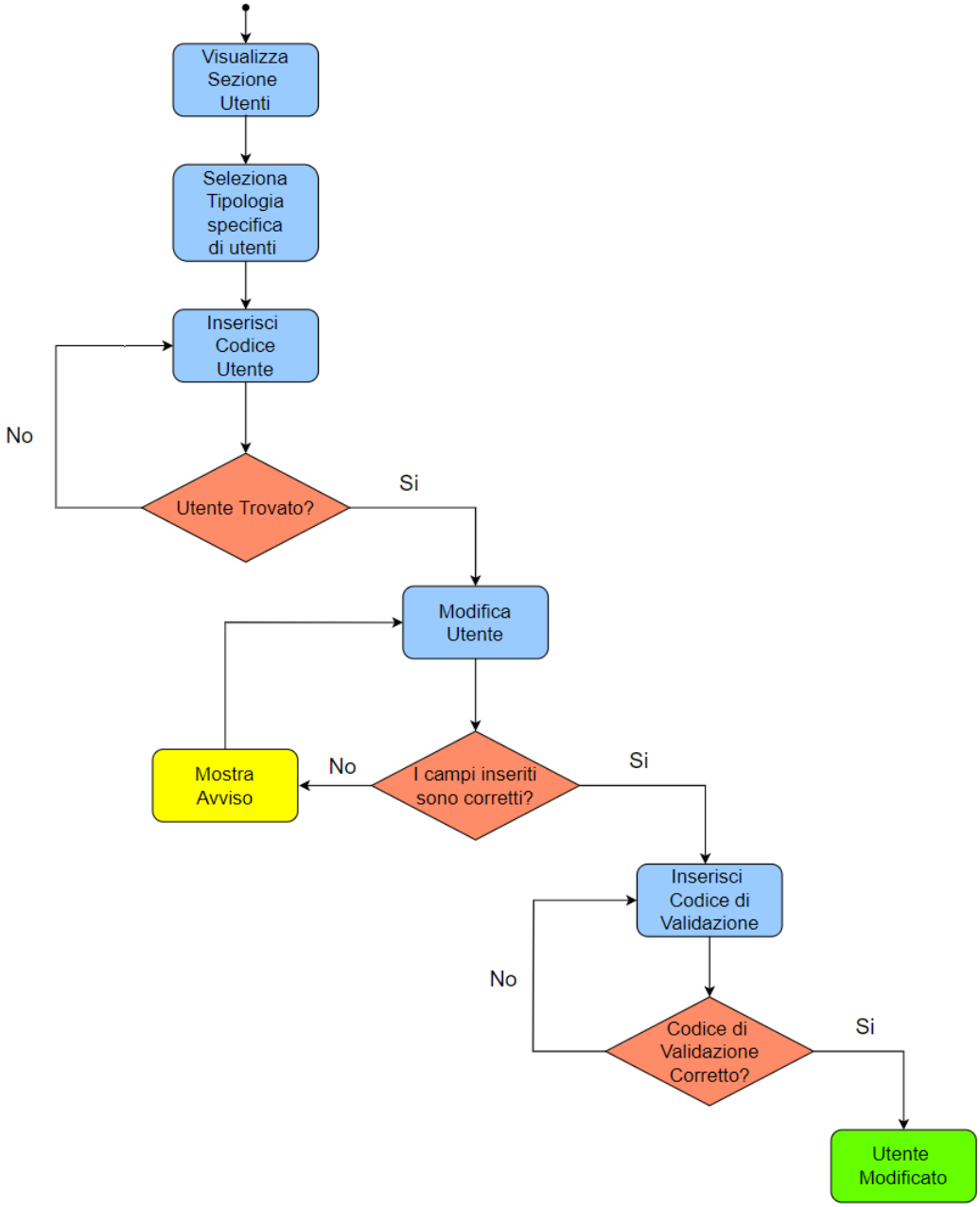


Figura 3.46: Sequenza di azioni relativa alla modifica di un utente presente nel sistema

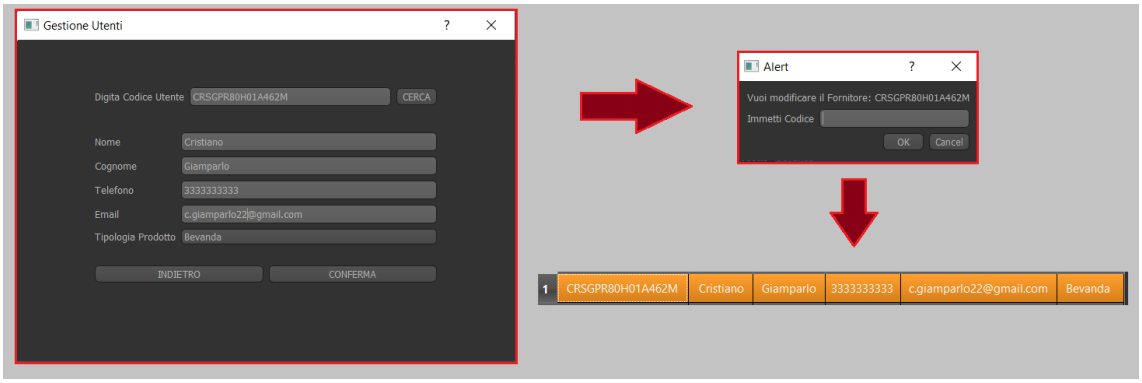


Figura 3.47: Modifica di un utente

Diagrammi delle sequenze associati

La Figura 3.48 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità.

Il titolare, premendo il tasto "modifica utente", richiama il metodo *ModifyUserView()* di *UsersSectionView()*, che manda in esecuzione un'istanza di *EditUserDialog()*; l'interfaccia risultante, descritta dal metodo *ModifyUi()*, permetterà di scegliere uno degli utenti e modificare i suoi dati.

La ricerca delle informazioni relative all'utente e il successivo aggiornamento delle stesse sarà gestito dalle classi *UsersSectionController()* e *UsersSectionModel()*.

3.4.4 Eliminazione di un utente presente le sistema

Logica degli eventi e descrizione grafica

Il diagramma, nella Figura 3.49, rappresenta il flusso di attività relativo all'eliminazione di un utente presente nel sistema. Esso può essere descritto come segue:

- Il titolare, che si trova nell'area utenti, in particolare nella sezione dedicata alla tipologia desiderata, seleziona la funzionalità per l'eliminazione di un utente presente nel sistema.
- Il sistema visualizza un'interfaccia, attraverso cui verrà richiesto di inserire il codice appartentente all'utente che si vuole eliminare.
- Il titolare conferma l'eliminazione.
- Il sistema, se trova un elemento associato al codice inserito, visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il nuovo utente dovrà essere eliminato dal sistema e rimosso correttamente nella tabella.

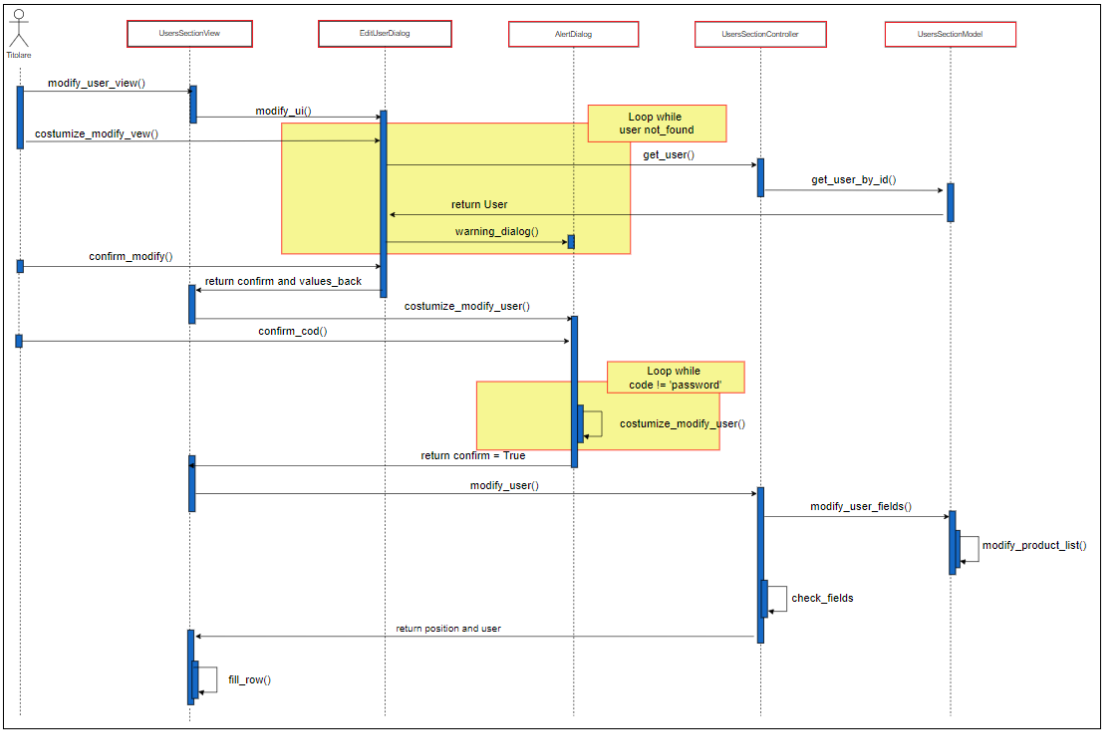


Figura 3.48: Diagramma delle sequenze relativo alla modifica di un utente esistente

La figura 3.50 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.51 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità. La ricerca e la successiva eliminazione dell'utente avvengono in modo analogo allo scenario relativo alla modifica.

Tuttavia, come nel caso dell'eliminazione di un prodotto, sarà possibile eseguire l'operazione soltant se l'utente, qualora sia un fornitore o un cliente, non sia associato ad ordini non pagati.

Il controllo di questa condizione è svolto dalla funzione *UserRefDeleted()*.

3.4.5 Pagamento di un dipendente presente nel sistema

Logica degli eventi e descrizione grafica

Il diagramma nella Figura 3.52 rappresenta il flusso di attività relativo al pagamento di uno dei dipendenti presenti nel sistema. Esso può essere descritto come di seguito specificato:

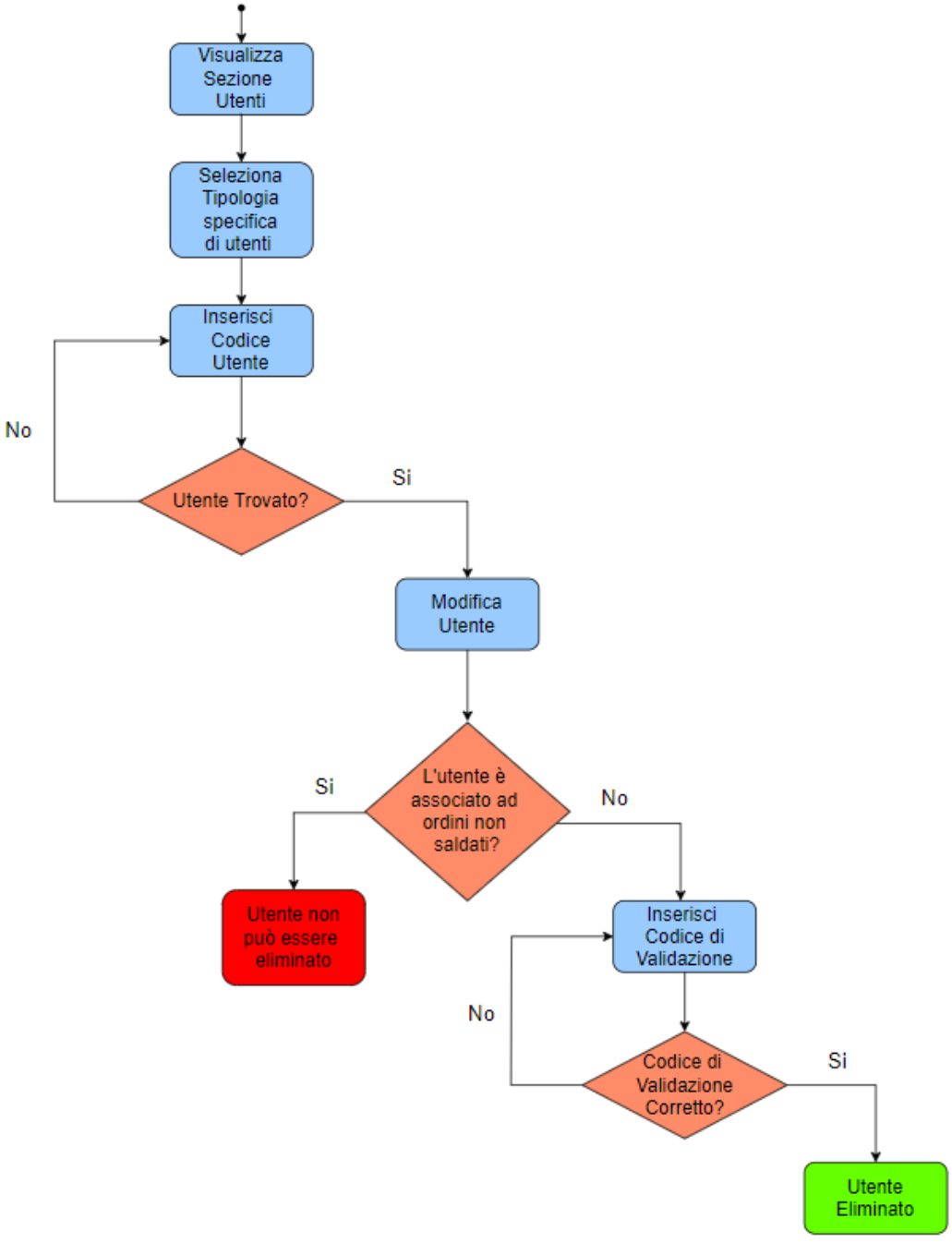


Figura 3.49: Eliminazione di un utente

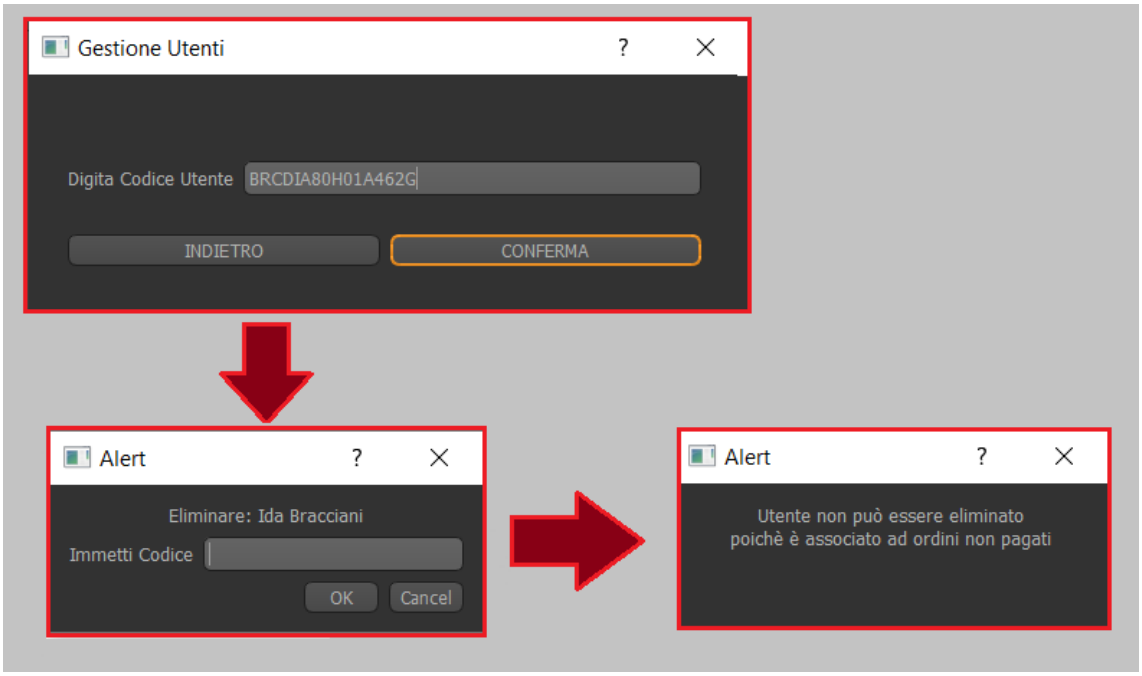


Figura 3.50: Eliminazione di un utente nel caso in cui sia associato ad un ordine non pagato

