Table Service:
sistema per la gestione di un ristorante

Componenti del Gruppo: *Biagini Piero, Nobile Marco*

Analisi del problema

Applicativo per la gestione elettronica degli ordini all'interno di un ristorante/bar/fast-food.
È richiesta la gestione di due differenti tipi di utenza, con relative funzionalità differenziate, in particolare *Cucina* e *Cameriere*.
Nello specifico, la *Cucina* deve essere in grado di:

- Creare un menu, formato da una serie di piatti e bevande con i seguenti attributi:
  - Codice identificativo
  - Nome
  - Prezzo
- Ricevere gli Ordini creati dal *Cameriere*, tramite protocollo TCP/IP
- Visualizzare i dettagli di ogni ordine ricevuto
- Impostare ogni pietanza/bevanda contenuta all'interno di un ordine come "Pronta"
- Inviare una notifica al cameriere, tramite protocollo TCP/IP, per avvisarlo che un certo ordine è pronto per essere servito

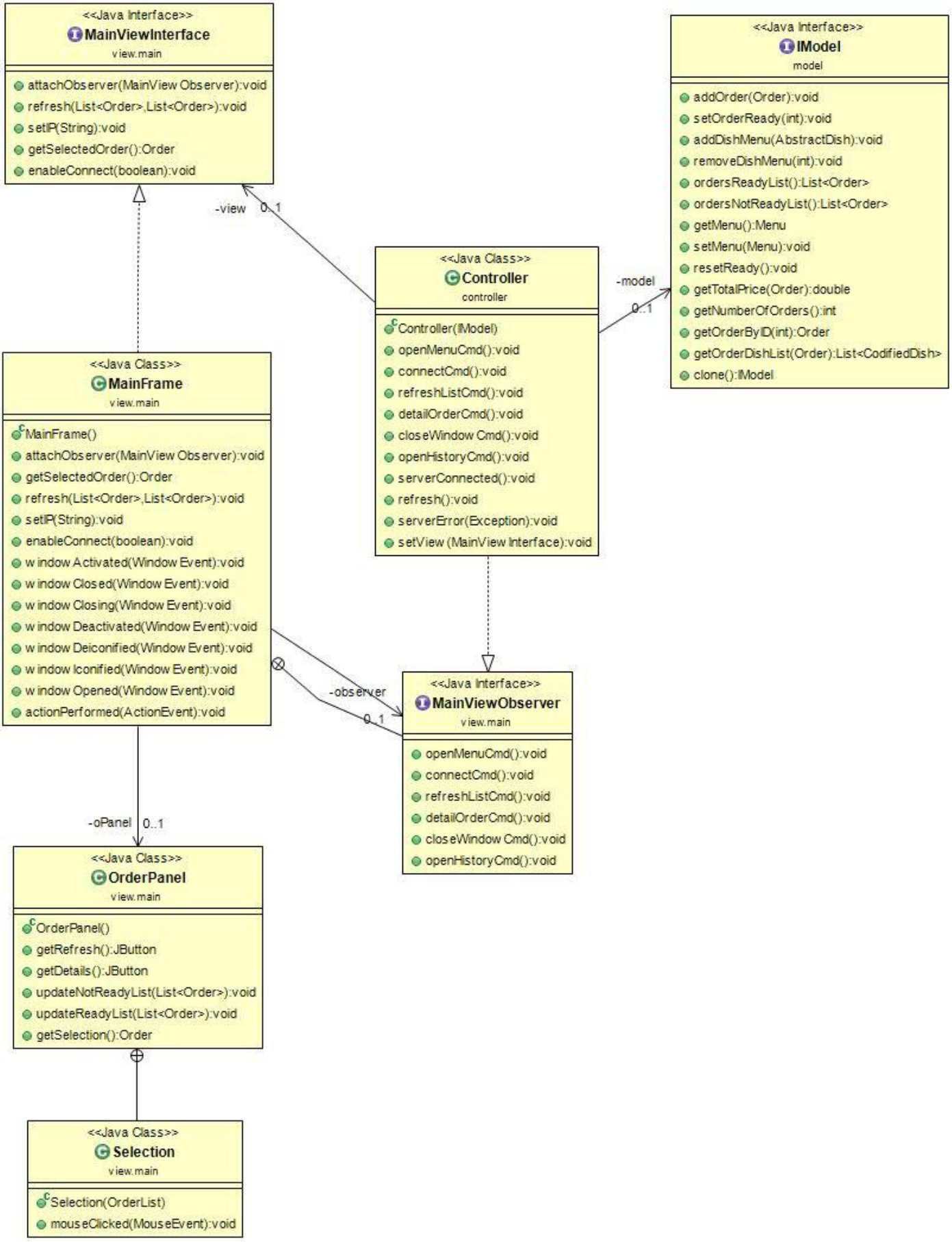
Il *Cameriere* deve essere in grado di:

