

Relazione relativa alla progettazione di Food in Town, un gestore di consegne a domicilio

Componenti del gruppo: Enrico Ceccolini 668975 , Eric Valeri 654873.

## 1 Analisi del problema:

FoodInTown è stata realizzata con lo scopo di simulare un servizio web ristorativo in grado di fornire una nuova modalità di prenotazione di consegne a domicilio.

L'idea nasce dall'esigenza di sostituire la vecchia modalità generalmente offerta dai ristoranti, cioè la prenotazione per via telefonica. Quest' ultima infatti prevede tempi di ricezione delle prenotazioni\* non definiti (dipendenti da un fattore umano) i quali creano svantaggi in primo luogo ai ristoranti (frequenza ricezione ordini limitata) ma anche ai clienti, in quanto il tempo di attesa per un'ordinazione è legato alla disponibilità della linea che spesso è occupata.

FoodInTown risolve questi problemi gestendo la ricezione simultanea di prenotazioni effettuate da più clienti, fornendo a quest' ultimi la possibilità di conoscere in dettaglio il menù d'asporto di un ristorante e di crearsi quindi in autonomia l'ordine per poi spedirlo.

FoodInTown prevede dunque due diversi applicativi GUI-based, uno ad uso dei clienti ed uno ad uso dei ristoranti, in grado di comunicare esclusivamente tramite una terza applicazione che andrà a simulare un web server, unico che potrà accedere ai dati persistenti.

### 1.1 ) Nello specifico un generico cliente dovrà essere in grado di:

- Registrarsi al servizio;
- Cercare ristoranti per tipologia di cucina;
- Navigare tra i menù dei ristoranti;
- Creare il proprio carrello aggiungendo prodotti;
- Personalizzare se possibile i prodotti da acquistare;
- Inviare ordinazioni a uno o più ristoranti;
- Votare i prodotti acquistati.

### 1.2 ) Un generico ristorante dovrà essere in grado di:

- Visualizzare le prenotazioni ricevute in ordine cronologico;
- Mantenere aggiornata la lista degli ordini ricevuti run-time;
- Confermare l'avvenuta acquisizione di un ordine;
- Modificare la disponibilità di un prodotto del proprio menù;
- Gestire più menù pur avendone solo uno attivo;
- Modificare i propri menù ed i prodotti contenuti;
- Definire e modificare i propri orari di consegna a domicilio.

- Per tempo di ricezione di una prenotazione si intende il tempo impiegato dal ristorante per ottenere le informazioni necessarie per poter effettuare il proprio servizio di consegna a domicilio.

### 1.3 ) Analizzando il problema abbiamo preso le seguenti decisioni:

Utenti non registrati hanno la possibilità di visualizzare la lista dei ristoranti ed i relativi menù senza però poter effettuare ordini, non possono esistere due clienti con lo stesso numero di telefono, un cliente che ha già effettuato un ordine verso un ristorante non può inviargliene altri fino alla avvenuta conferma del precedente, non possono esistere due ristoranti con lo stesso nome.

L'applicazione del ristorante possiede internamente le credenziali di accesso al server in quanto è stata concettualmente pensata per essere personalizzata e distribuita ad ogni ristorante.

Dovendo simulare una comunicazione via web, il nostro server non invierà notifiche di alcun genere senza essere interrogato.

L'aggiornamento della lista degli ordini avverrà tramite l'utilizzo di un thread il quale ad intervalli di tempo regolari interrogherà il server per ottenere i nuovi ordini a lui recapitati.

#### Conclusione analisi:

L'applicazione dovrà garantire la persistenza delle informazioni inserite dai vari utilizzatori oltre alla consistenza delle stesse, viste le varie problematiche di sincronizzazione createsi dovendo gestire contemporaneamente clienti e ristoranti.

## **2 Progettazione architetturale:**

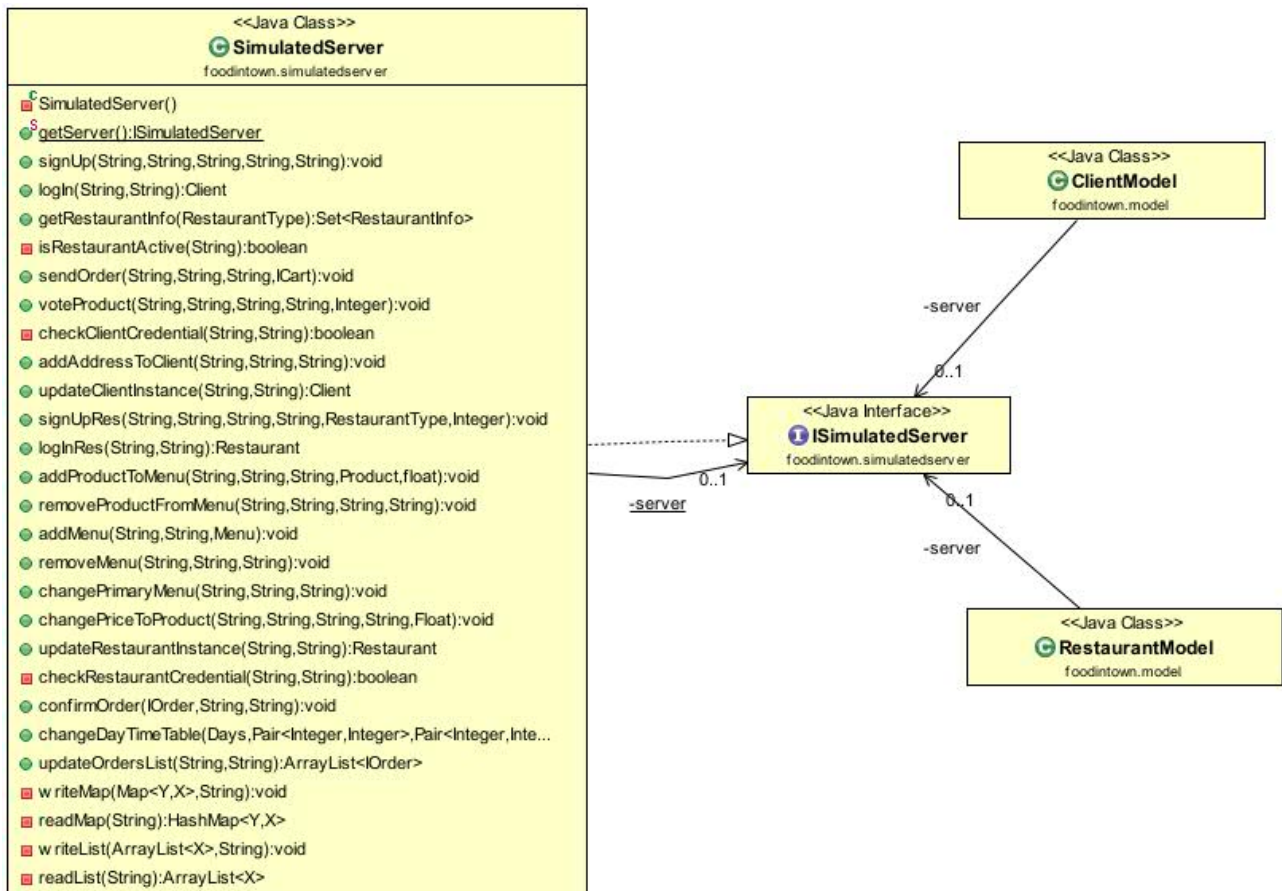
In questa fase si è effettuato lo studio delle varie entità introdotte e delle loro relazioni, operazione cruciale vista la necessità di mantenere separate le due applicazioni. Queste infatti dovranno poter accedere esclusivamente a dati di loro interesse, ad esempio un ristorante conosce dei propri clienti solo le informazioni utili alla consegna, e allo stesso modo un cliente conosce solo i menù messi a disposizione dai vari ristoranti più le informazioni a lui necessarie per decidere in quale ristorante effettuare un ordine.