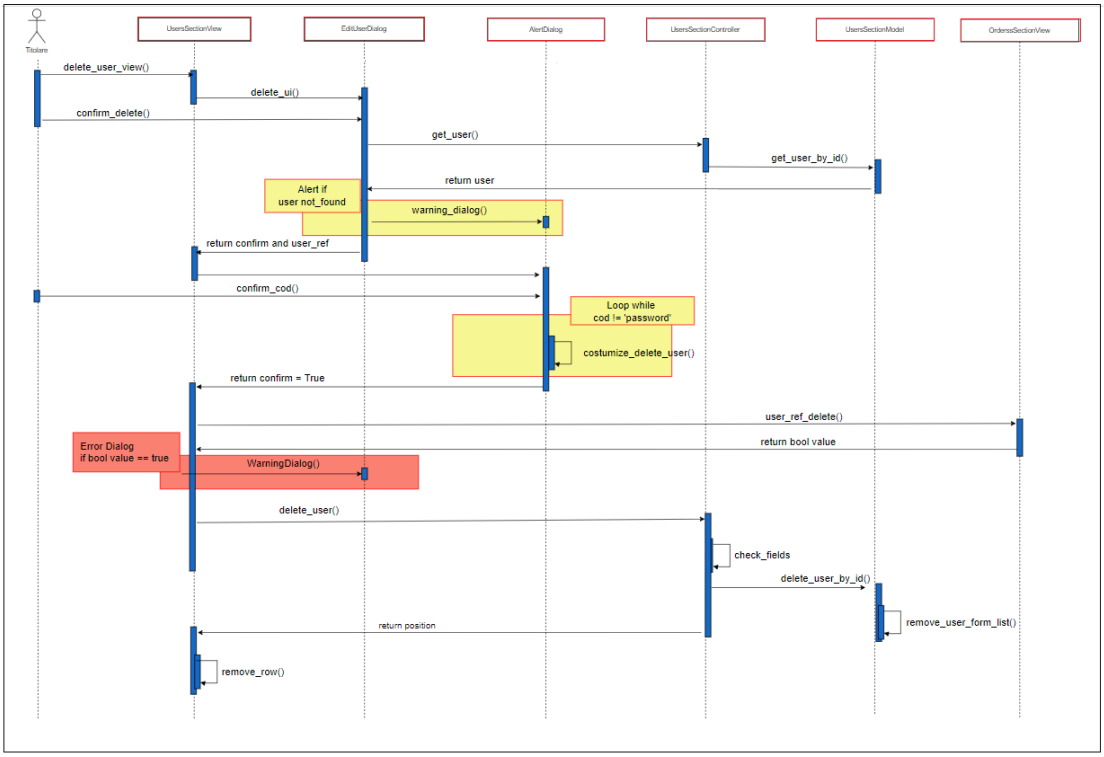


Figura 3.51: Diagramma delle sequenze relativo all'eliminazione di un utente esistente

- Il titolare, che si trova nell'area utenti, in particolare nella sezione dedicata ai dipendenti, seleziona la funzionalità per la modifica di uno dei dipendenti presenti nel sistema.
- Il sistema visualizza l'interfaccia di modifica dove, inizialmente, verrà richiesto di inserire il codice appartentente al dipendente che si vuole modificare.
- Il sistema, se trova un elemento associato al codice inserito, valorizza i campi dell'interfaccia con le informazioni relative all'utente.
- Il titolare seleziona l'opzione di pagamento (qualora la mensilità corrente non sia stata pagata).
- Il sistema visualizza un'interfaccia per la validazione dell'operazione.
- In caso di validazione, il campo del dipendente riguardante l'ultimo pagamento dovrà essere aggiornato; viene registrata una nuova transazione.

La Figura 3.53 descrive come dovranno essere realizzate graficamente le varie fasi di questo scenario.

Diagrammi delle sequenze associati

La Figura 3.54 mostra quali sono le interazioni tra gli oggetti del sistema coinvolti nella realizzazione di questa funzionalità, accessibile all'interno dell'interfaccia di modifica.

Se l'operazione viene validata, viene richiamata la funzione *AddTransaction()* per registrare la nuova transazione.

In seguito, *EmployeesSectionView()* richiama il metodo *ModifyUser()* della classe *EmployeesSectionController()*; quest'ultimo invoca le funzionalità della classeEmployeesSectionModel()) per aggiornare l'ultimo pagamento del dipendente.

3.5 Progettazione della sezione relativa alle Transazioni

L'area relativa alle trasazioni sarà organizzata come descritto nella Figura 3.55.

La descrizione si può effetturare individuando le due macrosezioni in cui è suddivisa la schermata:

- *Sezione Filtri*: si trova nella parte superiore dell'interfaccia, essa contiene i campi di inserimento per selezionare i parametri di la ricerca.
- *Tabella Transazioni*: disposta nella zona centrale, essa mostrerà le informazioni relative alle transazioni correntemente visualizzati.

Inoltre è presente un pulsante di switch per cambiare la tipologia di transazioni. Esse infatti possono riferirsi al pagamento di un ordine Oppure a quello di uno stipendio.

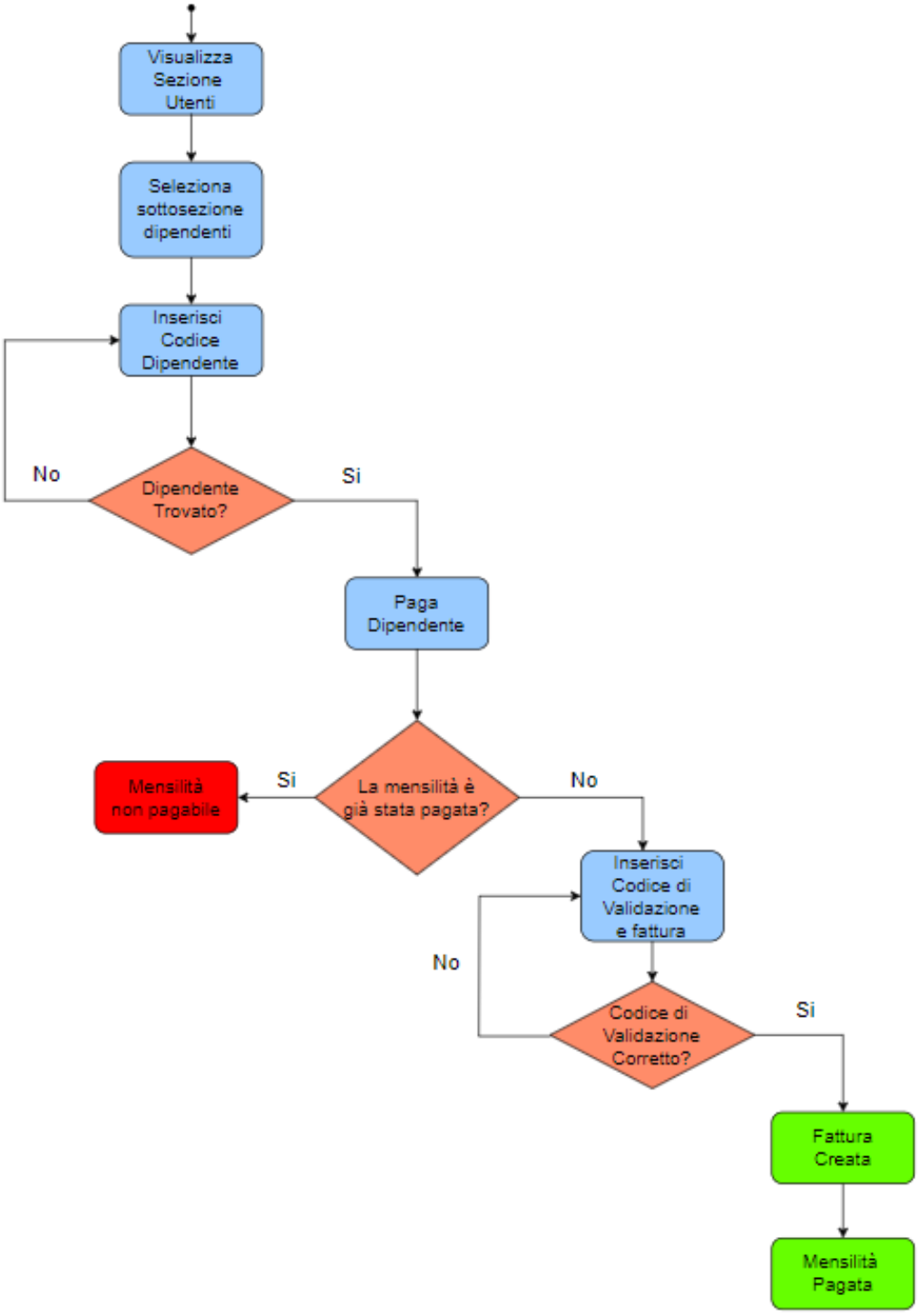


Figura 3.52: Pagamento di un dipendente

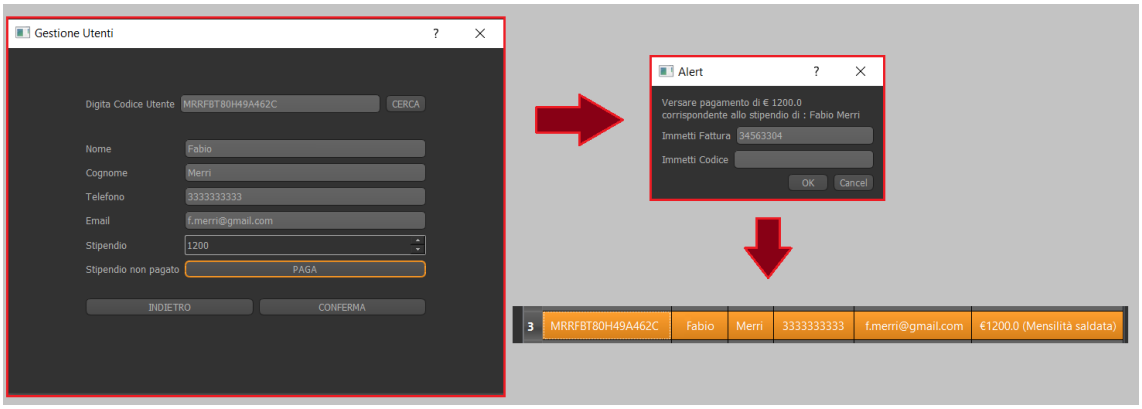


Figura 3.53: Eliminazione di un utente nel caso in cui sia associato ad un ordine non pagato

