Università degli Studi di Urbino Carlo Bo
Corso di Laurea Triennale in Informatica Applicata
Programmazione e Modellazione a Oggetti

App di Food Delivery
Relazione del Progetto

Procicchiani Luca
Matricola 290489
Tombari Giacomo
Matricola 306405

Anno Accademico 2021/2022

Indice

Capitolo 1

Analisi

Il gruppo si pone come obiettivo la realizzazione di un'applicazione gestionale di consegne a domicilio. L'applicazione dovrà consentire ai clienti di effettuare degli ordini e gestire l'assegnamento delle consegne ai fattorini, i quali avranno il compito di effettuare multiple consegne.

Funzionalità fornite dal progetto:

* Il cliente dovrà selezionare il luogo di consegna (in una delle città consentite), il ristorante da cui ordinare e i menu offerti da quest'ultimo.
* Terminato l'ordine sarà visualizzato un resoconto con l'importo totale.
* Il programma tiene conto per ogni fattorino delle città nelle quali egli consegna. Ci possono essere più fattorini che si occupano della stessa città, in questo caso durante l'assegnamento degli ordini il programma predilige chi ha guadagnato di meno.
* Verrà inoltre considerato per lo smistamento degli ordini lo spazio a disposizione nello zaino di ogni fattorino, il quale avrà una capienza massima prestabilita.
* Sarà infine gestita l'interfaccia dei fattorini, la quale mostrerà gli ordini a loro assegnati e quelli in attesa. I corrieri possono a loro piacimento partire e consegnare tutti i loro ordini, svuotando dunque il proprio zaino e guadagnando automaticamente un 10% del totale degli ordini consegnati.
* (Funzionalità opzionale) Composizione di menu personalizzabili dall'utente, oltre ai semplici menu standard proposti dai ristoranti.

1.1 Requisiti

Requisiti funzionali:

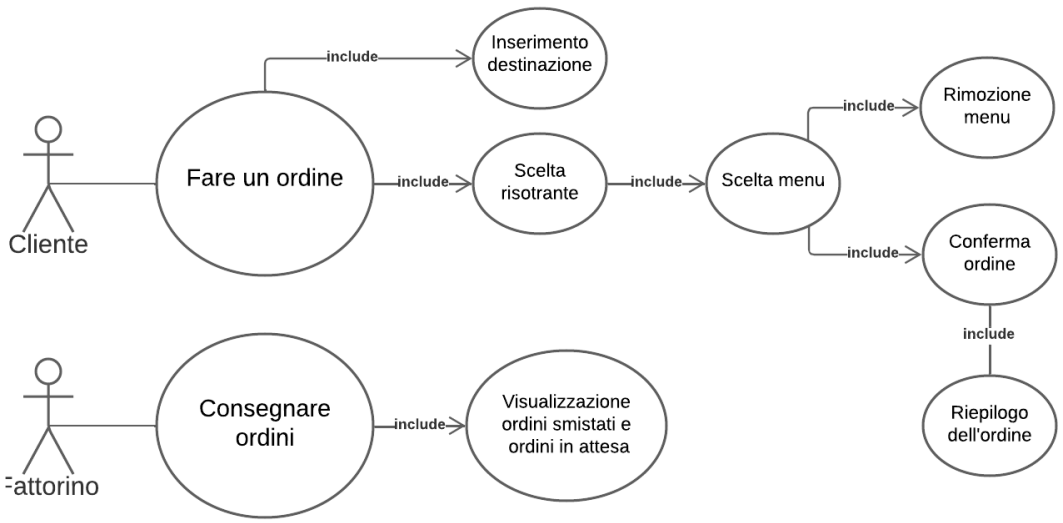
- L'applicazione permette l'inserimento della destinazione scegliendo tra le città preimpostate.
- L'applicazione permette la scelta del ristorante e mostra i relativi menu offerti.
- L'applicazione permette di scegliere i menu e la loro quantità.
- L'applicazione al termine di un ordine mostra un riepilogo dettagliato.
- Ai fattorini vengono mostrati gli ordini assegnati e quelli in attesa.
- Ai fattorini è permesso di segnalare una consegna, svuotando il loro zaino e aggiornando i loro profitti.
- Gli ordini che non possono essere assegnati vengono messi in attesa e smistati appena possibile, seguendo l'ordine di creazione.

Requisiti non funzionali:

- L'applicazione deve essere intuitiva e facile da usare.
- L'applicazione permette all'utente di comporre menu personalizzati.

Insieme all'analisi dei requisiti sono stati definiti i principali casi d'uso dell'applicazione.

Figura 1.1: Diagramma dei casi d'uso



Gli attori individuati sono il cliente e il fattorino, e i due casi d'uso generali sono rispettivamente la creazione di un ordine e la consegna degli ordini assegnati. Queste macro-funzioni ne implicano altre, abbiamo infatti dei casi d'uso "inclusi" in quelli principali.

Il diagramma dei casi d'uso ci ha aiutato a inquadrare sin da subito i requisiti e le interazioni degli utenti con il sistema.