A garantire questa divisione è la classe che simula il web server, la quale memorizza i dati delle due applicazioni in file binari e gestisce l'intera comunicazione tra le stesse.

Questa comunicazione è stata realizzata inserendo la stessa istanza del server in entrambi gli applicativi. Per questo motivo è risultato particolarmente adatto utilizzare il pattern Singleton a definizione della classe server che abbiamo chiamato SimulatedServer.

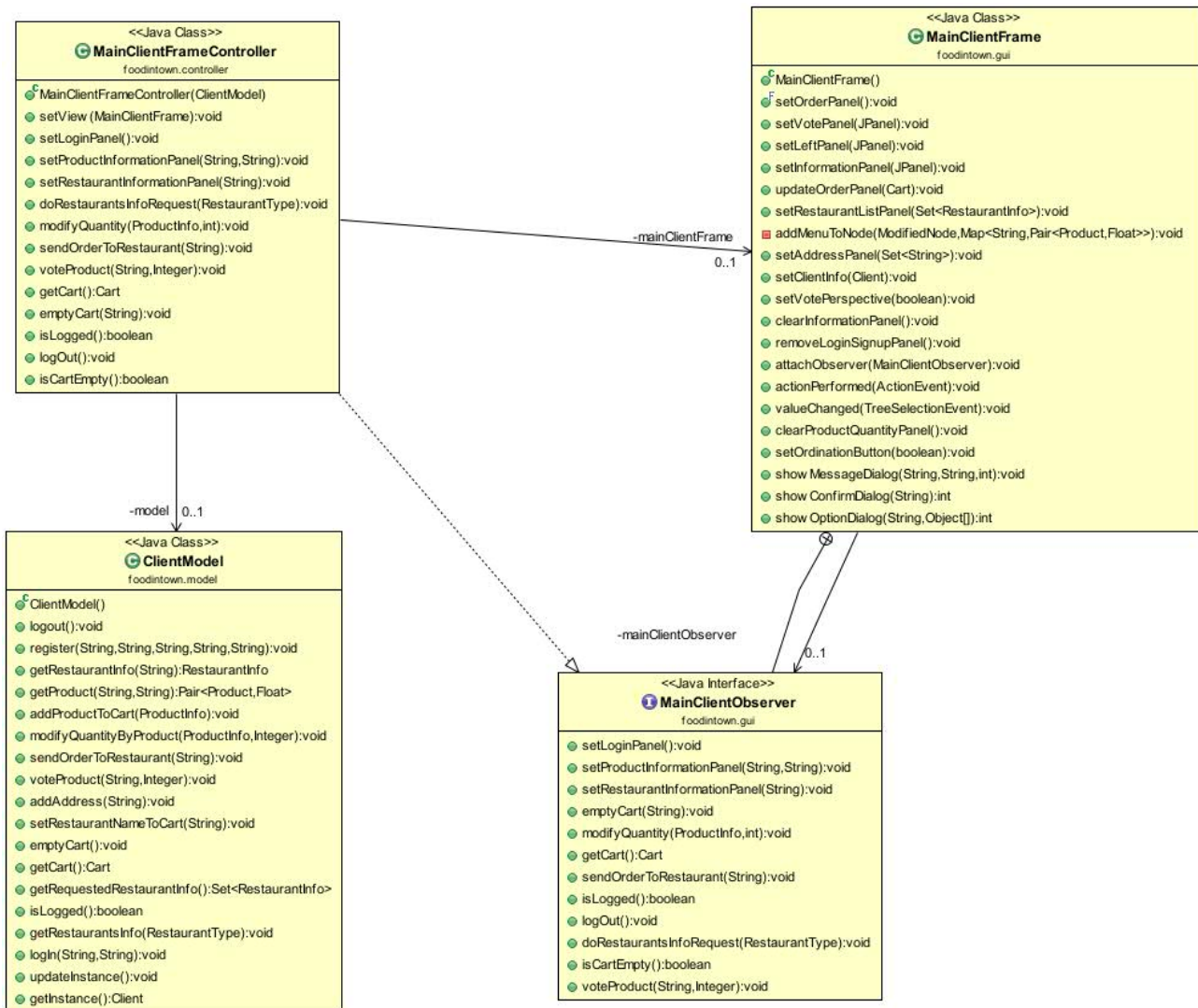
Il singleton, infatti, è un pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata una e una sola istanza, e di fornire un punto di accesso globale ad essa.