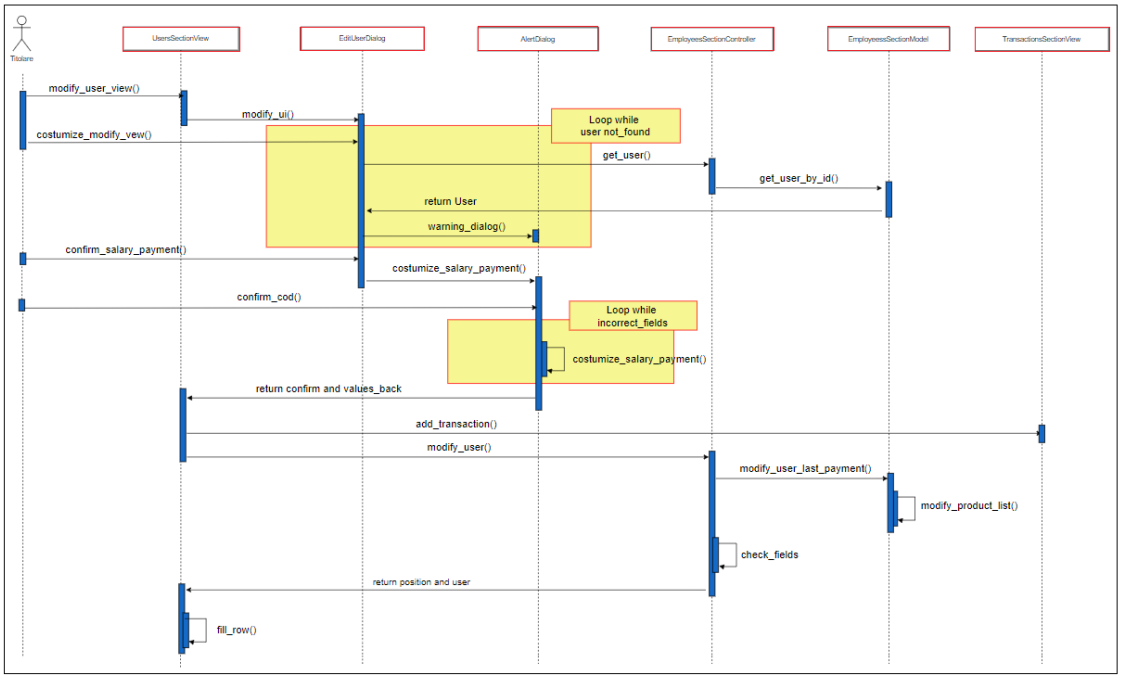


Figura 3.54: Diagramma delle sequenze relativo al pagamento di un Dipendente

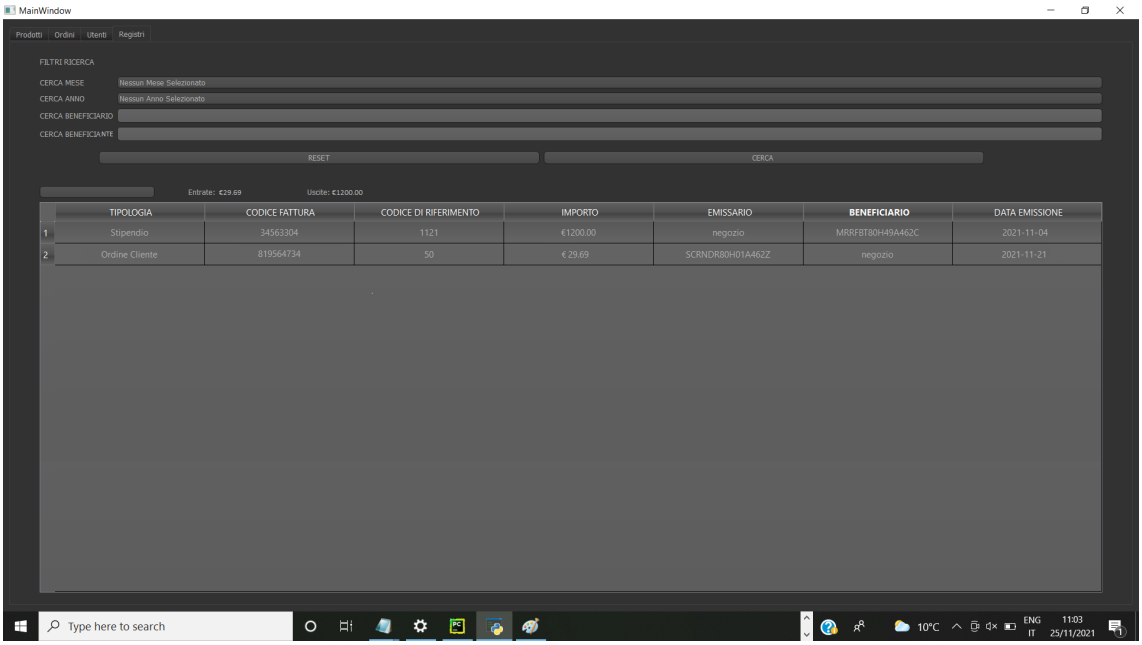


Figura 3.55: Sezione Transazioni

3.5.1 Ricerca delle Transazioni

L'unica funzionalità offerta direttamente da questa sezione è quella dei filtraggio degli ordini.

Questa operazione, la cui logica è analoga al filtraggio degli elementi delle altre sezioni, è riportata nella figura 3.56.

3.5.2 Aggiunta di una nuova transazione

Il diagramma nella Figura 3.57 rappresenta il flusso di attività relativo all'inserimento di una nuova transazione nel sistema.

Questo scenario si verifica ogni volta che viene saldato un ordine oppure quando si paga lo stipendio di un dipendente.

TransacionsSectioView() richiama il metodo *AddTransaction()* di *TransactionsSectionController()*, quest'ultimo invoca la funzionalità per aggiungere una nuova transazione della classe *TransactionsSectionModel()*.

TransactionsSectionController() controlla, attraverso il metodo *CheckFields()*, qualora siano stati applicati dei filtri, se la nuova transazione deve essere aggiunta alla tabella.

Il risultato del controllo viene restituito a *TransactionsSectionView()* che, in caso di esito positivo, aggiunge alla tabella delle transazioni il nuovo elemento creato.

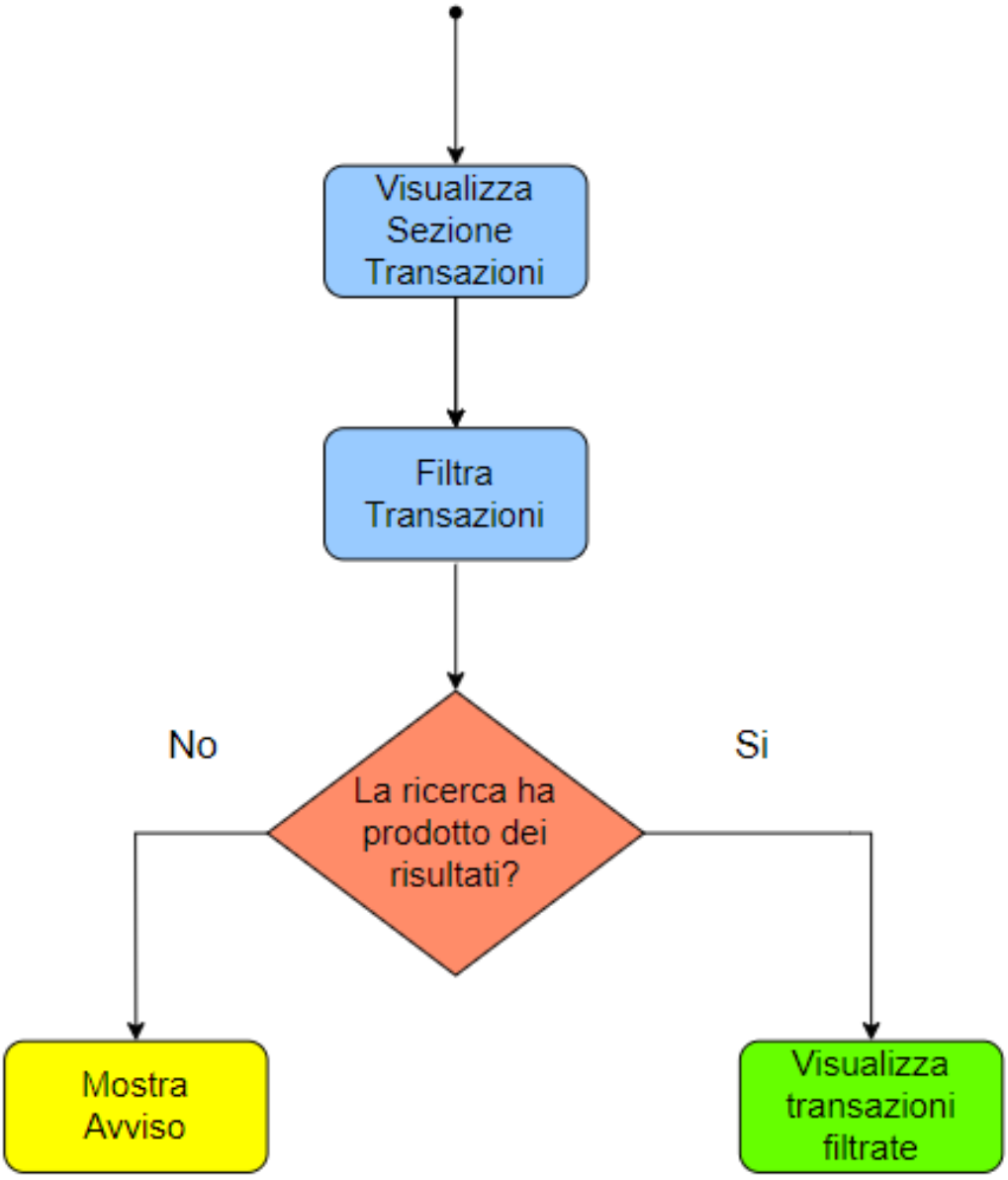


Figura 3.56: Sequenza di azioni associata filtraggio delle Transazioni

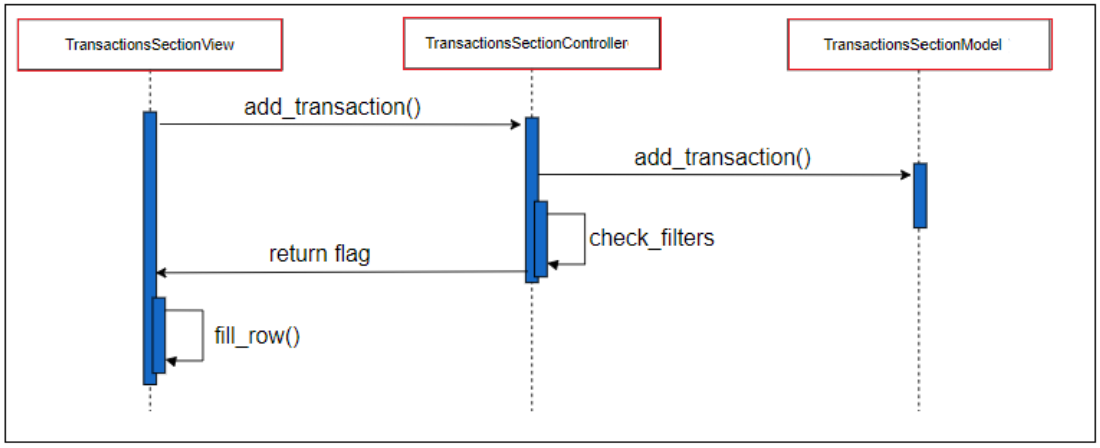


Figura 3.57: Creazione nuova transazione

3.6 Progettazione delle funzionalità di comunicazione tra i sottosistemi

In questa sezione saranno descritti quali sono i metodi che, quando vengono effettuate determinate operazioni da parte del titolare, permettono la comunicazione tra i vari sottosistemi del programma.

ReflectMethod()

Questa funzionalità, descritta nella Figura 3.58, ha il compito di aggiornare automaticamente la quantità dei prodotti in magazzino, qualora, a seguito di un'operazione effettuata nella sezione degli ordini, sia necessario.

ProductSectionView() delega le classi *EditProdDialog()*, *EditProdController()* e *EditProdModel()* all'operazione della modifica del prodotto.

ProductSectionController(), dopo aver modificato la lista dei prodotti all'interno di *ProductSectionModel()*, controlla, qualora siano stati applicati dei filtri, se il prodotto deve essere aggiornato nell'interfaccia.

ProductSectionView(), ricevuto il risultato del controllo, aggiorna, se necessario, le informazioni nella tabella.

UserRefDeleted()

Questa funzionalità, descritta nella Figura 3.59, ha il compito di controllare, qualora venga richiesta l'eliminazione di un cliente o di un fornitore, se uno degli ordini non pagati è associato all'utente in questione.

OrdersSectionView(), a seconda della tipologia dell'utente associato dell'ordine (cliente o fornitore), richiama il metodo *GetOrderList()* di uno dei due controller (*OrderCostumerSectionController()* o *OrderCostumerSectionController()*)

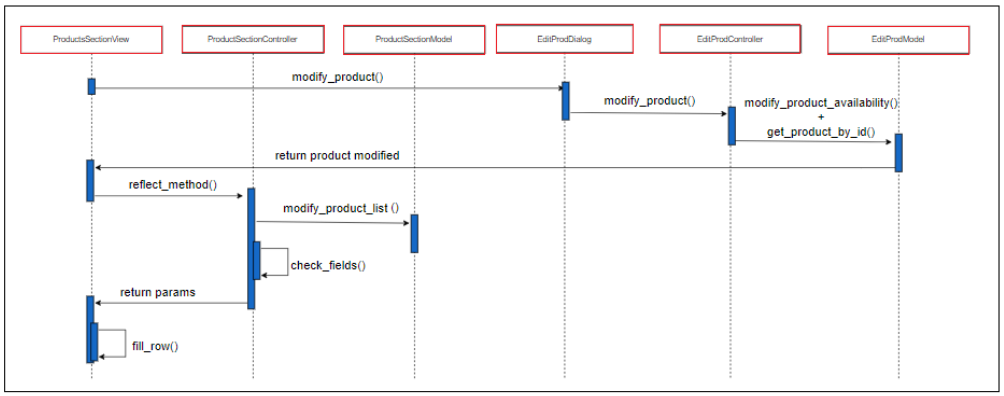


Figura 3.58: Aggiornamento dei prodotti

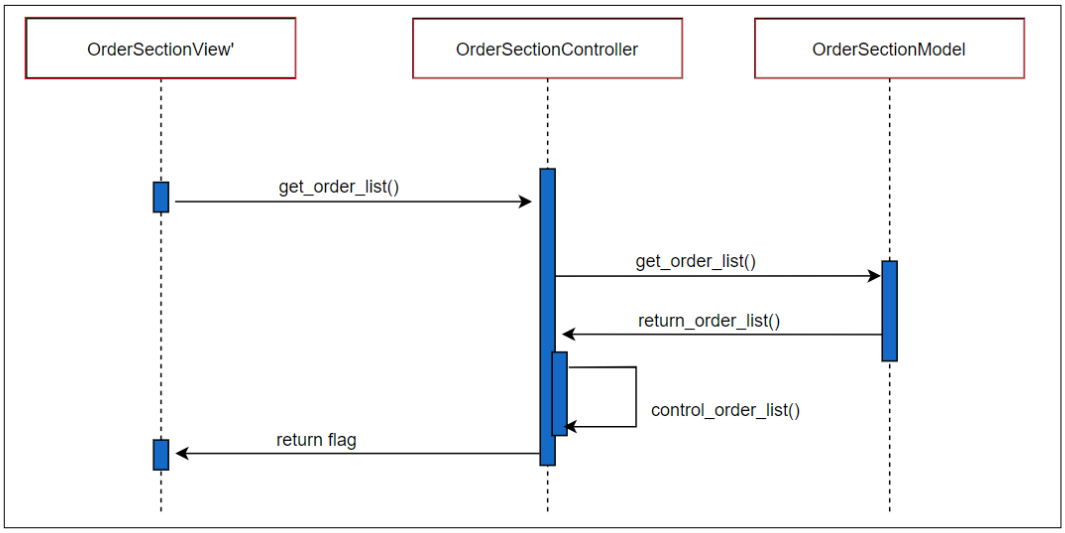


Figura 3.59: Controllo per l'eliminazione di un cliente o un fornitore

Il controller scelto recupera, dal model associato, la lista degli ordini. In seguito controlla se è presente almeno un ordine non pagato associato al cliente che si vuole eliminare.