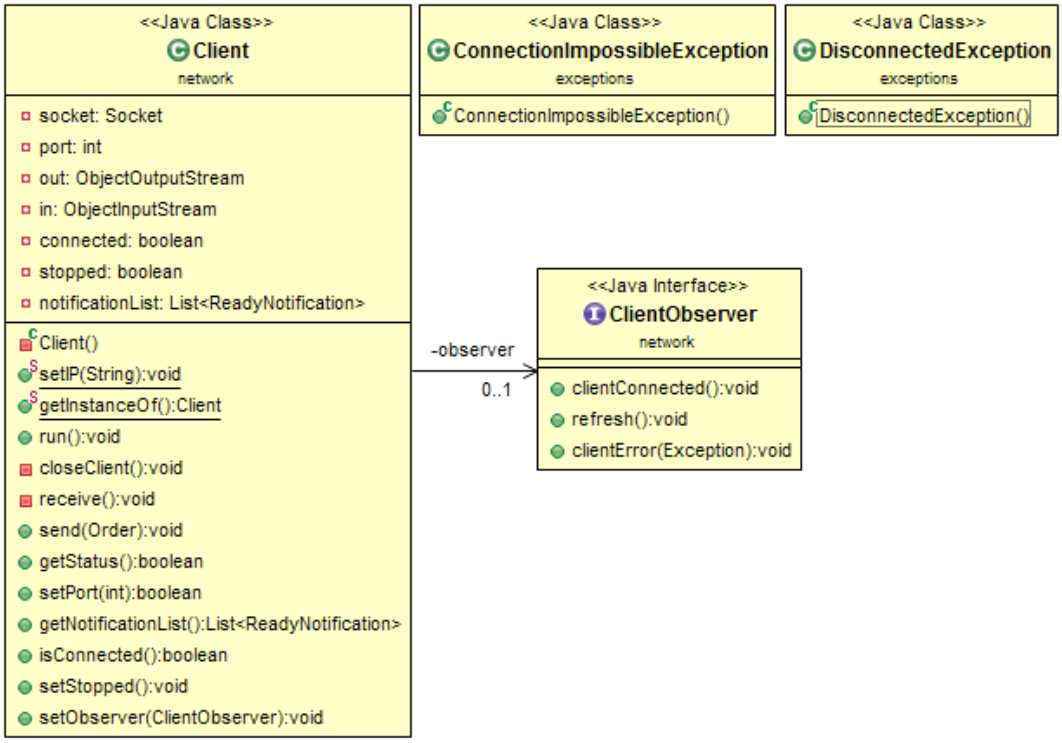
- Leggere il menu creato dalla *Cucina* e utilizzarlo per creare i propri ordini
- Creare gli ordini, assegnando ad ognuno di essi un numero di tavolo e aggiungendo le pietanze ordinate
- Inviare gli ordini alla *Cucina*, tramite protocollo TCP/IP
- Ricevere le notifiche per gli ordini dalla *Cucina*
- Visualizzare i dettagli di ogni ordine inviato
- Salvare periodicamente un backup dei dati della serata, per poterli eventualmente riaprire in caso di disconnessione involontaria

Si rende quindi necessari sviluppare due applicazioni con interfaccia grafica (GUI) che possono essere eseguite su due macchine diverse, collegate tra loro tramite rete LAN.
NOTA: per permettere il corretto funzionamento dell'applicazione, è necessario disabilitare il firewall (o comunque creare eccezioni adatte).