## 2.1 ) UML che mostra l'utilizzo del pattern Singleton:

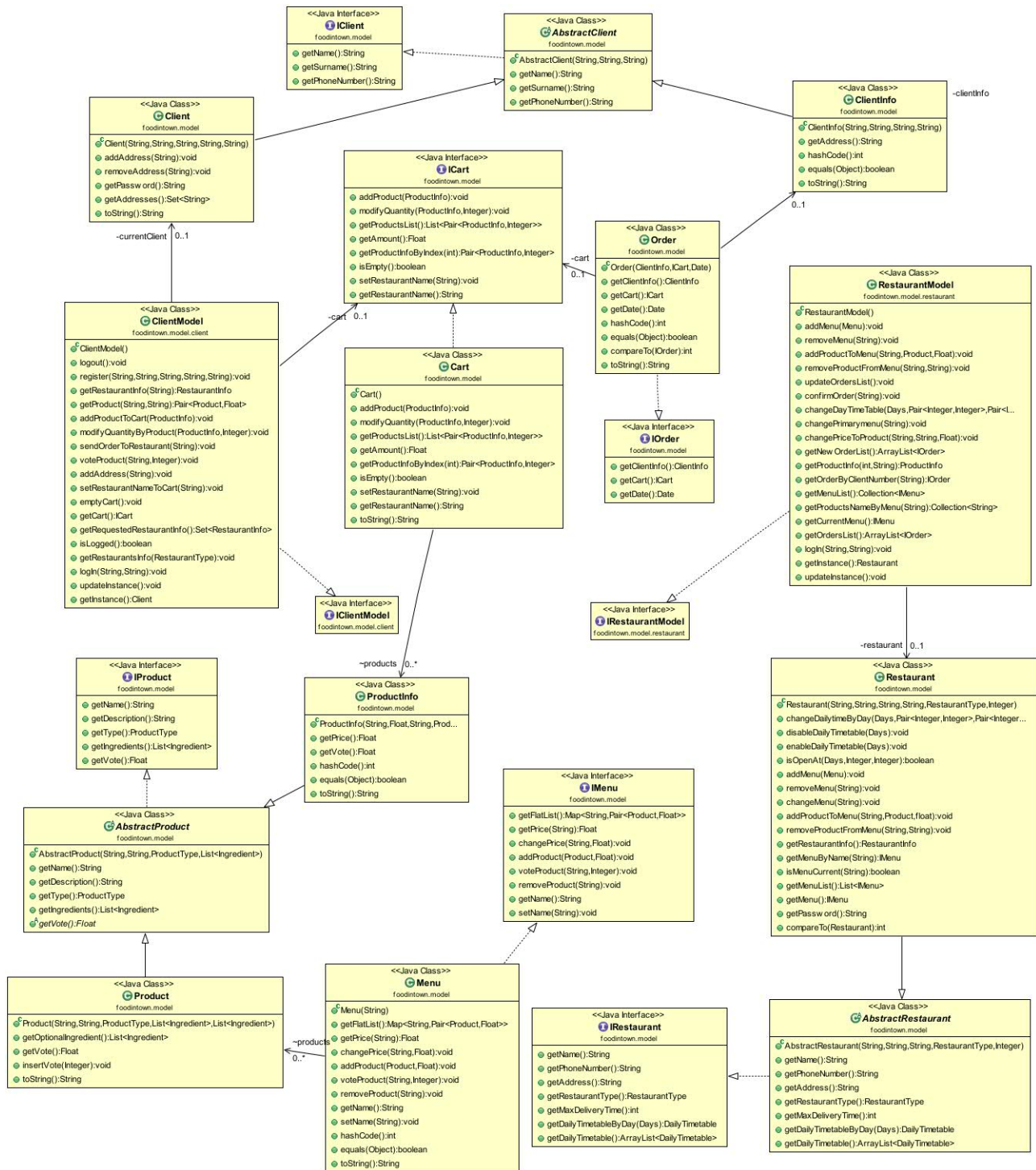


Definita la comunicazione, parliamo ora della progettazione delle applicazioni per cliente e per ristorante. L'intera fase di sviluppo di quest'ultime è stata guidata dall'uso del pattern architetturale Model-View-Controller il quale prevede la suddivisione tra rappresentazione del modello di dominio (model), interfaccia utente (view) e controllo dell'interazione uomo-macchina (controller). Abbiamo scelto la più comune delle implementazioni di MVC integrandolo con il design pattern Observer che si basa su uno o più oggetti, chiamati osservatori, i quali vengono registrati per gestire un evento che potrebbe essere generato dall'oggetto "osservato". Il pattern prevede anche un' implementazione concreta dell' observer che nello specifico risulterà essere un controller.

## 2.2) Esempio UML che mostra lo schema MVC per l'applicazione cliente:



## 2.3) Diagramma UML delle classi relative alla parte del modello dell'applicazione:



### Descrizione degli aspetti principali:

- **ISimulatedServer & SimulatedServer:** Interfaccia e implementazione concreta del nostro server simulato. Come già spiegato è la classe che gestisce la comunicazione tra le due applicazioni attraverso file binari organizzati nel seguente modo:
  - clients.dat : file contenente una Map<String,Client> che rappresenta l'insieme dei clienti registrati al servizio <numero telefono, cliente>.
  - restaurants.dat : file contenente una Map<String,Restaurant> che rappresenta l'insieme dei ristoranti aderenti al servizio <nome ristorante, ristorante>.
  - Un file binario per ogni ristorante presente su restaurants.dat contenente una List<Order> che rappresenta la lista degli ordini che il ristorante ha ricevuto.
- **IClientModel & ClientModel:** interfaccia utilizzata per l'applicazione cliente e la sua implementazione concreta. Contiene tutti i metodi per dialogare con il server.
- **IRestaurantModel & RestaurantModel:** interfaccia utilizzata per l'applicazione ristorante e la sua implementazione concreta. Contiene tutti i metodi per dialogare con il server.
- **IClient & AbstractClient:** interfaccia rappresentante un cliente e la sua implementazione astratta. Le classi che estendono AbstractClient sono:
  - Client: rappresenta un generico cliente registrato a foodInTown. Tale classe è accessibile esclusivamente dall'applicazione del cliente.
  - ClientInfo: racchiude esclusivamente i dati necessari ad un ristorante per poter effettuare le varie consegne a domicilio, sono quindi esclusi i dati sensibili o superflui del cliente.
- **IRestaurant & AbstractRestaurant:** interfaccia rappresentante un ristorante e la sua implementazione astratta. Le classi che estendono AbstractRestaurant sono:
  - Restaurant: rappresenta un generico ristorante aderente al servizio foodInTown. Tale classe è accessibile esclusivamente dall'applicazione del ristorante.
  - RestaurantInfo: racchiude esclusivamente i dati necessari ad un cliente per poter effettuare i propri ordini, sono quindi esclusi i dati sensibili o superflui del ristorante.
- **IOrder & Order:** interfaccia rappresentante un ordine e la sua implementazione concreta. E' composto da un carrello e dai riferimenti al cliente e al ristorante.
- **ICart & Cart:** interfaccia rappresentante un carrello e la sua implementazione concreta. E' composto dai prodotti selezionati dal cliente e dal rispettivo quantitativo.
- **IMenu & Menu:** interfaccia rappresentante un menù e la sua implementazione concreta. E' composta dalle varie associazioni prodotto/prezzo.
- **IProduct & AbstractProduct:** interfaccia rappresentante un prodotto e la sua implementazione astratta. Le classi che estendono AbstractProduct sono:
  - Product: rappresenta un generico prodotto di un ristorante.
  - ProductInfo: rappresenta un prodotto non più modificabile all'interno di un carrello.