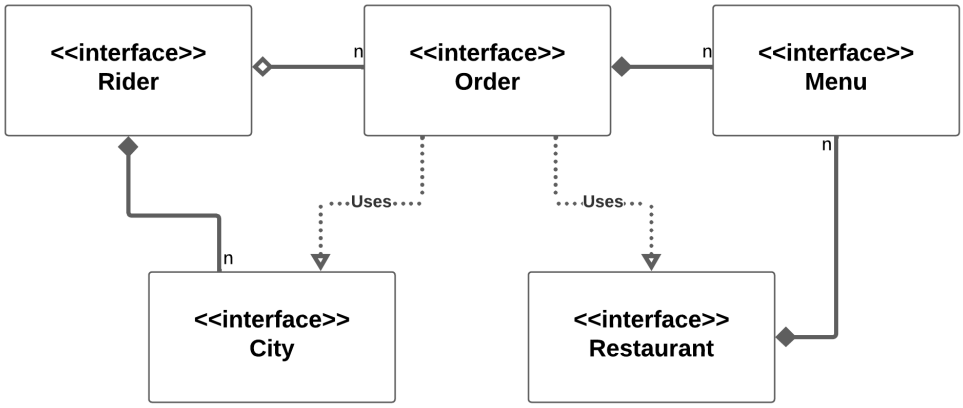
1.2 Modello del dominio

Analizzando la specifica, abbiamo individuato le principali entità del problema e le relazioni che intercorrono fra loro, creando così il modello del dominio.

Gli elementi principali del problema sono i seguenti:

- Menu, elemento alla base del nostro problema.
- Order, entità composta da più Menu.
- Rider, i quali hanno degli Order a loro assegnati.
- City, un numero limitato di città in cui si consegna.
- Restaurant, la cui offerta si compone di diversi Menu.

Figura 1.2: Schema del modello del dominio



Capitolo 2

Design

2.1 Architettura

Per lo sviluppo dell'architettura del nostro software si è deciso di adottare uno dei pattern architetturali più diffusi nell'ambito della programmazione ad oggetti, il Model-View-Controller (MVC). Questo permette di separare gli aspetti puramente algoritmici e logici, che fungono da scheletro del