Progettazione architetturale

Entrambe le applicazioni sono basata sul pattern **Model-View-Controller**. Ciò permette di separare le parti che si occupano, rispettivamente, della conservazione dei dati, della visualizzazione degli stessi su schermo e della loro manipolazione, comandata tramite interfaccia grafica.
Per la gestione degli eventi generati dall'interazione dell'utente con le varie parti dell'interfaccia è stato utilizzato il pattern **Observer**.
Inoltre le classi *Client* e *Server* sono state realizzate secondo il pattern **Singleton**. Ciò permette di avere un'unica istanza di tale classe, evitando così di doverla passare da un controller all'altro.
Seguono alcuni schemi UML di esempio:


<<Java Interface>>
MainViewInterface
view.main
attachObserver(MainView Observer):void
refresh(List<Order>,List<Order>):void
setIP(String):void
getSelectedOrder():Order
enableConnect(boolean):void
<<Java Interface>>
IModel
model
addOrder(Order):void
setOrderReady(int):void
addDishMenu(AbstractDish):void
removeDishMenu(int):void
ordersReadyList():List<Order>
ordersNotReadyList():List<Order>
getMenu():Menu
setMenu(Menu):void
resetReady():void
getTotalPrice(Order):double
getNumberOfOrders():int
getOrderByID(int):Order
getOrderDishList(Order):List<CodifiedDish>
clone():IModel
-view
0..1
<<Java Class>>
Controller
controller
Controller(IModel)
openMenuCmd():void
connectCmd():void
refreshListCmd():void
detailOrderCmd():void
closeWindow Cmd():void
openHistoryCmd():void
serverConnected():void
refresh():void
serverError(Exception):void
setView (MainView Interface):void
-model
0..1
<<Java Class>>
MainFrame
view.main
MainFrame()
attachObserver(MainView Observer):void
getSelectedOrder():Order
refresh(List<Order>,List<Order>):void
setIP(String):void
enableConnect(boolean):void
w indow Activated(Window Event):void
w indow Closed(Window Event):void
w indow Closing(Window Event):void
w indow Deactivated(Window Event):void
w indow Deiconified(Window Event):void
w indow Iconified(Window Event):void
w indow Opened(Window Event):void
actionPerformed(ActionEvent):void
-observer
0..1
<<Java Interface>>
MainViewObserver
view.main
openMenuCmd():void
connectCmd():void
refreshListCmd():void
detailOrderCmd():void
closeWindow Cmd():void
openHistoryCmd():void
-oPanel
0..1
<<Java Class>>
OrderPanel
view.main
OrderPanel()
getRefresh():JButton
getDetails():JButton
updateNotReadyList(List<Order>):void
updateReadyList(List<Order>):void
getSelection():Order
<<Java Class>>
Selection
view.main
Selection(OrderList)
mouseClicked(MouseEvent):void



Organizzazione in package

Di seguito un analisi dell'organizzazione in package di entrambe le applicazioni:

Kitchen

- **controller:** contiene i sorgenti che incapsulano il comportamento dei vari controller relativi alle view. In particolare:
  - **controller:** contiene la classe ***Controller***, osservatore della view principale, la quale viene aggiornata con le liste degli ordini pronti e non. Lancia a richiesta la View principale per la modifica del Menù, oppure quella per visualizzare il testo con lo storico di tutti gli ordini effettuati ordinati per data.
  - **controller.menu:** contiene la classe ***MenuViewController***, osservatore della GUI che mostra l'elenco dei piatti in menù, permette di salvare o caricare su/da file il menù attualmente rappresentato, di rimuovere la pietanza selezionata dal menù, o di aprire la View per aggiungere nuovi piatti, controllata a sua volta dal ***NewDishController.*** Quest'ultimo consente di raccogliere le informazioni dalla sua GUI(che inserirà opportunamente l'utente) per creare e inserire nel menù nuovi piatti.
  - **controller.order:** contiene la classe ***DetailOrderController,*** che gestisce la View che mostra un elenco dei piatti ancora non pronti di uno specifico ordine. Consente di selezionare uno di questi e indicarlo come pronto. Una volta che tutte le pietanze sono pronte, manda una notifica automaticamente al cameriere connesso.
- **exceptions:** contiene alcune eccezioni appositamente create per la gestione di errori all'interno del programma. In particolare:
  - ***ConnectionImpossibleException*:** eccezione unchecked utilizzata quando il Client non riesce a connettersi al Server (per esempio perché l'indirizzo IP è sbagliato, o perché il firewall è attivo, o semplicemente perché il programma "Kitchen" non è ancora stato lanciato).