## Specifiche grafiche:

Dovendo simulare due diverse applicazioni su un unico pc è risultato opportuno creare due frame separati, uno per il cliente e uno per il ristorante.

## Specifiche sulle viste per l'applicazione cliente:

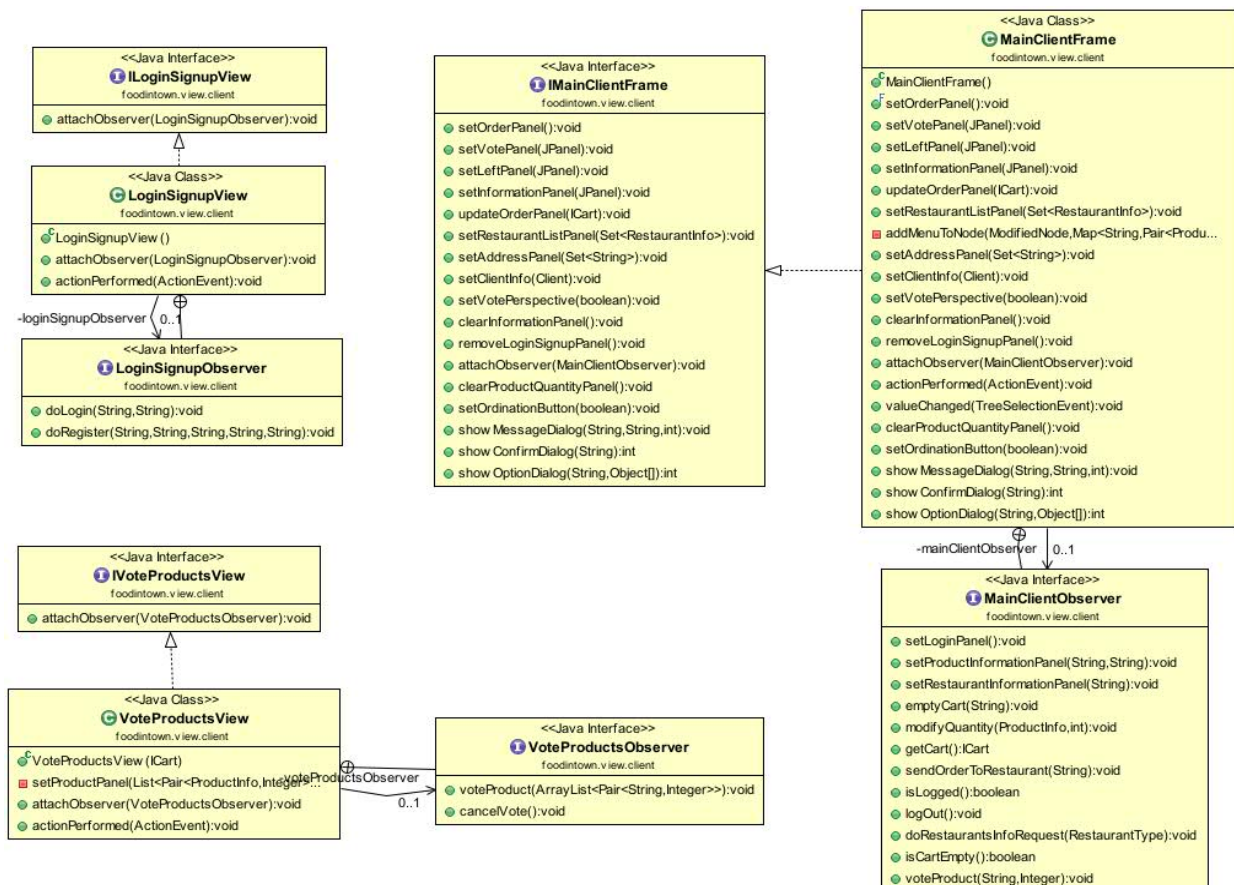
L'applicazione prevede l'utilizzo di diverse view, una principale (MainClientFrame) e altre che vengono integrate a quella principale quando vi è la necessità.

Il MainClientFrame presenta un menù nella parte superiore che può essere utilizzato dai clienti per accedere alla sezione di login o registrazione al servizio (LoginSignupView) o per effettuare una ricerca per tipologia tra i ristoranti.

La parte centrale prevede un'area utilizzata per mostrare una struttura dati ad albero contenente i ristoranti ed un'altra per visualizzare le informazioni dettagliate sul ristorante o sul prodotto selezionato visto che tramite l'albero sarà possibile visualizzare in dettaglio il menù d'asporto offerto da ogni ristorante. Se ad essere selezionato è un prodotto nella seconda area definita sarà presente un pannello (ProductInfoAndOrdinationView) che permetterà la personalizzazione del prodotto (scelta di ingredienti opzionali) e l'aggiunta di questo al proprio carrello.

La parte destra della vista principale viene utilizzata per mostrare il carrello e offre la possibilità di modificare il numero di pezzi per ogni prodotto contenuto al suo interno oltre alla possibilità di confermare il carrello e quindi effettuare l'ordine.