programma, da quelli iterativi. Nello specifico, come ci suggerisce il nome, vengono individuate tre componenti principali: il modello, la view e il controller.

Questo principio ci ha aiutato sia in fase di sviluppo per la suddivisione dei vari compiti, sia in caso di modifiche, poiché a seconda di cosa doveva essere cambiato sapevamo su quale delle tre parti dovevamo andare a lavorare.

2.1.1 Model

Il Model si occupa dell'accesso ai dati. Lo scopo ultimo del modello è quello di fornire al Controller una rappresentazione attuale dello stato interno dell'applicazione.

L'implementazione della parte di modello è avvenuta partendo dall'analisi del modello del dominio e creando interfacce per ogni entità individuata. L'ottima analisi svolta ci ha facilitato questo lavoro.

Figura 2.1: UML delle interfacce del componente Model

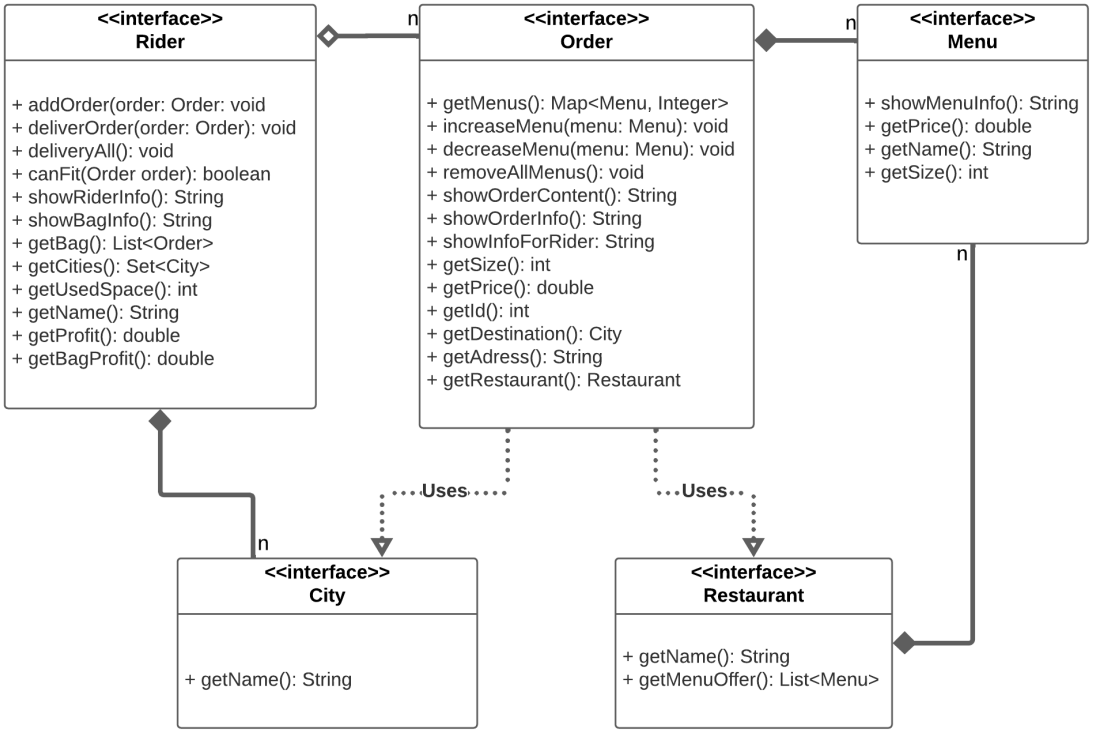
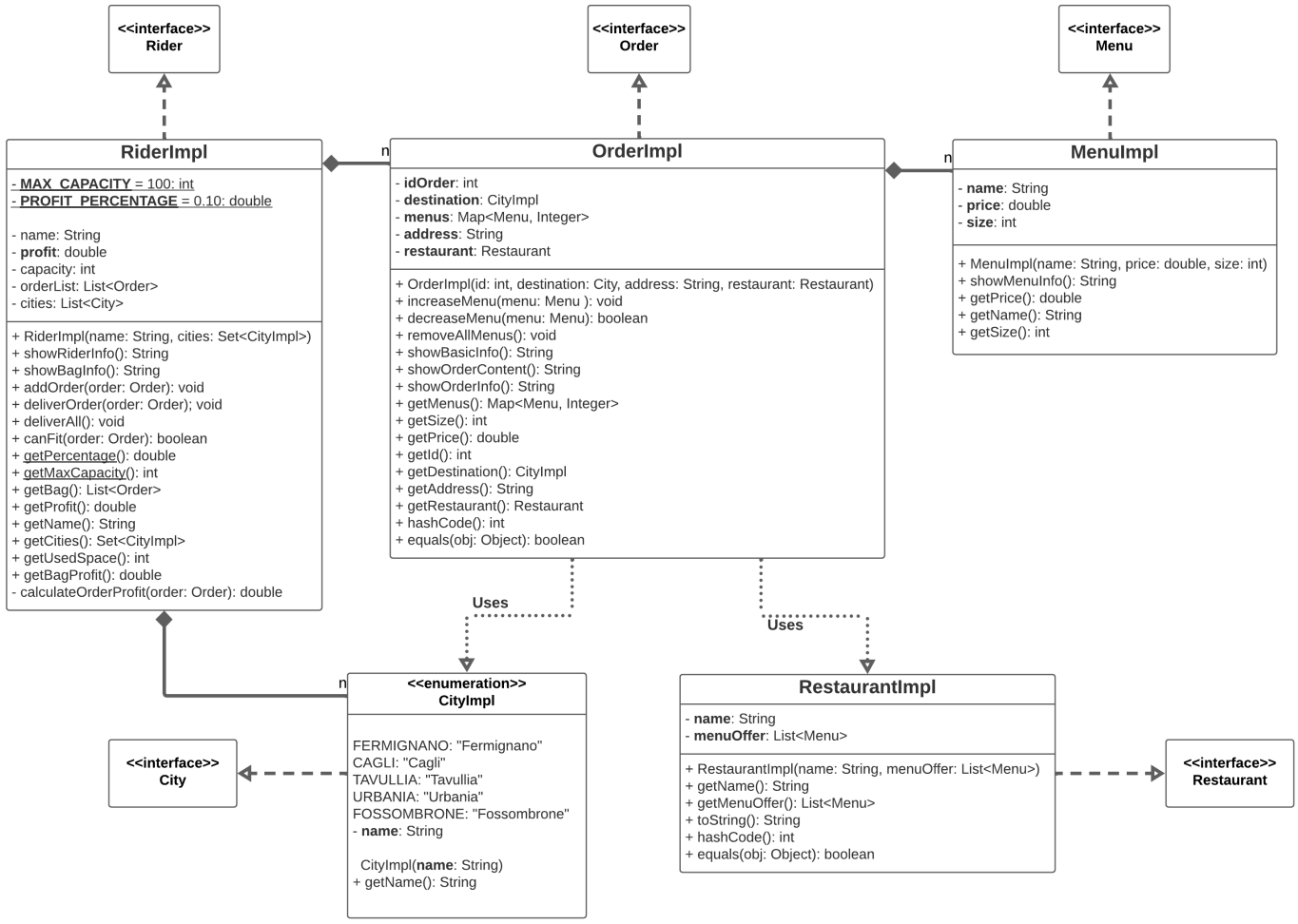