    - ***DisconnectedException*****:** eccezione uncheched utilizzata quando il Client per qualche motivo si disconnette dal server (per esempio perché il programma Cucina è stato chiuso).
    - ***TooMuchFoodException*****:** eccezione lanciata quando si tentano di aggiungere troppi piatti al menu.
- **historic:** contiene la classe che si occupa di salvare lo storico delle ordinazioni
    - ***DailyHistory***: thread che apre il file di testo indicato per salvare lo storico e vi appende la descrizione di un determinato ordine, correlato con la data corrente.
- **main:** contiene la classe ***TableServiceKitchen***, che si occupa dell'inizializzazione di alcuni dati e di lanciare l'applicazione vera e propria.
- **model:** contiene le classi che strutturano il corpo del programma
    - ***AbstractDish*****:** classe astratta che modella una pietanza con un determinato nome e prezzo
    - ***Beverage*****:** estende AbstractDish, rappresenta una bevanda
    - ***Food*****:** estende AbstractDish, rappresenta un cibo
    - ***CodifiedDish*****:** modella una coppia Intero-AbstractDish, utile a codificare ogni pietanza
    - ***Order*****:** modella un singolo ordine, distinguendo le pietanze pronte e quelle non, identificate dal loro codice all'interno di questa classe
    - ***Menu*****:** modella una lista di pietanze codificate che fanno parte del menù del ristorante, fornisce metodi per aggiungere e togliere pietanza, codificando univocamente ogni pietanza
    - ***Model*****:** è il modello dell'applicazione, tiene traccia degli ordini pronti e non, del menù al quale fare riferimento:
    - ***IModel*****:** è l'interfaccia implementata dal Model, descrive i metodi per apportare le opportune modifiche allo scheletro del programma.
    - ***ReadyNotification*****:** è la notifica che la cucina manda al cameriere quando un ordine è pronto, fornendogli l'id dell'ordine.
- **view:** contiene i sorgenti necessari alla creazione delle finestre dell'applicazione.
    - **view.main:** contiene le classi necessarie a costruire il frame principale dell'applicazione: ***DishTable*****:** JTable opportunamente modificate per contenere una lista di piatti (usata sia per la visualizzazione del menù che per la visualizzazione di un ordine), e per permettere l'aggiunta e la rimozione di singoli piatti. ***DishTableModel*****:** modello relativo alla tabella ***DishTable.*** ***OrderList*** invece estende una JList e viene utiizzata per mostrare a video una lista di ordini. ***MainFrame*** è la classe atta alla creazione della finestra principale, la quale mostrerà all'utente gli ordini pronti e non, permettendogli di passare alle view del menù o dello storico. Implementa l'interfaccia ***MainViewInterface***. Il pannello principale di questa interfaccia è definito da ***OrderPanel***, il quale contiene le due liste di ordini pronti e non, aggiornandosi opportunamente ad ogni modifica apportata al modello. ***HistoryFrame*** è un semplice Frame contenente un area di testo ove viene visualizzato il contenuto del file testuale contenente lo storico degli ordini
    - **view.menu:** contiene le classi necessarie a costruire la GUI che mostra il menù del ristorante e che può modificarlo. ***MainMenuFrame*** è la classe che crea il Frame principale per visualizzare il menù, implementa ***MenuViewInterface*** che fornisce i metodi per modificare il pannello principale, costituito dalla ***DishMenuList***, un JScrollPane contenente una tabella dei piatti. ***NewDishFrame*** è la classe che crea la

finestra per la creazione di nuovi piatti da inserire nel menu, ed implementa ***NewDishViewInterface.***

- **view.order:** contiene la classe ***DetailOrderFrame*** che si occupa di creare una finestra relativa ad uno specifico ordine, visualizzando, quando non è pronto, i piatti non pronti restanti, e, quando invece è pronto, tutto il dettaglio dell'ordine. Implementa l'interfaccia ***DetailOrderViewInterface*** che fornisce i metodi per aggiornare il pannello principale.