Il risultato del controllo viene restituito a *OrdersSectionView()*.

OriginalProductDeleted()

Questa funzionalità, descritta nella Figura 3.60, ha il compito di controllare, qualora venga richiesta l'eliminazione di un prodotto, se uno degli ordini non pagati contiene l'elemento in questione. *OrdersSectionView()* recupera , mediante il metodo *GetOrderList()*, la lista degli ordini relativi ai clienti e ai fornitori dai rispettivi model e controller.

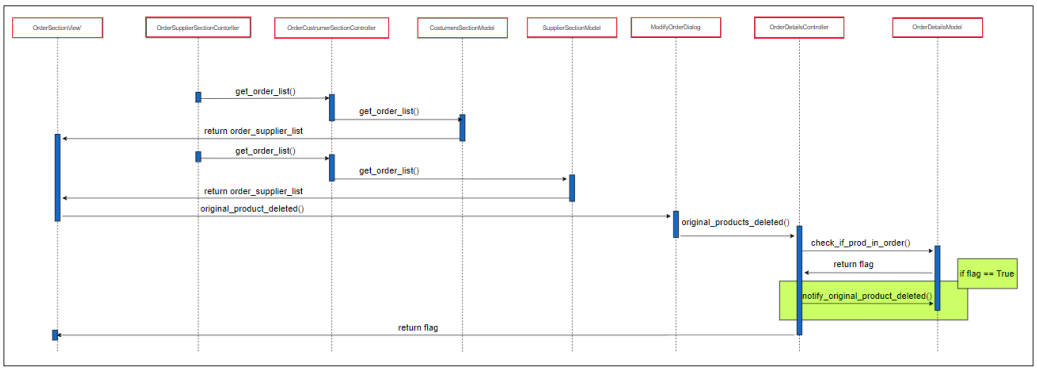


Figura 3.60: Controllo per l'eliminazione di un prodotto

In seguito, viene richiamato il metodo *OriginalProductDeleted()* della classe *ModifyOrderController()*.

ModifyOrderController() controlla, mediante il metodo *CheckIfProdInOrder()*, se il prodotto è associato ad un ordine non pagato.

Qualora venga trovato almeno un prodotto che soddisfi la precedente condizione, viene ritornato un valore che indica l'impossibilità di cancellare il prodotto.

Nel caso contrario, invece, i prodotti, negli ordini saldati, che si riferiscono all'elemento che si vuole eliminare, vengono modificati per indicare la rimozione del prodotto originale (*NotifyOriginalProductDeleted()*); successivamente viene permessa l'eliminazione.

Capitolo 4

Implementazione

Nel presente paragrafo saranno mostrati gli strumenti software e le strategie di programmazione utilizzati nella realizzazione del programma. L'analisi riguarda sia lo sviluppo delle interfacce e della logica degli eventi, ovvero il front-end del programma, sia l'organizzazione e la gestione dei dati, ovvero il back-end.

4.1 Python

I motivi per cui si è scelto di utilizzare questo linguaggio di programmazione sono molteplici; le caratteristiche di *Python*, infatti, si prestano perfettamente per lo sviluppo di applicazioni desktop.

Innanzitutto, si tratta di un linguaggio completo; le librerie fornite contengono tutti i moduli necessari per lo sviluppo del back-end e del front-end del nostro software, senza bisogno di integrazioni aggiuntive.

Inoltre, le variabili di *Python* non sono tipizzate; siccome la memoria viene allocata in modo dinamico, è possibile assegnare alla stessa variabile, durante l'esecuzione, tipi di dati diversi, senza dover necessariamente definirla a priori.

La proprietà appena descritta si adatta perfettamente a quelle che sono le caratteristiche del nostro prgoramma; i componenti del sistema, infatti, presentano molte analogie tra le singole funzionalità e, più in generale, tra le rispettive strutture.

Accade, dunque, che sia necessario assegnare alla stessa variabile tipi di valori diversi, a seconda dello stato del sistema durante l'esecuzione.

Tale caratteristica, unità all'alto livello di astrazione con cui lavora *Python*, permette di realizzare un codice snello e facilmente riutilizzabile.
Come ambiente di sviluppo è stata utilizzata la versione community di *Pycharm*. La scelta di quest'IDE è dovuta ai diversi strumenti di cui è dotato. Le funzionalità offerte sono, infatti, le seguenti:

- Assistenza e analisi della codifica, con completamento del codice ed evidenziazione degli errori.
- Navigazione nel progetto e nel codice, viste del progetto specializzate, viste della struttura dei docomenti e salti rapidi tra file, classi, metodi e utilizzi.
- Debugger visivo per visualizzare i dati di debug in fase di esecuzione.

```
def init_db(self):
    self.mydb = mysql.connector.connect(
    host='localhost',
    user='root',
    password='11910112',
    port='3306',
    database='db_ing_soft'
    )
    self.mycursor = self.mydb.cursor()
```

Figura 4.1: Creazione dell'oggetto connessione, i parametri passati al cotruttore riguardano le informazioni relative al nostro database

- Possibilità di effettuare test delle singole componenti del software.

4.2 Gestione dei dati

Per la creazione e l'amministrazione dei dati, si è scelto di utilizzare il software MySQL Workbench.

Pyhton offre i moduli necessari per definire una modalità di accesso ai database. La Figura 4.1 mostra come è stato realizzato il collegamento.

L'accesso è reso possibile mediante l'utilizzo di oggetti Connessione. I costruttori, con cui vengono definiti, accettano un certo numero di parametri, in dipendenza dal singolo database.

La variabile ritornata, *mycursor*, è l'oggetto cursore, esso permette di eseguire le varie query e leggerne i risultati.

La Figura 4.2 mostra la funzione per la creazione del modello relazionale associato ai prodotti, vengono definiti i tipi degli attributo e la chiave primaria dell'entità in questione.

Nelle Figure 4.3 e 4.4, invece, viene mostrato come le varie classi, adibite alla gestione dei dati, contengono i metodi nei quali vengono implementate le query.

Le funzioni appena introdotte mostrano l'utilizzo dei principali metodi messi a disposizione dalle API di *Python*:

- *execute(params)*: esegue i comandi e le istruzioni SQL passati sotto formato di stringa.
- *fetchone()*: restituisce il primo risultato della query.
- *fetchall()*: resistuisce tutti i risultati della query.
- *commit()*: serve a salvare e rendere operative le modifiche al database.

```
def create_product_table(self, cod):
    if cod == 2308:
        command ='CREATE TABLE products (id INT NOT NULL AUTO_INCREMENT, ' \
                                        'name VARCHAR(45) NOT NULL, ' \
                                        'type VARCHAR(45) NOT NULL, ' \
                                        'price DOUBLE NOT NULL, ' \
                                        'discount DOUBLE NOT NULL, ' \
                                        'availability INT NOT NULL, ' \
                                        'PRIMARY KEY (id));'
        self.mycursor.execute(command)
        self.mydb.commit()
```

Figura 4.2: Creazione della tabella dei prodotti

```
def get_product_by_id(self, id):
    command = "SELECT * FROM products WHERE id = %s"
    val = (str(id),)
    self.db.mycursor.execute(command, val)
    myresult = self.db.mycursor.fetchone()
    if myresult == None:
        return Product("none", "none", "none", "none", "none")
    else:
        prod = Product(myresult[1], myresult[2], myresult[3], myresult[4], myresult[5])
        prod.set_id(myresult[0])
        return prod

def get_all_products(self):
    self.product_list = []
    command = 'SELECT * FROM products'
    self.db.mycursor.execute(command)
    myresult = self.db.mycursor.fetchall()
    for product in myresult:
        prod = Product(product[1], product[2], product[3], product[4], product[5])
        prod.set_id(product[0])
        self.product_list.append(prod)
```

Figura 4.3: Metodi per lettura della tabella prodotti

```
def add_product(self, product):
    command = 'INSERT INTO products (name, type, price, discount, availability) VALUES (%s, %s, %s, %s, %s)'
    val = (str(product.name), str(product.type), str(product.price), str(product.discount), str(product.availability))
    self.db.mycursor.execute(command, val)
    self.db.mydb.commit()
    return self.db.mycursor.lastrowid

def modify_product_name(self, id, new_name):
    command = 'UPDATE products SET name = %s WHERE id = %s'
    val = (str(new_name), str(id))
    self.db.mycursor.execute(command, val)
    self.db.mydb.commit()

def delete_product_by_id(self, id):
    command = "DELETE FROM products WHERE id = %s"
    val = (str(id),)
    self.db.mycursor.execute(command, val)
    self.db.mydb.commit()
```

Figura 4.4: Metodi di scrittura sulla tabella prodotti

4.3 Implementazione Grafica

Il front-end dell'applicazione verrà realizzato mediante *PyQt5*, un plug-in *Python*, esso fornisce un insieme di classi che consentono l'accesso alle librerie grafiche Qt.

Qt è toolkit GUI multipiattaforma implementato in *C++*. Esso è dedicato alla creazione di interfacce grafiche che utilizzano lo stile nativo della piattaforma in uso, o, in alternativa, lo stile definito dall'utente mediante l'uso di stylesheet che adottano una sintassi simile a quella dei CSS.

Per lo sviluppo è stato utilizzato il tool grafico *QtDesigner*; esso permette di strutturare e personalizzare le GUI del programma utilizzando gli strumenti messi a disposizione da *Qt*.

Tutte le proprietà e le componenti dell'interfaccia impostate in Qt Designer possono essere modificate dinamicamente all'interno del codice; i moduli di *Python* permettono di convertire i file generati da *QtDesigner* dal formato *'.ui'* a quello *'.py'*

Inoltre, il comportamento degli elementi grafici viene definito utilizzando il meccanismo di segnali di *Qt*.

4.4 Organizzazione del Codice

Nella Figura 4.5 è riportata la gerarchia dei file che costituiscono il programma; ciascuno contiene una determinata classe che ne definisce il nome.

All'avvio dell'applicazione viene mandata in esecuzione un'istanza di *MainGui()*, essa, a sua volta, come mostrato in Figura 4.6, inizializza le view delle quattro sezioni principali, passando loro i parametri relativi a stile e dimensione, definendo quali sono le funzioni per la comunicazione tra le stesse; infine le collega ad un oggetto *QTabWidget*.

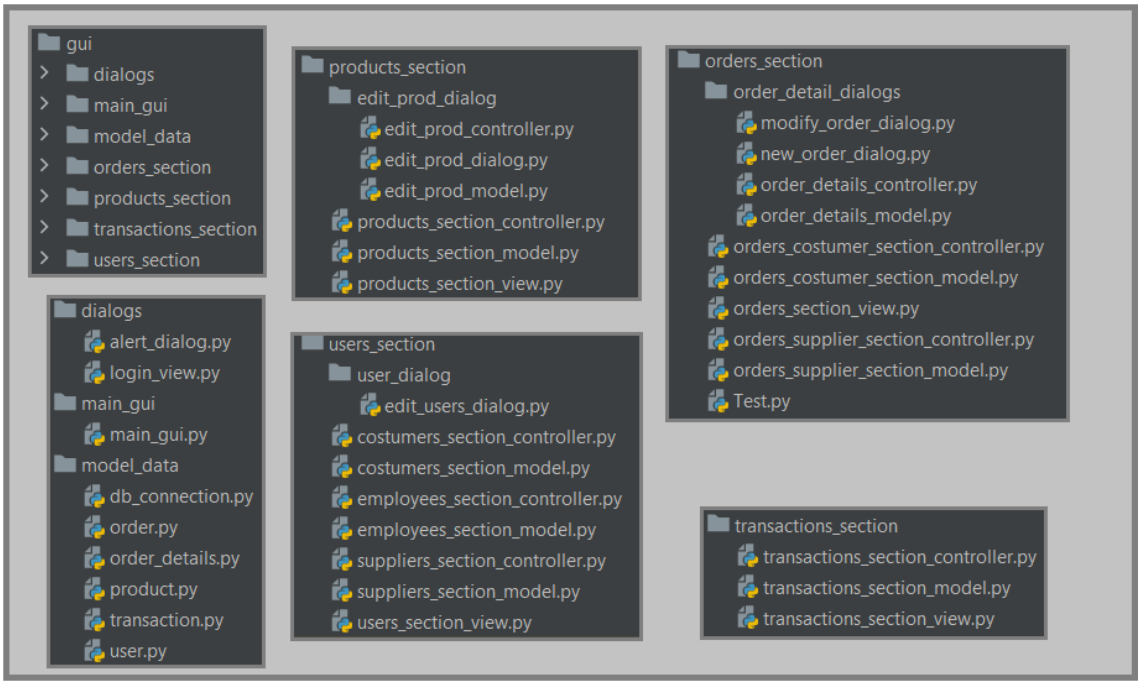


Figura 4.5: Gerarchia dei file

```
def setup_application(self):
    product_section_view = ProductsSectionView(self.style, self.w, self.h)
    order_section_view = OrdersSectionView(self.style, self.w, self.h)
    user_section_view = UserSectionView(self.style, self.w, self.h)
    transaction_section_view = TransactionsSectionView(self.style, self.w, self.h)

    product_section_view.order_detail_deleted = order_section_view.original_product_deleted
    order_section_view.reflect_method = product_section_view.reflect_method
    order_section_view.import_list_products = product_section_view.export_list_products
    order_section_view.import_list_users = user_section_view.export_users_list
    order_section_view.import_user_type = user_section_view.export_user_type
    order_section_view.add_transaction = transaction_section_view.add_transaction
    user_section_view.add_transaction = transaction_section_view.add_transaction
    user_section_view.user_ref_deleted = order_section_view.user_ref_deleted

    product_section_view.setup_product_section()
    order_section_view.setup_order_section()
    user_section_view.setup_user_section()
    transaction_section_view.setup_transaction_section()

    self.setup_ui(product_section_view, order_section_view, user_section_view, transaction_section_view)
    self.showMaximized()
```

Figura 4.6: Il setup delle funzionalità dell'applicazione

```python
def setup_order_section(self):
    self.setup_filter_section()
    self.setup_sub_filter_section()
    self.setup_table_section()
    self.filter_section_layout = QtWidgets.QVBoxLayout()
    self.filter_layout = QtWidgets.QHBoxLayout()
    self.filter_layout.addLayout(self.vertical_filter_layout)
    self.filter_layout.addLayout(self.subfilters_layout)
    self.filter_section_layout.addLayout(self.filter_layout)
    self.filter_section_layout.addWidget(self.button_reset)
    self.filter_section_layout.setContentsMargins(0, 0, 200, 20)
    self.layout = QtWidgets.QVBoxLayout()
    self.layout.addLayout(self.filter_section_layout)
    self.layout.addLayout(self.vertical_layout)
    self.layout.setContentsMargins(50, 25, 50, 20)
    self.setLayout(self.layout)
    self.retranslate_ui()
    self.init_order_section(True)
```

```python
def init_order_section(self, flag):
    if flag:
        self.controller = OrdersCostumerSectionController(self.import_list_users,
                                                           self.import_list_products,
                                                           self.reflect_method,
                                                           self.add_transaction)
        self.define_events()
    self.reset_filter_section()
    self.fill_table(self.controller.list_resources)
    cod_list = []
    j = 1
    for order in self.controller.list_resources:
        if order.cod_user not in cod_list:
            cod_list.append(order.cod_user)
    for elem in cod_list:
        self.edit_cod.addItem("")
        self.edit_cod.setItemText(j, elem)
        j += 1
```

Figura 4.7: Creazione delle interfacce per la scelta dell'ordine e la visualizzazione dei relativi dettagli

4.5 Esempio di implementazione

Come è stato ribadito in precedenza, sono molte le analogie funzionali e strutturali dei diversi insiemi di Model-View-Controller. Il pattern utilizzato, infatti, definisce delle regole implementative ben precise permettendo, di conseguenza, di ottenere un codice uniforme e organizzato.