Figura 2.2: UML delle classi del componente Model



2.1.2 View

Il compito della View è quello di visualizzare i dati contenuti nel model e occuparsi dell'interazione con utenti. Fornisce dunque all'utente una User Experience soddisfacente e funzionale. Questa interazione utente-view può avvenire tramite un'interfaccia grafica, come nel nostro caso, o anche tramite linea di comando. In particolare noi per l'implementazione della GUI abbiamo fatto uso della libreria Java Swing.

Analizzando la specifica abbiamo scelto di suddividere le schermate della View in due parti: quella dedicata ai clienti, che permette loro di creare degli ordini, e quella dedicata ai fattorini, che permette loro di visualizzare gli ordini assegnati e di consegnarli.

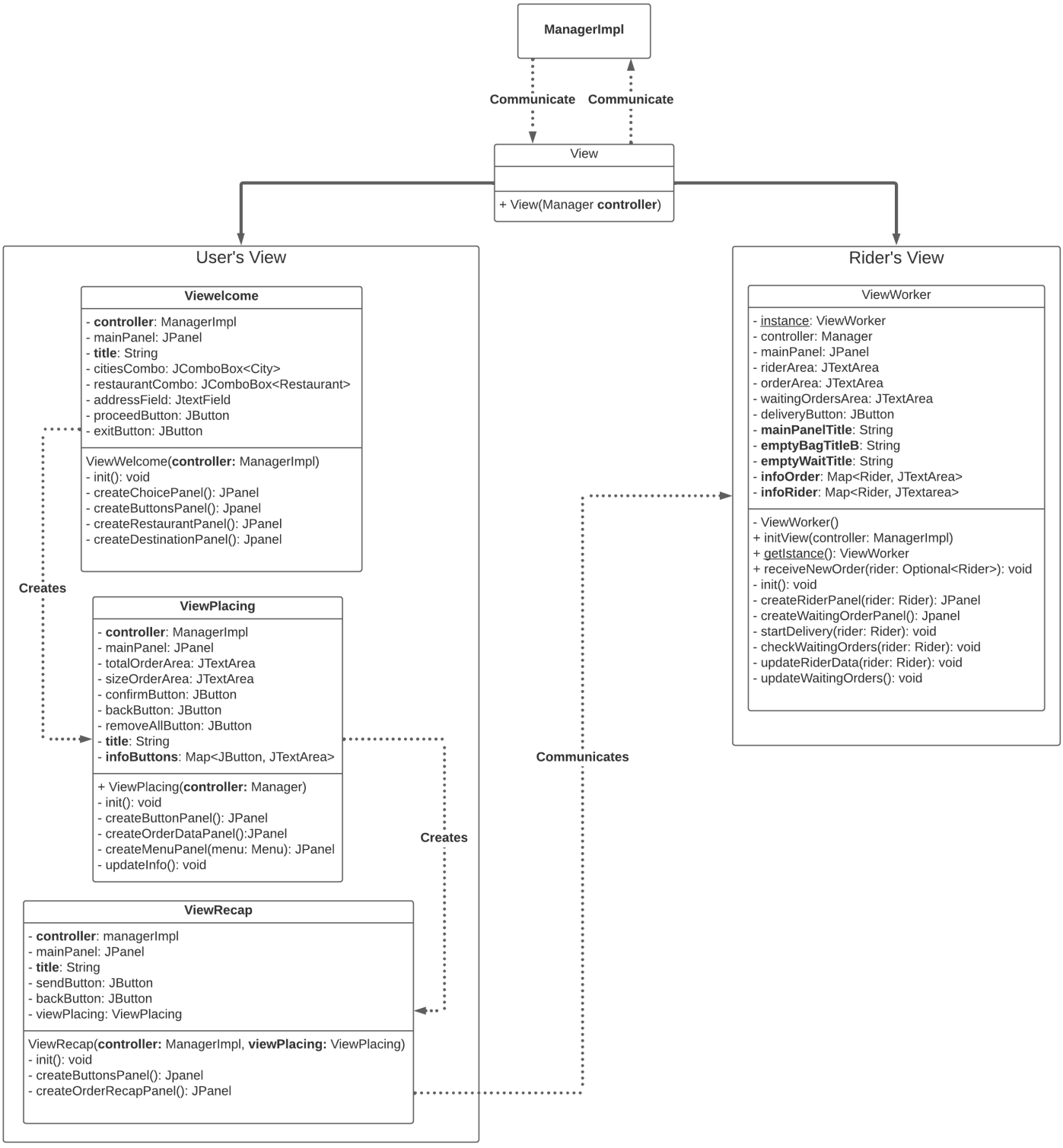
Nello specifico la parte di view dei clienti è stata divisa in 3 schermate sequenziali:

- `ViewWelcome`, fornisce all'utente la possibilità di inserire i primi dati necessari per la creazione del suo ordine, ovvero indirizzo di destinazione e ristorante.

- `ViewPlacing`, mostra all'utente tutti i menu offerti dal ristorante selezionato e permette di aggiungerli e rimuoverli dall'ordine. Inoltre, in caso l'utente volesse cambiare le scelte fatte nella schermata precedente, fornisce la possibilità di tornare indietro.
- `ViewRecap`, mostra all'utente un resoconto finale del suo ordine e gli permette di confermarlo. Anche questa schermata fornisce la possibilità di tornare indietro nel caso si voglia modificare qualcosa nell'ordine.

La parte dei fattorini, denominata `ViewWorker`, è invece data da una singola schermata che mostra il contenuto dello zaino di ogni rider con le principali informazioni relative ad ogni ordine e permette loro di segnalare quando svolgono una consegna. Inoltre mostra anche tutti gli ordini che sono in attesa.

Figura 2.3: UML del componente View

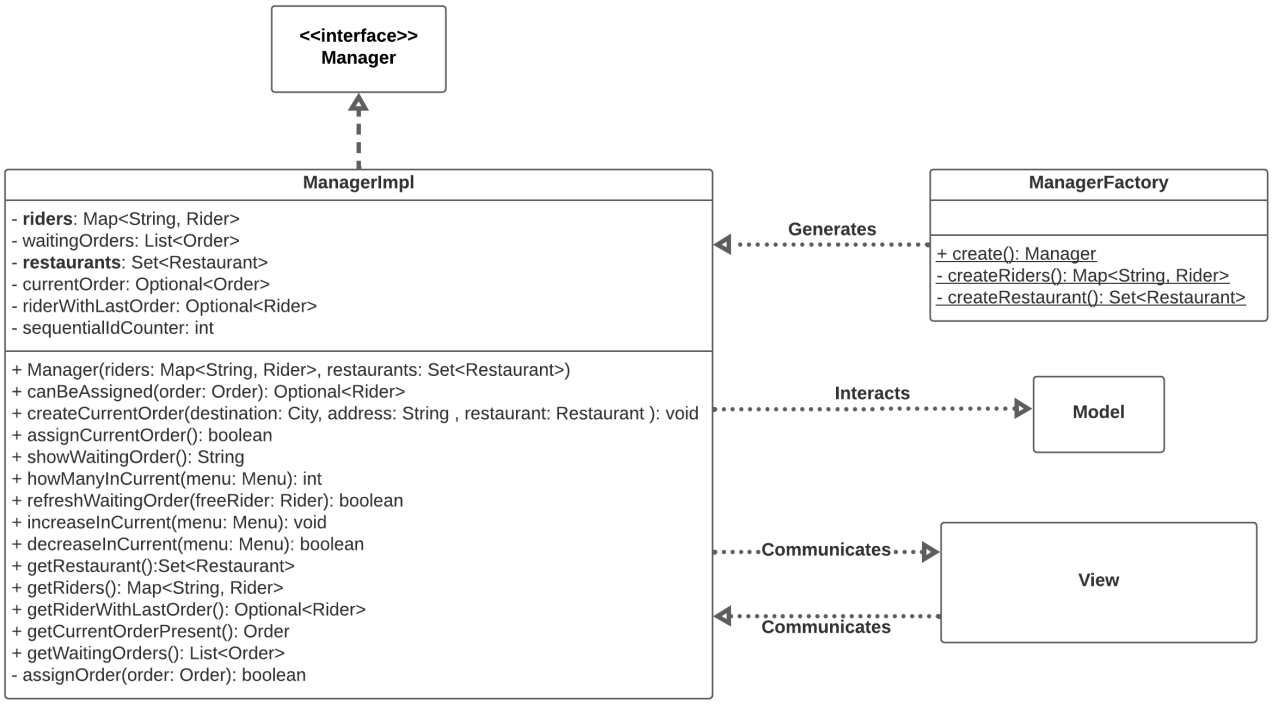


2.1.3 Controller

Il Controller ha il compito di ricevere le interazioni dell'utente dalla View e di cambiare di conseguenza lo stato del Model. Avvenuto ciò comunicherà i cambiamenti alla View in modo che questa possa aggiornarsi correttamente.