Una volta confermato l'ordine l'utente potrà scegliere se continuare con gli acquisti (tornando alla vista principale) oppure di dare un voto ai prodotti acquistati tramite la vista VoteProductView che sostituirà la parte destra.



### Specifiche sulle viste per l'applicazione ristorante:

Visto che l'applicazione ristorante richiede molteplici view si è deciso di implementarne una principale (MainRestaurantFrame) che le conterrà.

Tale vista presenta nella parte superiore un'area sempre visibile utilizzata per mostrare una struttura dati ad albero contenente gli ordini ricevuti dal ristorante. Selezionando gli ordini all'interno di quest' ultimo il ristorante potrà vedere in dettaglio le informazioni riguardanti l'ordine e confermarlo attraverso un pannello dinamico interno al MainRestaurantFrame che comparirà sulla destra.

L'area inferiore viene invece utilizzata per la gestione interna del ristorante. E' composta da una seconda vista (RestaurantManagementView) che tramite un menù contenente un insieme di bottoni offre la possibilità di visualizzare al suo interno la vista relativa all'operazione selezionata:

- aggiungere un menù (AddMenuView);
- aggiungere un prodotto (AddProductView);
- modificare gli orari di lavoro (ModifyTimetableView);
- selezionare un menù come primario o modificare un prodotto (ModifyProductMenuView).

### **3 Organizzazione in Package:**

Viene ora effettuata un'analisi dell'organizzazione in package dell'applicazione.

- **foodintown.model** : contiene i sorgenti necessari per l'implementazione dei modelli dell'applicazione.
- **foodintown.model.client** : contiene i sorgenti necessari per l'implementazione dei modelli dell'applicazione cliente.
- **foodintown.model.restaurant** : contiene i sorgenti necessari per l'implementazione dei modelli dell'applicazione ristorante.
- **foodintown.controller.client** : contiene i sorgenti che incapsulano il comportamento dei controller relativi alle view del cliente.
- **foodintown.controller.restaurant** : contiene i sorgenti che incapsulano il comportamento dei controller relativi alle view del ristorante.
- **foodintown.view.client** : contiene i sorgenti relativi alle view del cliente.
- **foodintown.view.restaurant** : contiene i sorgenti relativi alle view del ristorante.
- **foodintown.main** : contiene il main dell'applicazione.
- **foodintown.simulatedserver** : contiene i sorgenti relativi alla definizione del server simulato.
- **foodintown.exceptions** : contiene le classi relative a tutte le possibili eccezioni che



si possono verificare nel contesto.

- **foodintown.enumeration** : contiene le enumerazioni utilizzate all'interno dell'applicativo.
- **foodintown.test** : contiene una classe utilizzata per i test
- **foodintown.utility** : contiene le classi di utilità estrapolate dal contesto MVC perchè ampiamente utilizzate nel progetto.
  - CustomCellRenderer : classe che gestisce gli aspetti grafici di un generico JTree
  - DailyTimetable : classe che modella un generico orario di lavoro
  - ModifiedComboBox : classe utilizzata per associare una JComboBox ad un generico oggetto:
  - ModifiedNode : classe che estende DefaultMutableTreeNode, utile a rappresentare un generico nodo all'interno di un albero.
  - Pair : classe utilizzata per associare due generici valori.
  - PriceFormatterFactory : classe utilizzata per formattare una JTextField in modo appropriato per contenere un prezzo.

Il progetto prevede inoltre alcune risorse esterne che si dovranno trovare nella stessa posizione del file eseguibile jar .

## 4 Suddivisione del lavoro:

- Enrico Ceccolini :
  - responsabile dello sviluppo dei controller;
  - responsabile dello sviluppo dei modelli riguardanti l'applicazione del cliente;
  - responsabile dello sviluppo del thread nell'applicazione ristorante.
- Eric Valeri :
  - responsabile dello sviluppo delle viste;
  - responsabile dello sviluppo dei modelli riguardanti l'applicazione del ristorante.
- In comune :
  - realizzazione del server simulato;
  - eccezioni ed enumerazioni.

## 5.1 Progettazione di dettaglio: Parte di Enrico Ceccolini

### 5.1.1 ) Sviluppo controller:

Come già detto nella parte di progettazione, si è scelto di rappresentare il problema attraverso l'utilizzo del pattern architetturale Model View Controller.

Questa decisione trova in me pieno consenso poiché credo di trovarmi di fronte ad un problema nel quale è fondamentale garantire la separazione tra logica applicativa e interfaccia utente per diverse ragioni, tra cui mantenibilità del software e il suo riutilizzo.

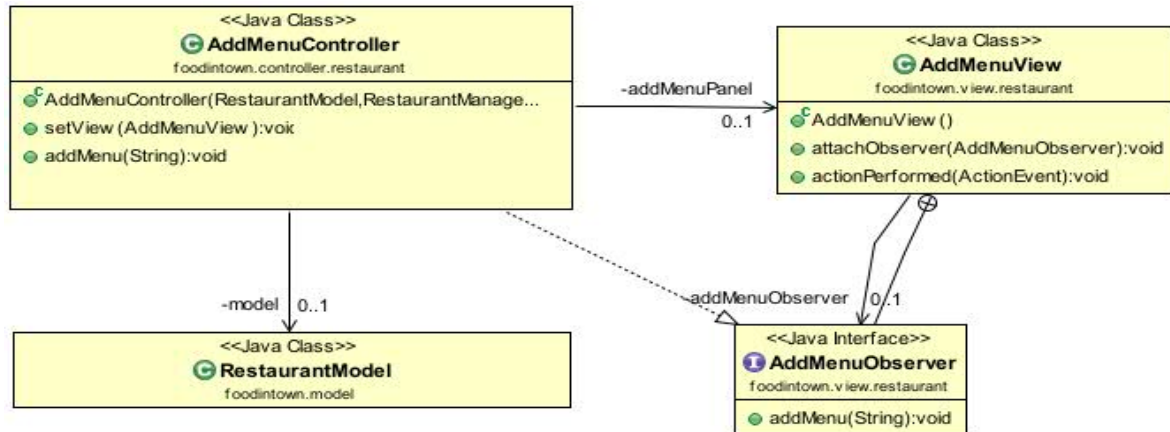
Nello specifico mi sono occupato dei vari controller presenti nell'applicazione, garantendo dunque la separazione tra modello e vista sia per l'applicazione cliente che ristorante.