Anche la comunicazione e lo scambio di dati tra le varie view con i propri dialog e tra le view stesse sono stati realizzati seguendo lo stesso modus operandi.

Per queste ragioni, al posto di illustrare ogni singola funzionalità del programma, si è deciso di descrivere unicamente l'implementazione della modifica di un ordine associato ad un cliente.

Questa funzionalità, infatti, è la più completa tra quelle sviluppate, nonchè quella che coinvolge il maggior numero di dati e informazioni.

In particolare, l'aggiornamento di un ordine di un cliente, a differenza di uno associato ad un fornitore, implica la modifica della quantità dei prodotti in magazzino.

La Figura 4.7 ha il compito di contestualizzare il codice che verrà descritto in seguito.

Recupero dell'ordine dal database e creazione dell'interfaccia di visualizzazione

Il metodo *SetupOrderSection()*, dopo aver inizializzato i layout e la struttura grafica dell'interfaccia, richiama *InitOrderSection()*.

Tra le varie operazioni eseguite dalla seconda funzione, le più importanti sono: l'inizializzazione della variabile *controller*, a cui vengono passati dei metodi supplementari, e l'invocazione del metodo *DefineEvents()*, il quale definisce quali eventi devono essere invocati a seconda degli input ricevuti.
Le operazioni preliminari di questo scenario, riportate nella Figura 4.8, sono eseguite da *SelectOrderView()*; viene mandata in esecuzione un'istanza di *AlertDialog()*, l'interfaccia mostrata permette di digitare il codice dell'ordine da modificare.

```
def select_order_view(self):
    wait = True
    i = 0
    while wait:
        alert_dialog = AlertDialog(self.style)
        alert_dialog.costumize_select_order(i)
        alert_dialog.exec_()
        if alert_dialog.confirm:
            o = self.controller.get_order_by_id(int(alert_dialog.selected_cod))
            if not (o == "none"):
                order = [o.get_nbill(),
                         o.date_emission,
                         o.date_payment,
                         o.cod_user,
                         o.payment_settled,
                         o.amount_due]
                wait = False
                break
            i += 1
        else:
            wait = False

    if not wait and alert_dialog.confirm:
        self.edit_order_details_view(order)
```

```
def edit_order_details_view(self, order):
    if self.type_user_flag:
        self.controller.get_support_list(str(self.controller.import_user_type(order[3])))
    else:
        self.controller.get_support_list()
    self.dialog_view = ModifyOrderDialog(self.style, self.w, self.h)
    self.dialog_view.create_show_order_view(self.controller.product_list, order, self.type_user_flag)
    self.dialog_view.exec_()
    if not(self.dialog_view.confirm == "none"):
        position = self.controller.modify_order(self.dialog_view.confirm, self.dialog_view.values_modified)
        if self.dialog_view.confirm == "delete":
            self.order_table.removeRow(position)
        elif self.dialog_view.confirm == "edit" or self.dialog_view.confirm == "pay":
            self.fill_row(self.controller.list_resources[position], position, True)
```

Figura 4.8: Metodi per mandare in esecuzione le interfacce di selezione e di visualizzazione dell'ordine

L'input ricevuto viene elaborato dal metodo *GetOrderById()* del controller, i dati dell'ordine trovato vengono inseriti in una lista; questa viene passata come parametro alla funzione *EditOrderDetail()*.

Nel secondo metodo viene creato un oggetto della classe *ModifyOrderDialog()*, il quale, mediante *CreateShowOrderView()*, descritto nella Figura 4.9, inizializza le proprie funzionalita e definisce, nella funzione *ConnectViewMode()*, gli eventi relativi alle interazioni del titolare con l'interfaccia.

Inizializzazione dei parametri necessari alla funzionalità di modifica

Nella Figura 4.10 viene mostrata la funzione *InitModifyOrderSection()*, essa serve ad adattare l'oggetto della classe *OrderDetailsController()*, istanziato in precendenza,

```
def create_show_order_view(self, product_list, order, type_user_flag):
    self.controller = controller
    self.confirm = "none"
    self.type_user_flag = type_user_flag
    self.controller = OrderDetailsController()
    self.controller.init_modify_order_section(product_list, order, type_user_flag)
    self.create_ui(False)
    self.fill_view_fields(False, self.order, self.controller.count)
    self.date_edit.setText(str(self.order[1]))
    self.fill_table(False, self.controller.list_support, len(self.controller.model_detail.list_order_details))
    self.connect_view_mode()
```

```
def connect_view_mode(self):
    self.confirm_button.clicked.connect(lambda: self.go_to_edit_view())
    self.delete_button.clicked.connect(lambda: self.confirm_delete_order())
    self.cancel_button.clicked.connect(lambda: self.close())
```

Figura 4.9: Definizione delle caratteristiche principali relative all'interfaccia di visualizzazione dei prodotti associata all'ordine

alle funzionalità di visualizzazione e modifica dell'ordine. La firma del metodo identifica quali sono i parametri ricevuti: la lista di tutti i prodotti, le informazioni relative all'ordine e un valore per indicare se l'utente associato è un cliente o un fornitore. In particolare, il controller recupera dal proprio model *ListOrderDetails*, la lista contenente i prodotti associati all'ordine, e, mediante un ciclo for, valorizza l'oggetto *ListSupport*.

Ogni elemento della nuova lista corrisponde ad un dizionario in cui sono presenti le informazioni riguardanti i prodotti.

Tuttavia, possiamo considerare questa struttura dati come l'unione di due struttre formalmente uguali, ma che si riferiscono a classi di dati diverse.

I primi n elementi si riferiscono agli n prodotti associati all'ordine, i restanti indicano i prodotti che sono presenti in negozio, ma che non sono stati selezionati in fase di creazione.

Le coppie chiave-valore forniscono le informazioni sul prodotto associato; per ciascun elemento sono riportati l'id, il prezzo, il nome, la disponibilità in negozio e la quantità originariamente ordinata (nulla nel caso in cui il prodotto non è associato all'ordine). Inoltre, vengono indicati altri tre campi, i quali, qualora vengano confermate le modifiche, indicheranno, rispettivamente, se il prodotto è presente nell'ordine, se è stato modificato e qual è l'ultimo valore scelto per la quantità da ordinare.

Gestione dell'interfaccia di modifica

Nel momento in cui il titolare, che sta visualizzando i dettagli dell'ordine, decide di passare alla modalità di modifica, viene richiamata la funzione *GoToEditView()*, descritta nella Figura 4.11. Viene mandata in esecuzione una seconda istanza

```
def init_modify_order_section(self, product_list, order, type_user_flag):
    self.list_original_products = product_list
    self.count = 0
    self.confirm = "none"
    self.order = order
    self.type_user_flag = type_user_flag
    if self.type_user_flag:
        self.type_order = "supplier"
    else:
        self.type_order = "costumer"
    self.nbill = self.order[0]
    self.count = self.order[5]
    self.model_detail.get_order_details_by_id(self.nbill, self.type_order)
    self.list_support = []
    for e in self.model_detail.list_order_details:
        for p in self.list_original_products:
            if e.id_prod == p[0]:
                name = p[1]
                ava = p[3]
                self.list_support.append({"id": e.id_prod, "price": round(e.price, 2),
                                          "name": name, "ava_prod": ava, "last_value_ava_modify": e.availability,
                                          "original_ava": e.availability, "modified": False, "removed": False})
                self.list_original_products.remove(p)

    self.list_support.append("")
    for p in self.list_original_products:
        self.list_support.append({"id": p[0], "price": p[2], "name": p[1], "ava_prod": p[3],
                                  "last_value_ava_modify": 1, "original_ava": 0, "modified": False, "removed": True})
```

Figura 4.10: Inizializzazione del controller per la gestione di un ordine presente nel sistema

```
def go_to_edit_view(self):
    if not (self.order[4]):
        self.edit_view = ModifyOrderDialog(self.style, self.w, self.h)
        self.edit_view.create_edit_order_view(self, self.order, self.type_user_flag, self.controller)
        self.edit_view.exec_()
        if not(self.edit_view.confirm =="none"):
            self.confirm = self.edit_view.confirm
            self.values_modified = self.edit_view.values_modified
            self.close()
    else:
        alert_dialog = AlertDialog(self.style)
        alert_dialog.warning_dialog("L'Ordine è stato saldato, impossibile Modificare")
        alert_dialog.exec_()
```

Figura 4.11: Funzione per passare alla modalità di modifica dell'ordine

di *ModifyOrderDialog()*, le cui funzionalità, stavolta, sono indicate all'interno di *CreateEditOrderView()*.

La Figura 4.12 mostra la funzione appena citata e il metodo per definire gli eventi dell'interfaccia.

Le azioni del titolare con la tabella di modifica saranno accettate dalla funzione *EventModifyViewTable()*, rappresentata nella Figura 4.13, e, successivamente, elaborate nel controller da *EventModifyViewTable()*, che indica quale azione deve essere eseguita, e *ActionModifyViewTable()*, che produce un determinato risultato.

```
def create_edit_order_view(self, show_view, order, type_user_flag, controller):
    self.controller = controller
    self.confirm = "none"
    self.type_user_flag = type_user_flag
    self.show_view = show_view
    self.order = order
    self.create_ui(True)
    self.fill_view_fields(True, self.order, self.controller.count)
    self.date_edit.setDate(QtCore.QDate.fromString(str(self.order[1]), 'yyyy-MM-dd'))
    current_date = today().strftime("%Y/%m/%d").replace('/', '-')
    self.date_edit.setDateRange(QtCore.QDate.fromString(str(current_date), 'yyyy-MM-dd'),
                                QtCore.QDate.fromString("2023-12-31", 'yyyy-MM-dd'))
    self.fill_table(True, self.controller.list_support, len(self.controller.model_detail.list_order_details))
    self.connect_edit_mode()
```

```
def connect_edit_mode(self):
    self.table_order_detail.clicked.connect(lambda: self.event_modify_view_table())
    self.confirm_button.clicked.connect(lambda: self.confirm_modify_order())
    self.delete_button.clicked.connect(lambda: self.confirm_delete_order())
    self.cancel_button.clicked.connect(lambda: self.close())
    self.change_state.clicked.connect(lambda: self.payment_order())
```

Figura 4.12: Definizione delle caratteristiche principali relative all'interfaccia di modifica dei prodotti associata all'ordine

```
def event_modify_view_table(self):
    new_ava = self.table_order_detail.cellWidget(self.table_order_detail.currentRow(), 2).value()
    value_back = self.controller.event_modify_view_table(self.table_order_detail.currentRow(),
                                                         self.table_order_detail.currentColumn(), new_ava)
    if not(value_back[0] == 'none'):
        self.fill_row(value_back[0], self.table_order_detail.currentRow(), True,
                      len(self.controller.model_detail.list_order_details))
        if value_back[1]:
            self.label_price.setText("PREZZO TOTALE: €" + str(round(self.controller.count, 2)))
```

Figura 4.13: Definizione degli eventi associati alle interazioni del titolare con la tabella dei prodotti

Le funzioni di *OrderDetailsController()* sono mostrate nella Figura 4.14. Le operazioni svolte da questi metodi riguardano l'aggiunta di nuovi prodotti alla lista *ListSupport* e la modifica o la rimozione degli elementi già presenti.

Aggiornamento del database

La funzione *ConfirmModifyOrder()* della classe *OrderDetailsController()*, nella Figura 4.15, dopo aver ricevuto la conferma riguardo la validazione dell'operazione, richiama il metodo *ModifyOrderDetails()* del controller, riportato nella Figura 4.16.