Dato che nel nostro caso la creazione del Controller, denominato `ManagerImpl`, necessita la creazione di varie strutture dati complesse, questo compito è stato delegato ad una classe apposita (come approvondito nella sezione 2.2.1).

Figura 2.4: UML del Controller e relativa factory



2.1.4 Punti d'accesso

Nel nostro programma i punti di accesso ai 3 componenti del pattern MVC sono i seguenti:

- Le interfacce del modello sono tutte quelle nel package `it.fooddelivery.model`, mentre le classi del modello sono tutte quelle in `it.fooddelivery.model.implementation`;
- Il controller è identificato dalla classe chiamata `ManagerImpl`;
- Il punto di accesso alla view è una classe chiamata `View` che fa uso delle altre classi nel package `it.fooddelivery.view`.

2.2 Design dettagliato

2.2.1 Procicchiani Luca

Controller e factory

Essendomi occupato del Controller per il pattern MVC, un primo problema riscontrato è stato quello di fornire anche un metodo di costruzione agevole, infatti il Controller va inizializzato configurandolo con i giusti ristoranti e fattorini.

L'idea di costruirlo all'interno del Main l'ho subito scartata per non andare a "sporcare" il Main stesso. L'idea di configurarlo direttamente all'interno della classe `ManagerImpl` mi è sembrata altrettanto sbagliata, per una questione di riuso ed estensione del codice non mi sembra la miglior cosa dover andare a toccare il codice del Controller ogni qualvolta il programma debba essere aggiornato con ad esempio un nuovo fattorino assunto.

Il Factory Pattern mi è sembrato fare al caso nostro, ci permette infatti di delegare a un'altra classe la configurazione del Controller, separando quindi le informazioni di configurazione dalla classe stessa.

Nel nostro caso essendo il Controller chiamato `Manager` è stata creata una `ManagerFactory`, e questi due sono rispettivamente nel pattern il nostro prodotto e il nostro produttore (UML alla figura 2.4).

Il Controller si occupa inoltre dello smistamento degli ordini. È stato infatti realizzato un metodo che controlla quale fattorino sia il più adeguato alla consegna, in base ai criteri descritti nella specifica del progetto: si opera una `stream` su tutti i fattorini, filtrando innanzitutto quelli che consegnano nella città di destinazione dell'ordine e poi rifiltrando solo coloro che hanno abbastanza spazio nello zaino, per poi ordinarli per profitto crescente e prendere il fattorino con meno guadagni. In caso si sia effettivamente trovato un candidato si assegna l'ordine al fattorino in questione, in caso non ci sia nessuno al momento disponibile l'ordine verrà messo in lista d'attesa.

Modello

Riguardo alla parte di modello io mi sono occupato dell'implementazione delle interfacce e relative classi dei ristanti e dei fattorini.

Per i rider bisognava tenere conto di qualche dato semplice come nome e profitto, più qualche dato complesso come l'elenco delle città in cui questi consegnano e la lista degli ordini a loro assegnati. Si è dunque fatto uso di `collections`: per le città si è optato in particolare per un `set`, non essendoci ovviamente ripetizioni in questo elenco ed essendo ininfluente l'ordine degli elementi; per gli ordini assegnati al fattorino si è invece fatto uso di una `list` dato che l'ordine di ricezione degli ordini può essere importante (ad esempio nella stampa del contenuto dello zaino, li si vuole per comodità in ordine di arrivo).

Inoltre per i fattorini sapevamo che tutti dovevano avere la stessa capienza massima dello zaino e anche lo stesso tasso di guadagno sulle consegne, si sono dunque implementati nella classe Rider due campi `static final` appositi.

ViewWelcome e ViewRecap

Le due schermate di cui mi sono occupato sono quella di benvenuto per il cliente e quella di riepilogo dell'ordine.

Nella `ViewWelcome` ho fatto uso di JComboBox per la selezione della città e del ristorante, essendo entrambe scelte di un solo elemento da una lista finita di possibili opzioni.

Per procedere con le fasi dell'ordine e andare alla successiva schermata ho creato un bottone con associata al click la generazione della finestra successiva.

Nella `ViewRecap`, oltre alla possibilità di procedere, vi è anche quella di tornare alla schermata precedente. Infatti il costruttore di questa classe prende in ingresso il riferimento alla classe della schermata precedente (la `ViewPlacing`) così, in caso di pressione del bottone "Indietro", si smantella questa finestra e si fa ricomparire la precedente.

2.2.2 Tombari Giacomo

Design Modello

I miei compiti in questa sezione comprendevano l'implementazione di tre delle componenti base del nostro sistema, l'enum City e le classi Menu e Order. Le prime due non hanno posto una particolare sfida implementativa quindi ho ritenuto non necessario approfondirle, mentre per la terza classe l'aspetto che intendo analizzare riguarda come ho scelto di gestire l'aggregazione di più menu in un singolo ordine.