Ogni controller corrisponde all'implementazione concreta dell' observer definito (attraverso

un' interfaccia) nella vista corrispondente, offrendo a quest'ultima un metodo indiretto, e quindi sicuro, per ottenere i dati che rimangono gestiti interamente dal modello. Altro vantaggio sta nella gestione in assoluta sicurezza dell'aggiornamento della view da parte del controller corrispondente.

Per l'utilizzo del pattern Observer, ad ogni controller è stato devinato un metodo “setView()” che presa in input una vista, prima ne salva l'istanza e poi su questa richiamando il metodo “attachObserver()”, si registra come suo “ascoltatore o osservatore”.

esempio di MVC integrato al pattern Observer:



### 5.1.2) Sviluppo ClientModel e relativi modelli:

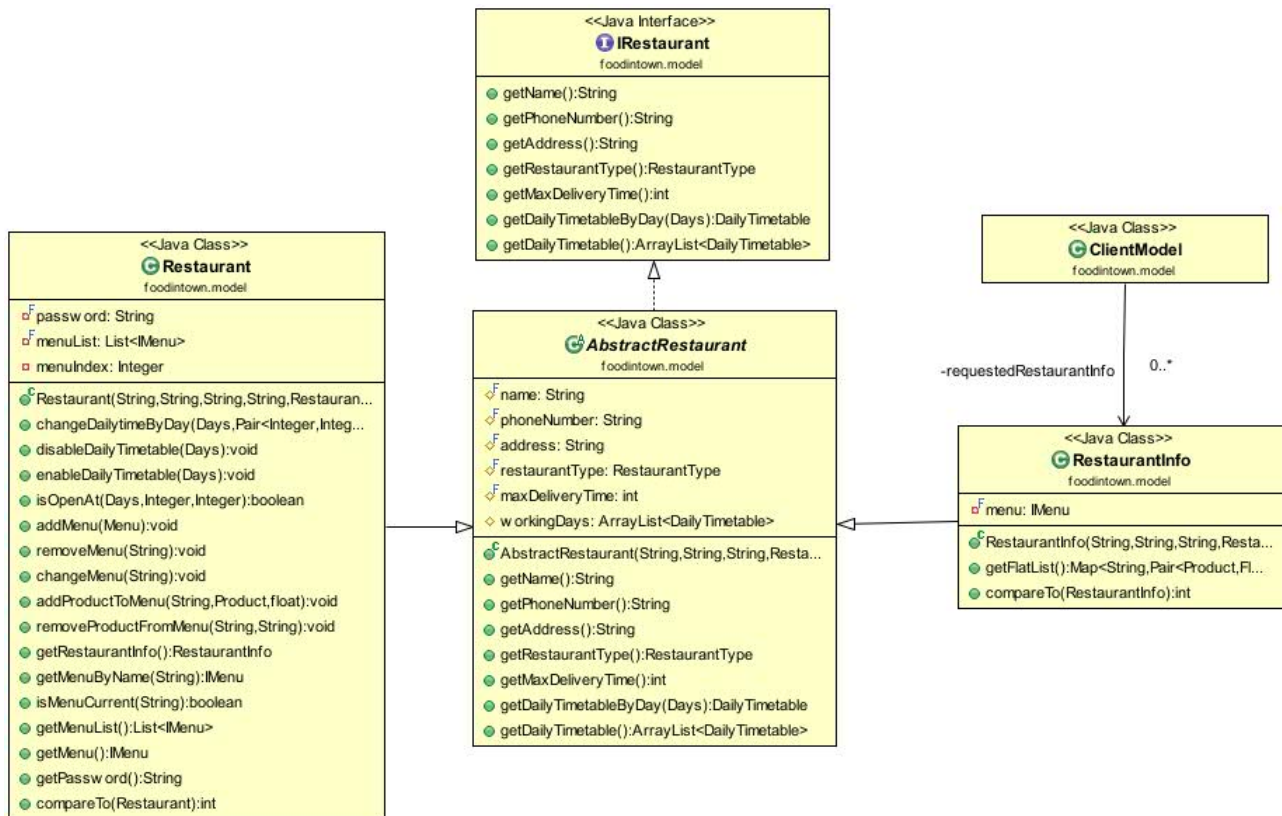
Per quanto riguarda la realizzazione del modello principale per l'applicazione cliente (ClientModel) ho scelto di mantenere al suo interno:

- l'istanza del server per ottenere accesso ai dati utili all'applicazione (oggetto di tipo `SimulatedServer`);
- l'istanza del cliente che effettua il login per simulare una connessione al server (oggetto di tipo `Client`);
- l'insieme delle informazioni dei ristoranti richiesti dall'utente per permettergli di costruire il proprio carrello in autonomia (`set<RestaurantInfo>`);
- un oggetto di tipo `Cart` che il cliente può inviare al server una volta finita la scelta e la personalizzazione dei prodotti da acquistare (oggetto di tipo `Cart`).

Ho inoltre deciso di creare in `ClientModel` un metodo per ogni abilità data ad un generico cliente come definito nell'analisi del problema al punto 1.1, tali metodi si differenziano tra quelli che hanno il compito di interagire con il server e quelli che invece svolgono esclusivamente un lavoro interno.

Nella definizione dei modelli per l'applicazione del cliente (e quindi le classi su cui agirà `ClientModel`) ho fatto particolare attenzione a fare in modo che questi non contengano informazioni non necessarie al cliente e che invece possono essere fondamentali ad un ristorante. Per fare ciò ho fatto in modo che il server, quando `ClientModel` richiede la lista dei ristoranti, non gli fornisca un ristorante nella sua interezza, ma una sua specifica “fotografia” contenente esclusivamente informazioni varie ed il menù offerto, creando una classe chiamata `RestaurantInfo`.