I campi "removed" e "modified" degli elementi di *ListSupport* vengono esaminati in un ciclo for; le condizioni che si possono verificare sono tre:

1. L'elemento, originariamente ordinaro, è stato rimosso; esso viene eliminato, mediante il metodo *DeleteOrderDetails()*, dall'elenco dei prodotti associati all'ordine.

```python
def event_modify_view_table(self, current_row, current_column, new_value_ava=0):
    if not (current_row == len(self.model_detail.list_order_details)):
        if self.list_support[current_row].get("removed"):
            if current_column == 4:
                return self.action_modify_view_table("add", current_row, new_value_ava)
        else:
            if current_column == 3:
                return self.action_modify_view_table("mod", current_row, new_value_ava)
            elif current_column == 4:
                return self.action_modify_view_table("del", current_row)
    else:
        return ["none"]
```

```python
def action_modify_view_table(self, act, row, new_value_ava=0):
    if act == "add":
        self.list_support[row].update({"last_value_ava_modify": new_value_ava})
        self.count += self.list_support[row].get("price") * self.list_support[row].get("last_value_ava_modify")
        self.list_support[row].update({"removed": False})
        return[self.list_support[row], True]

    elif act == "mod":
        if not (new_value_ava == self.list_support[row].get("last_value_ava_modify")):
            new_ava_diff = new_value_ava - self.list_support[row].get("last_value_ava_modify")
            self.list_support[row].update({"last_value_ava_modify": new_value_ava})
            self.count += self.list_support[row].get("price") * new_ava_diff
            count_change = False
        else:
            count_change = True

        if new_value_ava == self.list_support[row].get("original_ava"):
            self.list_support[row].update({"modified": False})
        else:
            self.list_support[row].update({"modified": True})

        return [self.list_support[row], count_change]

    elif act == "del":
        self.count -= self.list_support[row].get("price") * self.list_support[row].get("last_value_ava_modify")
        self.list_support[row].update({"removed": True})
        return [self.list_support[row], True]
```

Figura 4.14: Implementazione degli eventi associati alle interazioni del titolare con la tabella dei prodotti

```
def confirm_modify_order(self):
    alert_dialog = AlertDialog(self.style)
    alert_dialog.costumize_modify_order()
    alert_dialog.exec_()
    if alert_dialog.confirm:
        self.controller.modify_order_details()
        new_date = str(self.date_edit.date().toString("yyyy.MM.dd").replace('.', '-'))
        self.values_modified = [self.controller.nbill, new_date, self.controller.count]
        if not(self.type_user_flag):
            self.values_modified.append(self.controller.list_removed)
            self.values_modified.append(self.controller.list_modified)
            self.values_modified.append(self.controller.list_add)
        self.confirm = "edit"
        self.close()
```

Figura 4.15: Metodo associato alla conferma delle operazioni di modifica

```
def modify_order_details(self):
    self.list_removed = []
    self.list_modified = []
    self.list_add = []
    x = 0
    for e in self.list_support:
        if x < len(self.model_detail.list_order_details):
            if (e.get("removed")):
                self.model_detail.delete_order_detail(e.get("id"), self.nbill, self.type_order)
                if not(self.type_user_flag):
                    self.list_removed.append(e)
            elif (e.get("modified")):
                self.model_detail.modify_product_availability(e.get("id"), self.nbill, e.get("last_value_ava_modify"), self.type_order)
                if not (self.type_user_flag):
                    self.list_modified.append(e)
        elif x > len(self.model_detail.list_order_details):
            if not (e.get("removed")):
                self.model_detail.add_order_detail(ElementOrder(self.nbill, self.type_order, e.get("id"),
                                                                e.get("price"), e.get("last_value_ava_modify"), True))
                if not (self.type_user_flag):
                    self.list_add.append(ElementOrder(self.nbill, self.type_order, e.get("id"), e.get("price"),
                                                      e.get("last_value_ava_modify"), True))
        x += 1
```

Figura 4.16: Modifica dei prodotti associati all'ordine

```
def modify_order(self, string_confirm, value_modified):
    position = 0
    if string_confirm == "edit":
        for e in value_modified[3]:
            self.reflect_method(int(e.get("id")), int(e.get("original_ava")), False, self.type_user_flag)
        for e in value_modified[4]:
            if int(e.get("original_ava")) > int(e.get("last_value_ava_modify")):
                flag = True
                ava = int(e.get("original_ava")) - int(e.get("last_value_ava_modify"))
            else:
                flag = False
                ava = int(e.get("last_value_ava_modify")) - int(e.get("original_ava"))
            self.reflect_method([int(e.get("id")), ava, False, flag])

        for e in value_modified[5]:
            self.reflect_method([int(e.id_prod), int(e.availability), False, self.type_user_flag])

        self.model_order.modify_order_date_payment(value_modified[0], value_modified[1])
        self.model_order.modify_order_amount_due(value_modified[0], value_modified[2])
        order = self.model_order.get_order_by_id(value_modified[0])

        if self.filter_mode or self.sub_filter_mode:
            x = 0
            for o in self.list_resources:
                if o.get_nbill() == int(value_modified[0]):
                    self.list_resources[x] = order
                    position = x
                    break
                x += 1
        else:
            position = self.model_order.modify_element_list(int(value_modified[0]))

        return position
```

Figura 4.17: Modifica dell'ordine e dei prodotti nel magazzino

2. L'elemento, originariamente ordinato, è stato modificato; viene aggiornata la quantità associata, mediante il metodo *ModifyProductAvailability()*.

3. L'elemento, che non era associato all'ordine, è stato aggiunto; esso viene registrato nel sistema mediante il metodo *AddOrderDetails()*.

Inoltre, a seconda del risultato del controllo, il prodotto verrà aggiunto ad una lista tra *ListRemoved*, *ListSupport* o *ListAdd*.

Una volta eseguite tutte le operazioni di aggiornamento, i dati relativi alle modifiche effettuate vengono restituite alla classe *OrderSectionView()*.
Il flusso di esecuzione del programma riprende dalla funzione nella Figura 4.17, introdotta precedentemente. In particolare, viene letto il valore della variabile 'confirm' di *ModifyOrderDialog()*, essa indica quale operazioni sono state eseguite nella fase di modifica, affinchè possano essere invocate le opportune funzioni di *OrderCostumersSectionController()*.

Infatti il metodo *ModifyOrder()*, nella Figura 4.17, leggendo il valore *'edit'*, richiama il metodo *ReflectMethod()* a cui passa gli id e le nuove quantità dei prodotti modificati.

```
def reflect_method(self, params):
    edit_prod_dialog = EditProdDialog("", "", "")
    edit_prod_dialog.controller.modify_product_availability(params[0], params[1], [params[2], params[3]])
    prod = edit_prod_dialog.controller.get_product(int(params[0]))
    self.get_filter_params()
    values_back = self.controller.reflect_method(prod, self.filter_params)
    if values_back[0]:
        self.fill_row(values_back[1], values_back[2], True)
```

```
def reflect_method(self, prod, filter_params):
    row = self.model_list_product.modify_element_list(prod)
    if self.filters_applied:
        if self.check_filters(prod, filter_params):
            x = 0
            for p in self.list_resources:
                if p.get_id() == prod.get_id():
                    row = x
                    self.list_resources[x] = prod
                    return[True, prod, row]
                x += 1
        return [False, prod, row]
    else:
        return[True, prod, row]
```

Figura 4.18: Modifica dell'ordine e dei prodotti nel magazzino

Quest'ultimo, mostrato nella Figura 4.18, aggiorna la quantità dei prodotti sia nel database, che nell'interfaccia dedicata.

In seguito, vengono richiamati i metodi di *OrderCostumersSectionModel()* per modificare, qualora sia necessario, l'importo e la data di ritiro dell'ordine.

Infine viene effettuato un controllo sulla posizione dell'ordine nella tabella, per un eventuale aggiornamento nella stessa.

Capitolo 5

Testing

La pianificazione e lo sviluppo dei test richiedono un'attenta analisi volta a stabilire i componenti da controllare e a classificare parametri per la verifica dei risultati. Questa fase viene eseguita in parallelo all'implementazione del software, fornendo sia un'agevolazione nel processo di codifica sia un feedback continuo sulla bontà delle caratteristiche e delle funzionalità progettate per il software.

5.1 Obiettivi dei test

I risultati ottenuti ci hanno permesso di identificare quali funzionalità deve garantire il sofwtare e quali sono i risultati che il sistema deve generare in risposta agli input del titolare.

Non solo, dopo aver analizzato le caratteristiche e i requisti dell'applicazione, è stato possibile, innanzitutto, individuare quali fossero i componenti del sistema, classificandoli in base alle operazioni fornite e alla natura dei dati gestiti; successivamente, è stata stabilita la strategia per l'implementazione e l'integrazione dei singoli moduli.

I test svolti sui vari componenti del software dovranno verificare la validità delle seguenti proprietà:

- *Logica della gestione degli input:* le azioni del titolare dovranno essere correttamente gestite dai moduli dedicati; l'elaborazione dell'input dovrà seguire il workflow prestabilito, registrando i risultati ottenuti nelle apposite varibili, e, di conseguenza, apportare le modifiche desiderate allo stato del sistema.

- *Gestione dei dati:* le operazioni, fornite dai moduli di Python, per l'interrogazione e la modifica della base di dati, dovranno produrre i risultati attesi.

- *Comunicazione tra i sottosistemi:* qualora un modulo apporti determinate modifiche al database, rendendo obsoleti o incorretti i dati ottenuti dagli altri componenti nelle precedenti operazioni di lettura, sarà necessario notificare alle classi preposte gli aggiornamenti effettuati e le operazioni da eseguire.

5.2 Progettazione

5.2.1 Pianificazione

Nell'analisi dei casi d'uso del programma, svolta in fase di progettazione, sono state individuate, per ciascuno, le sequenze di azioni che portano al raggiungimento dell'obiettivo e quelle, chiamate "alternative", che che potrebbero condurre il programma in errore.

L'organizzazione dei test si baserà, di fatto, sugli *use case* del software; i moduli, infatti, verranno testati in funzione del ruolo svolto nella realizzazione degli scenari previsti per il sistema.

Un'ulteriore classificazione delle proprietà verificate dai test verrà effettuata in base al tipo dei operazioni effettuate dai singoli componenti e, in particolare, dalle specifiche funzioni.

L'affidabilità dei risultati ottenuti si fonda sulle caratteristiche implementative e strutturali del software. Innanzitutto le interfacce utente sono state realizzate in modo da garantire un accesso alle funzionalità sviluppate, guidato (e limitato) dalle specifiche e dai requisiti del sistema. Il programma, inoltre, è stato sviluppato per essere utilizzato da un unico attore, impedendo l'utilizzo ad altri.

Queste due caratteristiche ci permettono di limitare i nostri test ai soli errori riscontrabili nelle sequenze di azioni associate ai casi d'uso.

5.2.2 Tecniche implementative

Per lo sviluppo dei test e il controllo dei risultati sono stati utilizzati i moduli della libreria Python *unittest*. Questa, mettendo a disposizione del programmatore un ampio set di oggetti e funzionalità, permette sia di controllare il funzionamento di singole unità, sia di eseguire più test in parallelo per verificare i risultati di operazioni potenzialmente conflittuali.

Gli oggetti messi a disposizione dalla liberia *unittest* sono elencati di seguito:

- *test fixture*: fornisce tutte le operazioni preliminari per eseguire uno o più test. Queste possono includere, ad esempio, la creazione di una base dati, di un file o una directory.

- *test case*: rappresenta la singola unità di test; controlla il comportamento e le risposte del sistema in relazione ad un particoare insieme di input. La classe *TextCase*, fornita da unittest, permette di creare nuovi casi di test.

- *test suite*: viene utilizzato per permettere di aggregare i test che dovrebbero essere eseguiti insieme; questa classe consente di creare un insieme di *test case* o di *test suite*.

- *test runner*: è il componente adibito alla gestione dell'esecuzione dei test e alla restituzione del risultato all'utente.

Sulla base delle considerazione fatte finora, in relazione alle caratteristiche del sistema, delle funzionalità implementate e delle librerie utilizzate, si è deciso di strutturare l'implementazione dei test come esposto di seguito:

- Verrà implementata una classe "*Test*" per ogni caso d'uso del programma.
- Nell'analisi di ciascuno degli scenari bisognerà individuare la sequenza di azioni associata e, di seguito, identificare le singole fasi in base al tipo di operazioni eseguite (ad esempio un'interazione con la base di dati, l'elaborazione di input ricevuti dall'interfaccia o la restituzione dei risultati di una chiamata).
- Le singole fasi del caso d'uso verranno verificate, in modo sequenziale, mediante l'utilizzo dei *test case*; i risultati ottenuti, qualora l'esito sia positivo, vengono utilizzati per determinare lo stato del sistema da cui inizierà il test successivo.

5.3 Test della modifica di un ordine associato ad un cliente

Si è deciso di mostrare, come esempio, i test effettuati sulla modifica dei dati relativi ad un ordine effettuato da un cliente.

La scelta di questa funzionalità, la cui implementazione è stata descritta nel precedente capitolo, è stata dettata, come già spiegato, dal fatto che è quella che coinvolge il maggior numero di operazioni e di dati.

L'esempio, dunque, è quello che risulta, tra tutti, il più completo e che, di conseguenza, fornisce la maggior chiarezza possibile per la trattazione dell'argomento in esame.

Per ogni test effettuato verranno mostrati il codice di rifermento, i risultati ottenuti e le eventuali modifiche registrate nel database.

Inoltre, è stata utilizzata, all'interno del codice, la funzione *print()*, per identificare, qualora i test avessero un esito positivo, le azioni controllate e i dati elaborati. Tuttavia, questo metodo non è stato riportato nelle figure in cui viene mostrata l'implementazione, Al contrario, i log prodotti sono visibili in quelle relative ai risultati dei test.

Test sulle operazioni di scrittura e lettura del database

Il primo *test case* in analisi è *TestResourcesCreation()*, mostrato nella Figura 5.1, in particolare, si dovrà verificare la correttezza delle operazioni di scrittura e lettura della base di dati.

Innanzitutto, vengono creati i riferimenti alle classi adibite alla gestione dei dati, mediante il metodo *DBConnection()*, di seguito, vengono invocati i metodi per interagire con l'istanza della base di dati.