Inizialmente avevo utilizzato una lista di *Menu* con relativi metodi di inserimento/rimozione, in questo modo per conoscere la quantità di uno stesso menu in un ordine era necessario un metodo a parte che filtrava tale lista in base al nome del *Menu* da ricercare e ritornava la sua *size*. Questa non mi sembrava un'implementazione ideale, per questo ho scelto di sostituire la lista con una mappa da Menu a Integer, così da mappare i menu alla loro quantità in modo da facilitare l'accesso a questo dato. Questo ha portato anche a modificare i metodi di inserimento/rimozione, che prima erano dei semplici *"list".add()/.remove()*, come mostrato in seguito. In tali funzioni inizialmente si aggiornavano ogni volta anche i campi *price* e *size* dell'ordine mentre adesso questi valori vengono calcolati direttamente dai rispettivi metodi *"getX()"* tramite degli opportuni stream sulla mappa.

Figura 2.5: Modifiche principali della classe `OrderImpl`

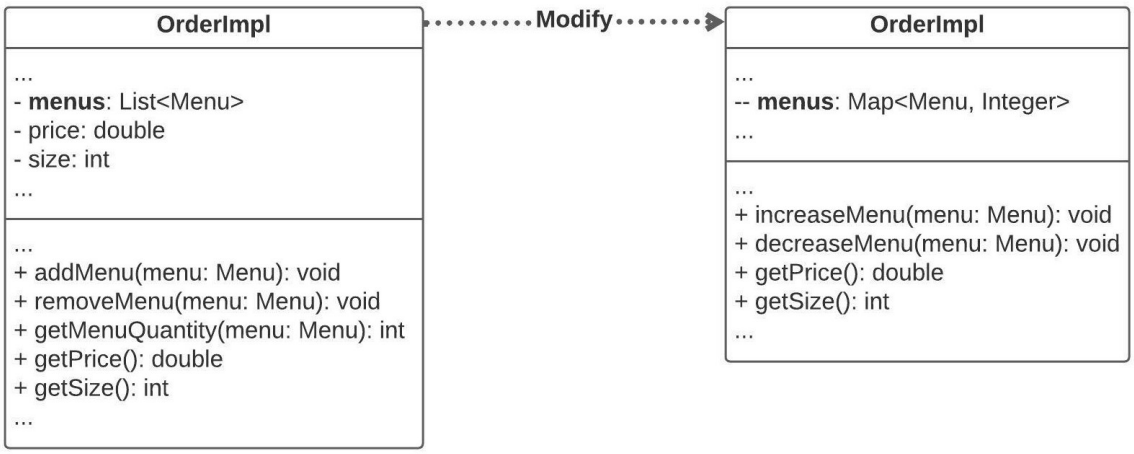


Figura 2.6: Modifica dei metodi di inserimento e rimozione

```
@Override
public void increaseMenu(Menu menu) {
    // Se la mappa contiene già tale chiave incrementiamo il suo valore
    if (menus.containsKey(menu))
        this.menus.put(menu, menus.get(menu)+1);
    // Altrimenti inseriamo una nuova chiave con valore 1
    else
        this.menus.put(menu, 1);
}

@Override
public void decreaseMenu(Menu menu) {
    // Se la mappa contiene tale chiave e il suo valore è > 0 lo decrementiamo
    if (menus.containsKey(menu) && menus.get(menu)>0) {
        this.menus.put(menu, menus.get(menu)-1);
        // Se dopo il decremento il valore è pari a 0 rimuoviamo la chiave
        if (menus.get(menu)==0)
            menus.remove(menu);
    }
}
```

Design Controller

A livello implementativo il mio contributo alla classe *ManagerImpl*, il controller del nostro sistema, riguarda il metodo *refreshWaitingOrder()* necessario per una delle mie interfacce grafiche. Tale metodo verifica per ogni ordine in attesa se questo possa essere assegnato al *Rider* passato come parametro di input. In caso di esito positivo aggiunge l'ordine a tale

Rider e ad una lista di supporto. Alla fine dell'iterazione rimuove tutti gli ordini contenuti in tale lista dalla lista d'attesa. Il metodo ritorna *true* se la size della struttura dati di supporto è maggiore o uguale a 1, questo implica che almeno un ordine in attesa è stato smistato, *false* altrimenti.

Riguardo al controller mi sono occupato anche del suo testing cercando il più possibile di creare situazioni varie che verificassero che l'assegnamento di ordini rispettassero le regole poste dalla nostra specifica.

Figura 2.7: Metodo *refreshWaitingOrder()*

```
@Override
public boolean refreshWaitingOrder(Rider freeRider) {
    // Lista di supporto
    List<Order> noMoreWaitingOrders = new ArrayList<Order>();
    // Iterazione su tutti gliordini in attesa
    this.getWaitingOrders().forEach(o ->{
        // Verifico se l'ordine verrebbe assegnato veramente al freeRider
        if(this.canBeAssigned(o).isPresent())
            if(this.canBeAssigned(o).get().equals(freeRider)) {
            noMoreWaitingOrders.add(o);
            freeRider.addOrder(o);
        }
    });
    // Rimuovo tutti gli ordini assegnati dalla lista d'attesa
    this.waitingOrders.removeAll(noMoreWaitingOrders);
    return(noMoreWaitingOrders.size()>=1);
}
```

Design View

Riguardo la view la mia implementazione comprendeva l'interfaccia grafica per gestire gli ordini da consegnare e quella per la selezione dei menu.

Nella prima, oltre all'aver utilizzato delle mappe per associare ad ogni rider le sue rispettive aree di testo, bisognava gestire l'aggiornamento dell'interfaccia nel momento in cui un nuovo ordine fosse stato smistato fra i fattorini o messo in attesa. Una prima implementazione era stata quella di passare a *ViewRecap*, dove l'ordine viene effettivamente smistato, un'istanza di *ViewWorker* in modo da segnalare il suo invio. Questa metodologia però mi è sembrata poco funzionale poiché per arrivare fino a *ViewRecap* il riferimento veniva passato nei costruttori di tutte le altre view senza mai essere utilizzato.