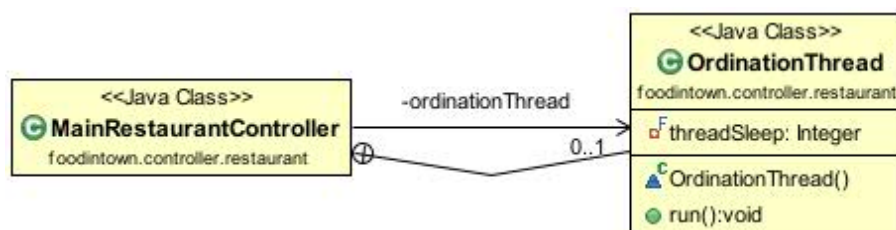
UML che mostra la differenza tra Restaurant e RestaurantInfo:



### 5.1.3) Sviluppo thread per verifica nuovi ordini ricevuti:

Per i motivi spiegati nell'analisi al punto 1.3, si è deciso che la richiesta di aggiornamento della lista degli ordini ricevuti da un ristorante sarà ad opera di un thread.

Per affrontare questo problema ho deciso di creare una Inner Class all'interno del controller principale dell'applicazione ristorante (MainRestaurantController) chiamata OrdinationThread. Tale classe andrà ad estendere Thread e grazie alla ridefinizione del metodo run() andrà ad utilizzare un metodo del controller che si occuperà di verificare la presenza nuovi ordini interrogando il server tramite il modello e, nel caso, dell'aggiornamento della view. Oltre a definirlo, il MainRestaurantController possiederà un'istanza di OrdinationThread potendo quindi avviare l'aggiornamento quando l'applicazione parte ed è pronta a presentare gli ordini ricevuti.



## 5.2 Progettazione di dettaglio: Parte di Eric Valeri

### 5.2.1 View

Per quanto riguarda la progettazione dell'interfaccia grafica di foodInTown, uno degli aspetti rilevanti è stata la scelta della disposizione degli elementi nei due frame principali che rappresentano le applicazioni (MainClientFrame: lato cliente e MainRestaurantFrame: lato ristorante). Infatti alla luce delle analisi sopra citate, sarebbe risultato molto sconveniente in termini di correttezza e leggibilità del codice includere all'interno dei due frame tutte le operazioni relative alla propria applicazione. Per questo motivo l'approccio scelto per la costruzione delle due view è stato di tipo gerarchico, cioè una vista principale (estensione di JFrame o JPanel) sempre visibile e tante altre viste (estensioni di JPanel) che vengono mostrate dinamicamente solo al manifestarsi di un particolare evento gestito dalla vista principale (una vista che è "figlia" di un'altra può a sua volta gestire viste di più alto livello.).

Avendo scelto un'implementazione dell'intero progetto basata sul pattern architetturale model-view-controller, foodInTown prevede una netta separazione tra l'interfaccia grafica e i dati da manipolare, è dunque necessario associare ad ogni vista del progetto un rispettivo controller. Questa fase è stata realizzata utilizzando il pattern Observer, infatti è stata definita un'interfaccia all'interno di ciascuna delle viste presenti in foodInTown. Tali interfacce definiscono i metodi fondamentali che una classe deve avere per poter fungere da controller per la vista stessa, questi metodi vengono quindi invocati a seguito di un evento verificatosi all'interno della GUI, eventi che vengono intercettati dalle viste stesse le quali implementano le opportune interfacce.

Una volta scelta l'architettura che il progetto avrebbe dovuto avere, si è pensato alle strutture dati da implementare, infatti, le due applicazioni necessitano di mostrare all'utente una serie di oggetti opportunamente raggruppati: nell'applicazione cliente si tratta dei ristoranti e dei loro menù, mentre nell'applicazione ristorante si tratta degli ordini ricevuti. Inoltre sia i clienti che i ristoranti devono poter interagire con i vari oggetti ottenendo informazioni aggiuntive su di essi. Volendo seguire nel modo più corretto possibile una logica MVC si è cercato di limitare al minimo la presenza di strutture dati all'interno delle viste, implementandone solo una per ciascuna applicazione, rendendole così più dinamiche e riutilizzabili.

Analizzato il problema, la scelta più efficace è stata l'implementazione di una struttura ad albero (JTree) a tre livelli in entrambe le applicazioni:

- albero applicazione cliente: rappresenta una lista di ristoranti ognuno dei quali contiene una o più categorie di prodotti che a loro volta raggruppano i prodotti. (ristorante->categoria->prodotto)
- albero applicazione ristorante: rappresenta una lista di ordini (composti da: nome cliente, indirizzo e orario) ognuno dei quali contiene i prodotti opportunamente raggruppati in categorie. (ordine->categoria->prodotto)

Questa struttura dati si è rivelata particolarmente appropriata alla situazione sia per il fatto che si adatta alla perfezione alle esigenze delle due applicazioni, sia da un punto di vista puramente estetico, infatti tramite l'implementazione di un treeCellRenderer personalizzato (foodintown.utility.CustomTreeCellRenderer) è risultato agevole modificarne le proprietà grafiche.

Le altre aree delle due viste principali sono composte da pannelli che possono essere sia oggetti costruiti all'interno della vista stessa, sia contenere al loro interno vere e proprie viste.

MainClientFrame (estensione di JFrame): vista principale dell'applicazione cliente, permette la visualizzazione dell'albero sopra citato e la modifica del carrello del cliente ed offre la possibilità di inviare ordini verso un ristorante, inoltre gestisce:

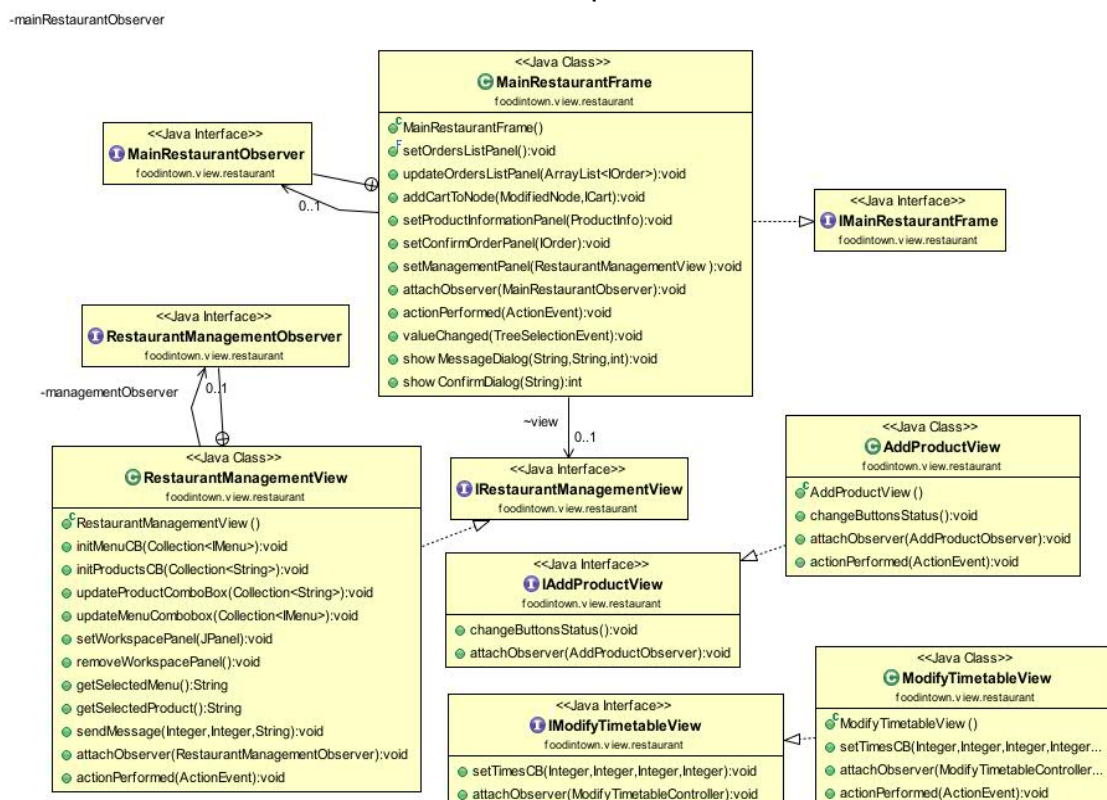
- LogInSignUpView (estensione di JPanel): vista che si occupa della sezione di accesso e registrazione dei clienti.
- VoteProductView (estensione di JPanel): vista che si occupa della votazione dei prodotti acquistati.
- ProductInfoAndOrdinationView (estensione di JPanel): vista che si occupa dell'inserimento di un prodotto all'interno del carrello e di fornire all'utente tutte le informazioni relative allo stesso.

MainRestaurantFrame (estensione di JFrame): vista principale dell'applicazione ristorante, permette la visualizzazione in ordine cronologico degli ordini e la conferma di questi ultimi, inoltre gestisce:

- RestaurantManagementPanel (estensione di JPanel): vista composta da due parti, una statica (principalmente bottoni) e una dinamica nella quale si alternano diverse viste a seconda dell'operazione selezionata. Consente di eliminare prodotti e menù da un ristorante ed inoltre gestisce:
  - AddMenuView: vista che si occupa dell'aggiunta di un menu al ristorante.
  - AddProductView: vista che si occupa di aggiungere un prodotto ad un menu del ristorante.
  - ModifyTibleTableView: vista che si occupa di modificare gli orari di lavoro dei vari giorni della settimana.
  - ModifyProductMenuView: vista che si occupa di selezionare un menù come primario e di cambiare il prezzo ad un prodotto.

UML di esempio che mostra in parte l'architettura di viste nell'applicazione ristorante: come precedentemente spiegato, le viste AddMenuView e ModifyTimetableView (qui prese come esempio per chiarire la struttura gerarchica delle viste) vengono inserite all'interno di RestaurantManagementView.

Nel diagramma UML però, non è possibile vedere questa relazione in quanto tutte le viste vanno a sostituire dinamicamente lo stesso pannello.



### 5.2.2 RestaurantModel

RestaurantModel rappresenta il modello principale dell'applicazione lato ristorante, ed è l'unica classe di tale applicazione a contenere l'istanza del server (SimulatedServer) per questo motivo RestaurantModel include anche un'istanza della classe Restaurant (che modella un generico ristorante) la quale rappresenta il ristorante loggato all'applicazione. Essendo la classe che simula la comunicazione del ristorante con il server (contiene i dati del ristorante loggato e l'istanza del server) al suo interno contiene tutti i metodi e le strutture dati utilizzate per fornire all'utente la possibilità di compiere le operazioni citate nell'analisi al punto 1.2.

## 6 Testing:

Nel corso dello sviluppo del progetto abbiamo mantenuto aggiornata una classe di test (foodintown.test.Test) che presenta un metodo con il quale è stato possibile verificare il corretto funzionamento delle nuove funzionalità implementate.

Grazie a questa classe è possibile effettuare inoltre interrogazioni al server in modo diretto, semplificando il processo di debug dello stesso non dovendo attraversare in ogni occasione le classi incluse nell'architettura mvc.

Esempio utilizzo dell'applicazione:

Prima di avviare l'applicazione verificare la presenza nella stessa cartella dell'eseguibile delle risorse esterne: data/clients.dat, data/restaurants.dat e orders/Antica Rosticceria\_orders.dat. Se non presenti verrà visualizzata una JDialog contenente un messaggio di errore.

L'applicazione ristorante è preimpostata per gestire il ristorante Antica Rosticceria, per questo motivo per poter visualizzare un ordine sarà necessario che quest' ultimo venga indirizzato verso Antica Rosticceria dall'applicazione cliente.

Se si vuole testare l'applicazione senza registrare un nuovo cliente ci si può loggare utilizzando i seguenti dati: Numero di telefono = 3334445557; Password = password .

## 7 Note finali:

Lo sviluppo del progetto si è rivelato lineare. Gli unici cambiamenti apportati dopo l'analisi non hanno riguardato la stesura del codice, ma esclusivamente la suddivisione dei compiti all'interno del gruppo. La causa di tale cambiamento è stata l'inesperienza nell'utilizzo di sistemi di sviluppo sinergico del codice.

Una volta terminata l'analisi del problema è stata definita una tabella di lavoro che ci ha permesso di sviluppare parallelamente alcune funzionalità del progetto sincronizzandole una volta terminate. Per l'intera durata del progetto è stata adottata questa strategia di lavoro.

L'unico bug particolarmente complicato da risolvere è stato causato da un errore di concorrenza che si verificava durante l'aggiornamento automatico dell'albero contenente gli ordini. Tale bug, che era causato da errori di sincronizzazione tra il thread OrdinationThread e l' EventDispatcherThread, non influiva sul corretto funzionamento lato utente.

Pur sapendo che nella realtà un servizio come Food in Town dovrebbe offrire molte più funzionalità, siamo soddisfatti del risultato ottenuto visto il numero di ore limitate.