Di seguito sono state riportate le verifiche effettuate:

```python
def test_resources_creation(self):
    self.db_references()
    #TEST CREAZIONE PRODOTTI
    id1 = self.edit_prod_model.add_product(Product("prod1", "Lavorato", 11.90, 0, 10))
    id2 = self.edit_prod_model.add_product(Product("prod2", "Grezzo", 10.50, 0, 10))
    id3 = self.edit_prod_model.add_product(Product("prod3", "Bevanda", 12.50, 0, 10))
    self.edit_prod_model.add_product(Product("prod4", "Bevanda", 9.90, 0, 10))
    self.edit_prod_model.add_product(Product("prod5", "Lavorato", 16.00, 0, 10))
    self.edit_prod_model.add_product(Product("prod6", "Grezzo", 5.50, 0, 10))
    self.assertNotEqual(id3, 0, "Errore: non è stato possibile creare prod3")
    #TEST RECUPERO PRODOTTI DAL DB
    prod2 = self.edit_prod_model.get_product_by_id(id2)
    self.assertIsInstance(prod2, Product, "Errore: la query non ha restituito alcun risultato")
    self.assertEqual(id2, prod2.get_id(), "Errore: la query non ha restituito il risultato desiderato")
    # TEST CREAZIONE UTENTI E RECUPERO DAL DB
    creating_user = User("XXXXXX00X00X000X", "Nome", "Cognome", "mail@mail.com", "0000000000")
    self.costumers_section_model.add_user(creating_user)
    user_created = self.costumers_section_model.get_user_by_id("XXXXXX00X00X000X")
    self.assertIsInstance(user_created, User, "Errore: la query non ha restituito alcun risultato")
    self.assertEqual(user_created.cod_user, creating_user.cod_user, "Errore: la query non ha restituito il risultato desiderato")
    #TEST CREAZIONE ORDINI
    nbill = self.order_costumer_section_model.add_order(Order("2021/11/21", "2021/11/29", "XXXXXX00X00X000X", False, 45.4))
    self.assertNotEqual(nbill, 0, "Errore: non è stato possibile creare l'ordine")
    #TEST RECUPERO ORDINI DAL DB
    created_order = self.order_costumer_section_model.get_order_by_id(nbill)
    self.assertIsInstance(created_order, Order, "Errore: la query non ha restituito alcun risultato")
    self.assertEqual(nbill, created_order.get_nbill(), "Errore: la query non ha restituito il risultato desiderato")
    element1 = ElementOrder(nbill, "costumer", id1, 11.90, 1, True)
    self.order_detail_model.add_order_detail(element1)
    self.order_detail_model.add_order_detail(ElementOrder(nbill, "costumer", id2, 10.50, 2, True))
    self.order_detail_model.add_order_detail(ElementOrder(nbill, "costumer", id3, 12.50, 1, True))
    created_element = self.order_detail_model.get_order_detail(nbill, id1, "costumer")
    self.assertIsInstance(created_element, ElementOrder, "Errore: la query non ha restituito alcun risultato")
    self.assertEqual([element1.id_prod, element1.type, element1.n_bill], [created_element.id_prod, created_element.type, created_element.n_bill],
                     "Errore: la query non ha restituito il risultato desiderato")
```

Figura 5.1: Test relativi alla creazione di nuovi elementi nel database e al successivo recupero

1. *Test sulla creazione di un nuovo prodotto:* l'aggiunta viene realizzata dal metodo *AddProduct()*, questo restituisce l'id del nuovo elemento, generato automaticamente dal sistema. Mediante la funzione *AssertNotEqual()*, si verifica che l'id del prodotto inserito non sia uguale a zero e che, di conseguenza, l'operazione abbia avuto un esito positivo.

2. *Test sul recupero di un prodotto dal database:* la lettura dei dati relativi ad un prodotto viene realizzata dal metodo *GetProductById()*, questo restituisce l'elemento associato all'id passato come parametro. Il primo controllo, effettuato mediante il metodo *AssertIsIstance()*, controlla che la query restituisca una variabile di tipo *Product*. di seguito, qualora il primo test abbia avuto un esito positivo, richiamando il metodo *AssertEqual()*, vengono comparati l'identificativo passato come parametro, per la ricerca, e quello del prodotto ottenuto.

3. *Test sulle operazioni di scrittura e di lettura per gli utenti e gli ordini:* i controlli effettuati seguono un paradigma uguale a quelli utilizzati per la verifica delle funzioni relative ai prodotti.

I risultati, mostrati nella Figura 5.2, indicano il corretto esito del primo test. Inoltre, esse ci hanno permesso di aggiungere nel database dei dati fittizi, come riportato nella Figura 5.3, che verranno utilizzati nei test successivi.

```
Ran 1 test in 0.870s

OK

Process finished with exit code 0
RISULTATI TEST 1
1) Prodotto aggiunto correttamente nel database
2) Prodotto recuperato correttamente dal database
3) Utente creato e recuperato correttamente dal database
4) Ordine aggiunto correttamente nel database
5) Ordine recuperato correttamente dall database
6) Utente creato e recuperato correttamente dal database
```

Figura 5.2: Risultati del primo test

SELECT * FROM products

id	name	type	price	discount	availability
252	prod1	Lavorato	11.9	0	10
253	prod2	Grezzo	10.5	0	10
254	prod3	Bevanda	12.5	0	10
255	prod4	Bevanda	9.9	0	10
256	prod5	Lavorato	16	0	10
257	prod6	Grezzo	5.5	0	10

SELECT * FROM orders_detail

n_bill	type	id_prod	price	availability	still_available
87	costumer	252	11.9	1	1
87	costumer	253	10.5	2	1
87	costumer	254	12.5	1	1

SELECT * FROM costumer_orders

n_bill	date_emission	date_payment	cod_user	payment_settled	amount_due
87	2021-11-21	2021-11-29	XXXXXX00X00X000X	0	45.4

SELECT * FROM user

cod_user	name	surname	email	telephone
XXXXXX00X00X000X	Nome	Cognome	mail@mail.com	0000000000

Figura 5.3: Dati creati dopo il primo test

```
def test_list_in_modify_operation(self):
    self.db_references()
    self.init_modify_section(87)
    self.assertEqual(len(self.order_details_controller.list_original_products), 3, "non sono state rimossi i prodotti presenti nell'ordine")
    self.assertEqual(len(self.order_details_controller.list_support), 7, "problemi nella creazione di list_support")
    self.assertEqual(self.order_details_controller.list_support[3], "", "problemi nella divisione di list_support")
    i = 0
    #CONTROLLO PRODOTTI PRESENTI
    while i < 3:
        self.assertEqual(self.order_details_controller.list_support[i].get("id"),
                         self.order_details_controller.model_detail.list_order_details[i].id_prod, "errore")
        self.assertEqual(self.order_details_controller.list_support[i].get("modified"), False, "errore elem n "+str(i))
        self.assertEqual(self.order_details_controller.list_support[i].get("removed"), False, "errore elem n "+str(i))
        self.assertEqual(self.order_details_controller.list_support[i].get("last_value_ava_modify"),
                         self.order_details_controller.list_support[i].get("last_value_ava_modify"), "errore")
        i += 1
    i = 4
    #CONTROLLO PRODOTTI NON PRESENTI
    while i < 7:
        self.assertEqual(self.order_details_controller.list_support[i].get("id"), self.order_details_controller.list_original_products[i - 4][0], "errore")
        self.assertEqual(self.order_details_controller.list_support[i].get("modified"), False, "errore elem n "+str(i))
        self.assertEqual(self.order_details_controller.list_support[i].get("removed"), True, "errore elem n "+str(i))
        self.assertEqual(self.order_details_controller.list_support[i].get("last_value_ava_modify"), 1, "errore elem n "+str(i))
        self.assertEqual(self.order_details_controller.list_support[i].get("original_ava"), 0, "errore elem n "+str(i))
        i += 1
```

Figura 5.4: Test relativi alla struttura delle liste di supporto per la modifica dell'ordine

Test sulle operazioni di modellazione dei dati recuperati dal database

La funzione *TestListInModifyOperation()*, riportata nella Figura 5.4, innanzitutto, simula la scelta del titolare di visualizzare i dettagli relativi all'ordine 87, aggiunto durante il primo test, e valorizza *ListSupport* con le informazioni sui prodotti presenti nel sistema, in relazione alla modalità di modifica.

Bisogna ricordare, inoltre, che, di seguito alle precedenti operazioni di scrittura, all'interno del database sono presenti sei prodotti, tre dei quali sono stati associati all'ordine.

I test dovranno verificare le seguenti condizioni:

1. La lista dei prodotti non associati all'ordine deve contenere tre elementi.
2. *ListSupport* deve contenere sette elementi.
3. Gli elementi alle posizioni 0, 1 e 2 devono corrispondere ai prodotti associati all'ordine
4. Gli ultimi tre elementi della lista, al contrario, sono quelli non associati all'ordine. L'elemento 4, nella posizione centrale, non contiene alcun valore, ma funge da separatore.

I risultati, mostrati nella Figura 5.5, indicano il corretto esito del secondo test.

Test sulle operazioni di elaborazione degli input dell'utente

Una volta asserita la correttezza delle operazioni di inizializzazione della modalità di modifica, si dovrà verificare se la logica del sistema produce, a seguito di una interazione dell'utente con la tabella dei prodotti, i risultati attesi.

La funzione *TestEventsTable()*, mostrata nella Figura 5.6, simula cinque diversi casi di input e, per ciascuno, verifica le modifiche apportate allo stato del sistema.

```
Ran 1 test in 0.777s

OK
RISULTATI TEST 2
1) Sono stati rimossi i prodotti presenti nell'ordine da list_original_products
2) Creazione list_support avvenuta con successo
3) list_support valorizzata correttamente
```

Figura 5.5: Risultati del secondo test

di seguito vengono riportati le fasi del test:

1. Viene modificata la quantità di un prodotto presente nell'ordine, selezionando un valore uguale a quello registrato all'interno del database. Il sistema verifica sia che il prodotto non venga indicato come modificato, sia che la variabile contenente il prezzo dell'ordine non venga aggiornata.

2. Viene modificata la quantità di un prodotto presente nell'ordine, selezionando un valore diverso da quello attualmente registrato. Il sistema verifica sia che il prodotto venga indicato come modificato, sia che la variabile contenente il prezzo dell'ordine venga aggiornata.

3. Viene modificata la quantità di un prodotto presente nell'ordine, selezionando un valore uguale a quello attualmente registrato. Il sistema verifica che la variabile contenente il prezzo dell'ordine non venga aggiornata.

4. Viene rimosso un prodotto presente nell'ordine, selezionando un valore uguale a quello attualmente registrato. Il sistema verifica sia che il prodotto venga indicato come rimosso, sia che la variabile contenente il prezzo dell'ordine venga aggiornata.

5. Viene aggiunto un prodotto non presente nell'ordine, selezionando un valore uguale a quello attualmente registrato. Il sistema verifica sia che il prodotto venga indicato come non rimosso, sia che la variabile contenente il prezzo dell'ordine venga aggiornata.

6. Viene aggiunto un prodotto non presente nell'ordine, selezionando un valore uguale a quello attualmente registrato. Il sistema verificaì che la variabile contenente il prezzo dell'ordine venga aggiornata.

I risultati, mostrati nella Figura 5.7, indicano il corretto esito del terzo test.

Test sulle operazioni di aggiornamento del database

L'ultimo test effettuato è, sicuramente, quello più importante. Le funzioni controllate sono quelle adibite alla modifica dei dati riguardanti gli ordini, i relativi

```python
def test_events_table(self):
    self.db_references()
    self.init_modify_section(87)
    # CONTROLLO MODIFICA PRODOTTO PRESENTE NELL'ORDINE SENZA MODIFICARE QUANTITA INIZIALE
    self.order_details_controller.event_modify_view_table(2, 3, str(self.order_details_controller.list_support[2].get("original_ava")))
    self.assertEqual(self.order_details_controller.list_support[2].get("modified"), False, "errore, il prodotto è stato segnato come modificata")
    self.assertEqual(self.old_order.amount_due, self.order_details_controller.count, "prezzo ordine non modificato")
    # CONTROLLO MODIFICA PRODOTTO PRESENTE NELL'ORDINE
    self.order_details_controller.event_modify_view_table(2, 3, 3)
    self.assertEqual(self.order_details_controller.list_support[2].get("last_value_ava_modify"), 3, "quantità prodotto non modificata")
    self.assertEqual(self.order_details_controller.list_support[2].get("modified"), True, "errore, il prodotto non è stato segnato come modificato")
    new_price = self.old_order.amount_due + 2 * self.order_details_controller.list_support[2].get("price")
    self.assertEqual(self.order_details_controller.count, new_price)
    # CONTROLLO MODIFICA PRODOTTO PRESENTE NELL'ORDINE MA LA QUANTITA E QUELLA APPENA MODIFICATA
    self.order_details_controller.event_modify_view_table(2, 3, self.order_details_controller.list_support[2].get("last_value_ava_modify"))
    self.assertEqual(self.order_details_controller.list_support[2].get("last_value_ava_modify"), 3, "quantità prodotto non modificata")
    self.assertEqual(self.order_details_controller.list_support[2].get("modified"), True, "errore, il prodotto non è stato segnato come modificato")
    self.assertEqual(self.order_details_controller.count, new_price)
    # CONTROLLO RIMOZIONE PRODOTTO PRESENTE NELL'ORDINE
    self.order_details_controller.event_modify_view_table(1, 4)
    new_price = new_price - 2 * self.order_details_controller.list_support[1].get("price")
    self.assertEqual(self.order_details_controller.list_support[1].get("removed"), True, "prodotto non è stato rimosso")
    self.assertEqual(self.order_details_controller.count, new_price)
    # CONTROLLO AGGIUNTA NUOVO PRODOTTO ALL'ORDINE
    self.order_details_controller.event_modify_view_table(4, 4, 2)
    new_price = new_price + 2 * self.order_details_controller.list_support[4].get("price")
    self.assertEqual(self.order_details_controller.list_support[4].get("removed"), False, "prodotto non aggiunto")
    self.assertEqual(self.order_details_controller.list_support[4].get("last_value_ava_modify"), 2, "quantità prodotto non modificata")
    self.assertEqual(self.order_details_controller.count, new_price)
    # CONTROLLO MODIFICA NUOVO PRODOTTO AGGIUNTO ALL'ORDINE
    self.order_details_controller.event_modify_view_table(4, 3, 3)
    new_price = new_price + self.order_details_controller.list_support[4].get("price")
    self.assertEqual(self.order_details_controller.list_support[4].get("last_value_ava_modify"), 3, "quantità prodotto non modificata")
    self.assertEqual(self.order_details_controller.count, new_price)
```

Figura 5.6: Test relativi all'elaborazione degli input del titolare

prodotti e quelli presenti in negozio. Un qualsiasi errore nella logica di funzionamento potrebbe, di fatto, danneggiare tre diverse aree del database.

La funzione *TestModifyOrderAndDetail*, mostrata nella Figura 5.8, dopo aver ripetuto la simulazione degli eventi, invoca il metodo *ModifyOrderDetails()* per aggiornare i dati dei prodotti associati all'ordine.

di seguito sono descritti i controlli relativi alla prima delle operazioni di modifica:

- Le liste *ListAdd*, *ListModified* e *ListRemoved* devono contentenere un solo elemento ciascuno.
- L'elemento contenuto in *ListRemoved*, che è stato rimosso dall'ordine, non deve essere presente nel database.
- La disponibilità dell'elemento contenuto in *ListModified*, selezionata in fase di modifica, deve essere uguale a quella che è stata prelevata dal database.
- L'elemento contenuto in *ListAdd*, ovvero quello che è stato aggiunto all'ordine, deve essere presente nel database.

Una volta verificato l'aggiornamento dei prodotti associati all'ordine, viene invocato il metodo *ModifyOrder()* a cui vengono passati i parametri contenenti le informazioni sulle modifiche effettuate.