Ho quindi deciso di rendere *ViewWorker* un Singleton, ossia una classe che possa essere istanziata una e una sola volta. Questo ha permesso un accesso semplificato in *ViewRecap* tramite il metodo *getIstance()* che restituisce l'unico oggetto esistente di tale classe se già presente altrimenti provvede a crearlo. Inizialmente tale metodo riceveva come parametro di input un oggetto ManagerImpl che fungeva da controller, ma informandomi sull'utilizzo

di questo pattern ho capito che formalmente un Singleton non riceve parametri all'inizializzazione. Ho quindi creato un metodo a parte, *initView()*, che si occupa della inizializzazione dei parametri necessari.

Figura 2.8: Prima implementazione Singleton

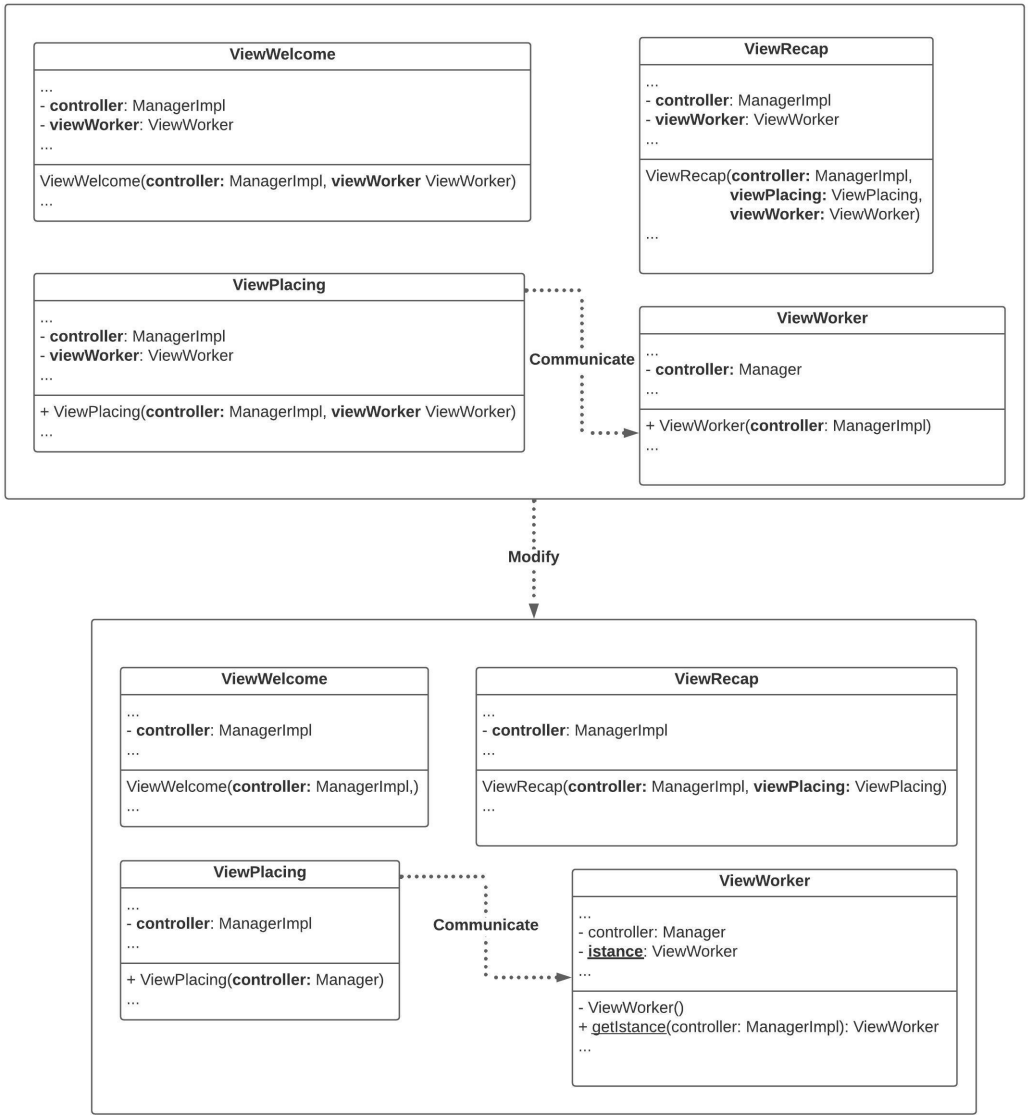
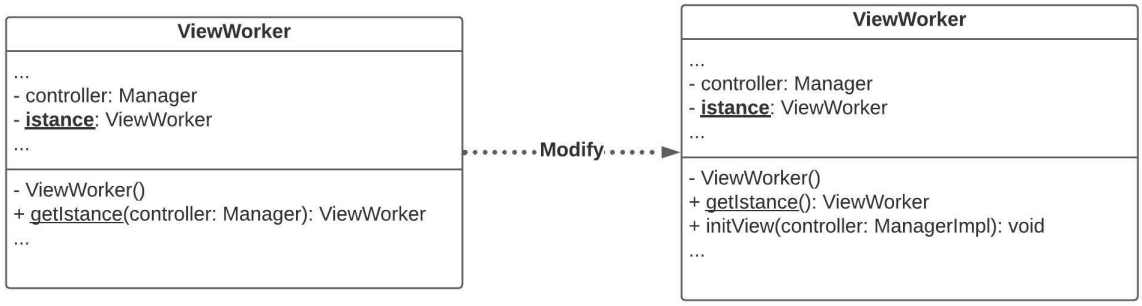


Figura 2.9: Seconda implementazione Singleton

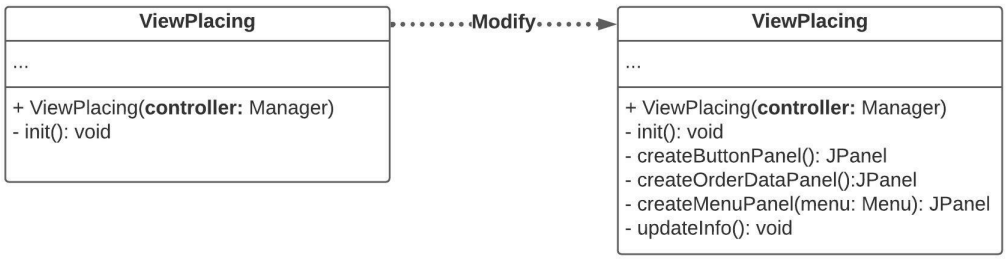


La seconda interfaccia mi è stata assegnata dopo l'abbandono di uno dei membri del team, quindi il mio lavoro di implementazione è stato solo parziale.

Fondamentalmente mi sono occupato di una revisione e pulizia del codice già implementato precedentemente. Erano presenti varie ripetizioni di codice, dovute all'aggiornamento delle textArea di prezzo e spazio dell'ordine qualora si modificasse il suo contenuto, che ho eliminato creando un metodo apposito per lo scopo, *updateInfo()*.

Successivamente, in maniera simile a come avevo gestito il lavoro nell'altra mia view, ho scomposto il metodo *init()*, il quale andava a creare ogni parte dell'interfaccia, in vari metodi di supporto in modo che ogni JPanel avesse la sua funzione dedicata e il tutto risultasse più leggibile.

Figura 2.10: Modifiche a `viewPlacing`



Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per testare le funzionalità core del nostro progetto si è fatto uso di JUnit 5 per definire dei test per ogni classe. Queste batterie di test hanno il compito di simulare il funzionamento dei vari metodi in situazioni differenti e confrontarlo con il comportamento che ci si aspetterebbe in quel contesto.

L'implementazione dei vari test cercava di seguire il più possibile l'implementazione dei metodi in modo da individuare e risolvere al più presto eventuali problemi.

3.2 Metodologia di lavoro

All'inizio tutto il team ha avuto delle sessioni di brainstorming per ideare possibili proposte di specifica, sulle quali operavamo una piccola analisi preventiva in modo da valutare se si trattasse di un problema adatto agli insegnamenti del corso.

Una volta individuata la specifica adatta, proposta ed approvata dal docente, abbiamo iniziato con la fase di analisi del problema, momento nel quale si è lavorato sempre in gruppo.

Abbiamo studiato il problema definendo innanzitutto i casi d'uso, che ci hanno permesso di individuare i diversi attori che avrebbero interagito con l'applicativo, inquadrare bene i requisiti principali e le interazioni degli utenti col sistema.

Successivamente si è descritto il modello del dominio applicativo, cominciando quindi a farci un'idea delle entità principali del problema e di come si relazionano fra loro.

Abbiamo poi individuato quelle che sarebbero state le interfacce del nostro programma e, analizzando la loro complessità, ci siamo suddivisi i vari compiti implementativi nella maniera che ci risultava più equa, ciò ci ha permesso un proseguimento allo sviluppo in modo parallelo.

Al sorgere di un nuovo problema che richiedeva una nuova scelta di progetto, il team si riuniva per discuterne. Per scelte minori ovviamente ognuno era libero di decidere per la propria porzione di progetto; periodicamente ogni membro aggiornava brevemente i colleghi sul proprio operato, permettendo uno scambio di opinioni e consigli sulle decisioni prese singolarmente.

Strumenti di collaborazione

Per il sistema di controllo della versione si è fatto uso di Git. Come metodologia, trattandosi di un piccolo gruppo con alta fiducia tra i membri e nessuna gerarchia, si è optato per un semplice singolo branch.

Per quanto riguarda il codice, per tenere traccia del da farsi si è fatto uso della funzione "Tasks" di Eclipse, la quale organizza i commenti "TODO" in un'apposita scheda così da avere una lista delle cose da fare organizzata per file e ordinabile per priorità.

Non meno importanti per la collaborazione sono stati OverLeaf per la stesura collettiva della relazione in Latex e Lucid per la creazione di schemi condivisi.

3.2.1 Porzione di Procicchiani Luca

Implementazione delle funzionalità relative ai ristoranti, ai rider e al Controller.

Implementazione dei relativi test.

Sviluppo delle interfacce grafiche per la schermata di benvenuto del cliente e quella di riepilogo dell'ordine.

3.2.2 Porzione di Tombari Giacomo

Implementazione delle funzionalità relative alle città, ai menù e agli ordini.

Implementazione dei test per tali classi, più la classe ManagerImpl.

Sviluppo dell'interfaccia grafica usata dai Rider e di quella per la selezione dei menu.

3.3 Note di sviluppo

3.3.1 Note di Procicchiani Luca

All'interno della mia porzione ho fatto uso dei seguenti aspetti avanzati di implementazione:

- Uso di Stream nell'algoritmo di assegnamento di ordini e all'interno di `RiderImpl`.
- Uso di espressioni lamdba, nella creazione di eventListener per i bottoni nelle mie view.
- Uso di Optional all'interno del Controller per campi che possono essere vuoti.

3.3.2 Note di Tombari Giacomo

All'interno della mia porzione ho fatto uso dei seguenti aspetti avanzati di implementazione:

- Uso di Stream nelle classi `OrderImpl` e `ViewWorker`.
- Gestione di oggetti Optional.
- Uso di lamdba per migliorare la leggibilità del codice nelle view.
- Metodo *refreshWaitingOrder()* di *ManagerImpl*.