I test effettuati, di seguito a questa seconda operazione, sono i seguenti:

```
Ran 1 test in 0.893s

OK

Process finished with exit code 0
RISULTATI TEST 3

1.
 Controllo: Modifica del prodotto numero 254 originariamente presente nell'ordine, impostando come nuova quantità quella originale
Quantità originale: 1
Modifico impostando come nuova quantità il valore 1
Risultato 1: Campo prodotto relativo allo stato di modifica rimane settato a False
Risulato 2: Prezzo ordine non modificato

2.
 Controllo: Modifica del prodotto numero 254 originariamente presente nell'ordine, impostando una quantità differente da quella attualmente registrata
Quantità attuale: 1
Modifico impostando come nuova quantità il valore 3
Risultato 1: Campo prodotto relativo allo stato di modifica settato a True
Risultato 2: Campo prodotto relativo all'ultimo valore impostato per la quantità aggiornato
Risultato 3: Prezzo ordine aggiornato

3.
 Controllo: Modifica del prodotto numero 254 originariamente presente nell'ordine, impostando una quantità uguale a quella attualmente registrata
Quantità attuale: 3
Modifico impostando come nuova quantità il valore 3
Risultato 1: Campo prodotto relativo allo stato di modifica rimane settato a True
Risultato 2: Campo prodotto relativo all'ultimo valore impostato per la quantità non aggiornato
Risultato 3: Prezzo ordine non aggiornato

4.
 Controllo la rimozione del prodotto numero 253 originariamente presente nell'ordine
Risultato 1: Campo prodotto relativo allo stato di rimozione settato a True
Risultato 2: Prezzo ordine aggiornato

5.
 Controllo l'aggiunta del prodotto numero 255 originariamente non presente nell'ordine
Aggiungo impostando come quantità il valore 3
Risultato 1: Campo prodotto relativo allo stato di rimozione settato a False
Risultato 2: Campo prodotto relativo all'ultimo valore impostato per la quantità aggiornato
Risultato 3: Prezzo ordine aggiornato

6.
 Controllo la modifica del prodotto numero 255appena aggiunto all'ordine
Quantità attuale: 2
Modifico impostando come nuova quantità il valore 3
Risultato 1: Campo prodotto relativo all'ultimo valore impostato per la quantità aggiornato
Risultato 2: Prezzo ordine aggiornato
```

Figura 5.7: Risultati del terzo test

```
def test_modify_order_and_details(self):
    self.db_references()
    nbill = 87
    self.init_modify_section(87)
    self.events_table()
    self.order_details_controller.modify_order_details()
    self.assertEqual(len(self.order_details_controller.list_removed), 1, "Errore nella creazione di list_removed")
    self.assertEqual(len(self.order_details_controller.list_modified), 1, "Errore nella creazione di list_modified")
    self.assertEqual(len(self.order_details_controller.list_add), 1, "Errore nella creazione di list_add")
    self.assertEqual(self.order_details_controller.list_removed[0].get("id"), self.order_details_controller.list_support[1].get("id"),
                     "non è stato inserito l'elemento corretto")
    removed_detail = self.order_detail_model.get_order_detail(nbill, self.order_details_controller.list_removed[0].get("id"), "costumer")
    self.assertIsNone(removed_detail, "prodotto non è stato eliminato")
    modified_detail = self.order_detail_model.get_order_detail(nbill, self.order_details_controller.list_modified[0].get("id"), "costumer")
    self.assertEqual(self.order_details_controller.list_modified[0].get("id"), self.order_details_controller.list_support[2].get("id"),
                     "non è stato inserito l'elemento corretto")
    self.assertEqual(self.order_details_controller.list_modified[0].get("last_value_ava_modify"), modified_detail.availability)
    added_detail = self.order_detail_model.get_order_detail(nbill, self.order_details_controller.list_add[0].id_prod, "costumer")
    self.assertEqual(self.order_details_controller.list_add[0].id_prod, self.order_details_controller.list_support[4].get("id"),
                     "non è stato inserito l'elemento corretto")
    self.assertEqual(self.order_details_controller.list_add[0].availability, added_detail.availability)
    self.order_costumer_section_controller.modify_order("edit", [self.order_details_controller.nbill, "2021/11/30",
                                                                 self.order_details_controller.count, self.order_details_controller.list_removed,
                                                                 self.order_details_controller.list_modified, self.order_details_controller.list_add])
    self.order_costumer_section_model = OrdersCostumerSectionModel()
    new_order = self.order_costumer_section_model.get_order_by_id(nbill)
    self.assertEqual(new_order.amount_due, self.order_details_controller.count, "prezzi non uguali poichè amount_due = " +
                     str(new_order.amount_due) + " count = "+str(self.order_details_controller.count))
    self.assertEqual(new_order.date_payment, QtCore.QDate.fromString("2021-11-30", 'yyyy-MM-dd'))

    self.edit_prod_model = EditProdModel()
    prod_added = self.order_details_controller.list_add[0]
    prod_added_get = self.edit_prod_model.get_product_by_id(self.order_details_controller.list_add[0].id_prod)
    original_ava = self.order_details_controller.list_support[4].get("ava_prod")
    self.assertEqual(original_ava - prod_added.availability, prod_added_get.availability,
                     "Non è stata aggiornata la quantità del prodotto aggiunto all'ordine")

    diff = self.order_details_controller.list_modified[0].get("last_value_ava_modify") - \
           self.order_details_controller.list_modified[0].get("original_ava")
    ava_prod_modified = self.order_details_controller.list_modified[0].get("ava_prod") - diff
    prod_modified = self.edit_prod_model.get_product_by_id(self.order_details_controller.list_modified[0].get("id"))
    self.assertEqual(ava_prod_modified, prod_modified.availability,
                     "Non è stata aggiornata la quantità del prodotto modificato")

    ava_prod_removed = self.order_details_controller.list_removed[0].get("ava_prod") + \
                       self.order_details_controller.list_removed[0].get("original_ava")
    prod_removed = self.edit_prod_model.get_product_by_id(self.order_details_controller.list_removed[0].get("id"))
    self.assertEqual(ava_prod_removed, prod_removed.availability,
                     "Non è stata aggiornata la quantità del prodotto rimosso")
```

Figura 5.8: Test relativi all'aggiornamento del sistema

```
Ran 1 test in 1.001s

OK

Process finished with exit code 0

1.
Controllo che il prodotto numero 253 sia stato aggiunto alla lista dei prodotti rimossi ed eliminato dal database
Risultato 1: Prodotto inserito in list_removed
Risultato 2: Prodotto eliminato correttamente dal database

2.
Controllo che il prodotto numero 254 sia stato aggiunto alla lista dei prodotti modificati e che la sua quantità sia stata aggiornta dal database
Risultato 1: Prodotto inserito in list_modified
Risultato 2: Quantità prodotto aggiornata correttamente nel database

3.
Controllo che il prodotto numero 255 sia stato aggiunto alla lista dei prodotti aggiunti e che la sua quantità sia stata aggiornta dal database
Risultato 1: Prodotto inserito in list_add
Risultato 2: Prodotto inserito correttamente nel database

4.
Controllo che l'ordine e i relativi dettagli sia stati modificati correttamente
Risultato 1: quantità del prodotto n. 255 aggiornata
Risultato 2: quantità del prodotto n. 254 aggiornata
Risultato 3: quantità del prodotto n. 253 aggiornata
Risultato 4: dati ordine relativi a prezzo e data aggiornati.
```

Figura 5.9: Risultati del quarto test

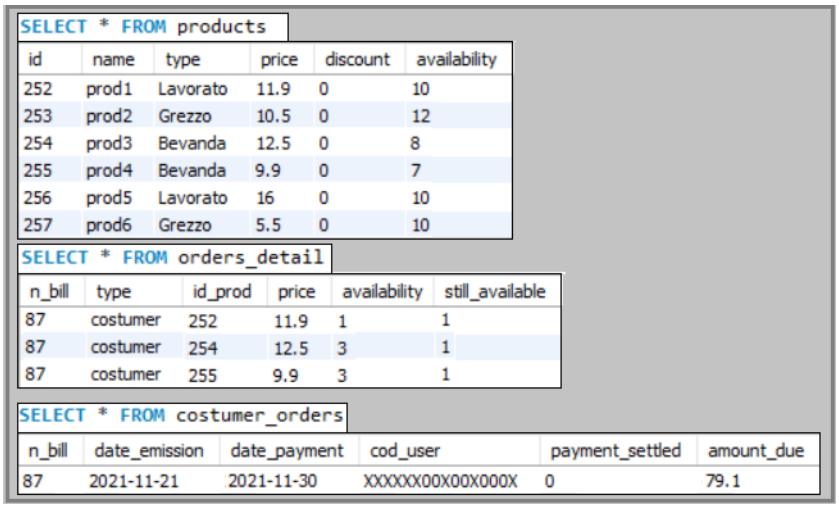
1. Si deve verificare che i dati riguardanti il prezzo e la data di ritiro dell'ordine siano stati aggiornati.

2. La disponibilità in negozio del prodotto aggiunto all'ordine deve risultare decrementata di un valore pari alla quantità selezionata durante la fase di modifica.

3. La disponibilità in negozio del prodotto rimosso dall'ordine deve risultare incrementata di un valore pari alla quantità ordinata prima della fase di modifica.

4. Per quanto riguarda l'elemento la cui quantità è stata modificata, la differenza selezionata dovrà essere pari all'incremento o al decremento registrato per il corrispettivo prodotto.

I risultati, mostrati nella Figura 5.9, indicano il corretto esito dell'ultimo test. Inoltre, nella Figura 5.10, sono riportare le modifiche effettuate dalle operazioni testate sul database



```
SELECT * FROM products
```

id	name	type	price	discount	availability
252	prod1	Lavorato	11.9	0	10
253	prod2	Grezzo	10.5	0	12
254	prod3	Bevanda	12.5	0	8
255	prod4	Bevanda	9.9	0	7
256	prod5	Lavorato	16	0	10
257	prod6	Grezzo	5.5	0	10

```
SELECT * FROM orders_detail
```

n_bill	type	id_prod	price	availability	still_available
87	costumer	252	11.9	1	1
87	costumer	254	12.5	3	1
87	costumer	255	9.9	3	1

```
SELECT * FROM costumer_orders
```

n_bill	date_emission	date_payment	cod_user	payment_settled	amount_due
87	2021-11-21	2021-11-30	XXXXXX00X00X000X	0	79.1

Figura 5.10: Dati aggiornati dopo l'ultimo test

Capitolo 6

Conclusioni

Nel corso della trattazione è stato studiato l'utilizzo delle le best practice dell'ingegneria del software in relazione allo sviluppo di un sistema per la gestione di un punto vendita di prodotti alimentari. Analizzare singolarmente le diverse attività ci ha permesso di fornire una descrizione del nostro software sotto più punti di vista, tra loro diversi, ma fortemente interconnessi.

Il primo passo è stato definire sia le caratteristiche e le funzionalità del software sia i vincoli da rispettare. Attraverso un'indagine preliminare sul contesto di riferimento e sulle esigenze del cliente, è stato possibile indentificare i requisiti e le specifiche del sistema.

Una volta ottenute tutte le premesse necessarie, abbiamo definito le questioni progettuali. Sono stati individuati, innanzitutto, quali sono i sottosistemi in cui è stato possibile suddividere il nostro software, in base alle funzionalità svolte e al tipo di dati gestiti. In seguito, si è deciso di basare l'organizzazione del sistema sul modello proposto dal pattern architetturale *Model-View-Controller*. Esso ci ha fornito gli strumenti necessari per definire la struttra dei componenti e i metodi di comunicazione tra gli stessi.

Dopo aver definito le caratteristiche strutturali, ci siamo occupati degli aspetti funzionali del programma; attraverso l'analisi effettuata nella terza fase abbiamo mostrato, per ogni scenario del programma, il flusso di operazioni associato e gli output prodotti.

Nell'ultima parte della trattazione sono state descritte le attività di codifica e di collaudo del software. Innanzitutto, abbiamo analizzato le funzionalità e le caratteristiche di *Python* in relazione ai moduli utilizzati. In seguito sono stati presentati l'ambiente di sviluppo *Pycharm* e il tool grafico *QtDesigner*, utilizzati per lo sviluppo e la realizzazione del back-end e del front-end del programma.

Lo studio effettuato ci ha permesso di ottenere un software caratterizzato dalle proprietà di qualità, di seguito riportate:

- *Accettabilità*: le funzionalità offerte dal programma soddisfano tutti i requisti e i vincoli imposti in fase di progettazione.

- *Mantenibilità*: lo studio delle best practice dell'ingegneria del software e il successivo utilizzo durante la fase di progettazione, ci hanno permesso di realizzare

un sistema software ben strutturato. L'organizzazione delle classi, i moduli di comunicazioni tra le stesse e la logica delle operazioni effettuate seguono dei modelli generali e delle regole definite rigorosamente. Di conseguenza, il codice sviluppato risulta facilmente comprensibile, rendendo possibile, di conseguenza, effettuare le operazioni di aggiornamento, integrazione e manutenzione.

- *Fidatezza*: i test sulle funzionalità sviluppate ci hanno fornito i risultati attesi. I controlli effettuati hanno riguardato principalmente la coerenza delle operazioni di lettura e scrittura del database, la modellazione e l'aggiornamento dei dati nello stato del programma e la correttezza della logica di elaborazione degli input del titolare.

Infine, alla luce dei risultati ottenuti, riteniamo che le strategie progettuali e le soluzioni implementative possano essere riutilizzate per progetti futuri, qualora esse presentino le specifiche e i requisiti adatti.

Bibliografia

[1] Sommerville. *Ingegneria del Software, Decima Edizione. Pearson Education*, 2017.

[2] Shvets. *Dive into Design Patterns. Ebook.*

[3] M.Boscaini. *Imparare a programmare in Python. Apogeo.*, 2020.

[4] Python 3.10.0 documentation. *https://docs.python.org/3/.*

[5] Get started pycharm - jetbrains. *https://www.jetbrains.com/help/pycharm/quick-start-guide.html.*

[6] Mysql workbench manual. *https://dev.mysql.com/doc/workbench/en/.*

[7] Design patterns in python. *https://refactoring.guru/design-patterns/python.*

[8] Web frameworks for python. *https://wiki.python.org/moin/WebFrameworks.*

[9] Pyqt5 reference guide. *https://www.riverbankcomputing.com/static/Docs/PyQt5/.*

[10] Qt designer manual. *https://doc.qt.io/qt-5/qtdesigner-manual.html.*