Waiter

- **controller:** contiene i sorgenti che incapsulano il comportamento dei vari controller relativi alle view. In particolare:
  - **controller.main:** contiene la classe ***Controller***, controller della view principale (*view.main*), in cui è possibile visualizzare l'elenco degli ordini effettuati e da cui si possono lanciare le due view per la creazione di un nuovo ordine e per la visualizzazione dei dettagli di un ordine già effettuato.
  - **controller.newOrder:** contiene la classe ***NewOrderPanelController***, controller della view per la creazione di un nuovo ordine (*view.newOrder*), in cui è possibile aggiungere varie pietanze, scelte dal menu, impostare un numero di tavolo e inviare l'ordine al cameriere.
- **exceptions:** contiene alcune eccezioni appositamente create per la gestione di errori all'iterno del programma. In particolare:
  - ***ConnectionImpossibleException*:** eccezione uncheched utilizzata quando il Client non riesce a connettersi al Server (per esempio perché l'indirizzo IP è sbagliato, o perché il firewall è attivo, o semplicemente perché il programma "Kitchen" non è ancora stato lanciato).
  - ***DiconnectedException*:** eccezione uncheched utilizzata quando il Client per qualche motivo si disconnette dal server (per esempio perché il programma Cucina è stato chiuso).
  - ***TooMuchFoodException*:** eccezione lanciata quando si tentano di aggiungere troppi piatti al menu.
- **main:** contiene la classe ***TableServiceWaiter***, che si occupa dell'inizializzazione di alcuni dati e di lanciare l'applicazione vera e propria.
- **model:** contiene le classi che strutturano il corpo del programma
  - ***AbstractDish*:** classe astratta che modella una pietanza con un determinato nome e prezzo
  - ***Beverage*:** estende AbstractDish, rappresenta una bevanda
  - ***Food*:** estende AbstractDish, rappresenta un cibo
  - ***CodifiedDish*:** modella una coppia Intero-AbstractDish, utile a codificare ogni pietanza
  - ***Order*:** modella un singolo ordine, distinguendo le pietanze pronte e quelle non, identificate dal loro codice all'interno di questa classe
  - ***Menu*:** modella una lista di pietanze codificate che fanno parte del menu del ristorante, fornisce metodi per aggiungere e togliere pietanza, codificando univocamente ogni pietanza
  - ***Model*:** è il modello dell'applicazione, tiene traccia degli ordini pronti e non, del menu al quale fare riferimento:

    - ***IModel*:** è l'interfaccia implementata dal Model, descrive i metodi per apportare le opportune modifiche allo scheletro del programma.
    - ***ReadyNotification*:** è la notifica che la cucina manda al cameriere quando un ordine è pronto, fornendogli l'id dell'ordine.
- **network:** contiene la classe ***Client***, ovvero il thread che si occupa della connessione di rete, della ricezione e dell'invio dei dati tra le due applicazioni, e l'observer ***ClientObserver***, utile per monitorare il client.
- **view:** contiene i sorgenti necessari alla creazione delle finestre dell'applicazione. In particolare le seguenti classi, posizionate qui perché comuni a più finestre:
    - ***DishTable*:** JTable opportunamente modificate per contenere una lista di piatti (usata sia per la visualizzazione del menu che per la visualizzazione di un ordine), e per permettere l'aggiunta e la rimozione di singoli piatti.
    - ***DishTableModel*:** modello relativo alla tabella ***DishTable***

  E i seguenti sotto-package:
    - **details:** classi relative alla finestra per la visualizzazione dei dettagli di un ordine. In particolare contiene le due classi ***DetailFrame*** e ***DetailPanel*** che modellano rispettivamente la frame e il pannello della suddetta finestra.
    - **main:** classi relative alla finestra principale. In particolare contiene un'interfaccia, ***MainPanelInterface***, implementata dal pannello ***MainPanel***, contenuto in ***MainFrame*** e l'interfaccia ***MainViewObserver***, ovvero l'observer della view principale. La classe ***OrderList*** modella una JList opportunamente modificata per contenere un elenco di ordini.
    - **newOrder:** classi relative alla finestra per la creazione di un nuovo ordine. In particolare contiene un'interfaccia, ***NewOrderPanelInterface***, implementata dal pannello ***NewOrderPanel***, contenuto in ***NewOrderFrame*** e l'interfaccia ***NewOrderPanelObserver***, ovvero l'observer della view per la creazione di un nuovo ordine.

Suddivisione del lavoro

La realizzazione del progetto è stata suddivisa tra i partecipanti nel seguente modo:

- **Marco Nobile:**
    - Interfaccia grafica lato *Cucina*
    - Model (comune a entrambe le applicazioni):
        - Model
        - Order
        - Dish/CodifiedDish
        - ...
- **Piero Biagini:**
    - Interfaccia grafica lato *Cameriere*
    - comunicazione tramite rete (thread Server e Client, impiegati rispettivamente nell'applicazione della *Cucina* e del *Cameriere*)
    - Testing (Model e classi correlate, Network e classi correlate)
    - Creazione delle JTable e del loro modello.

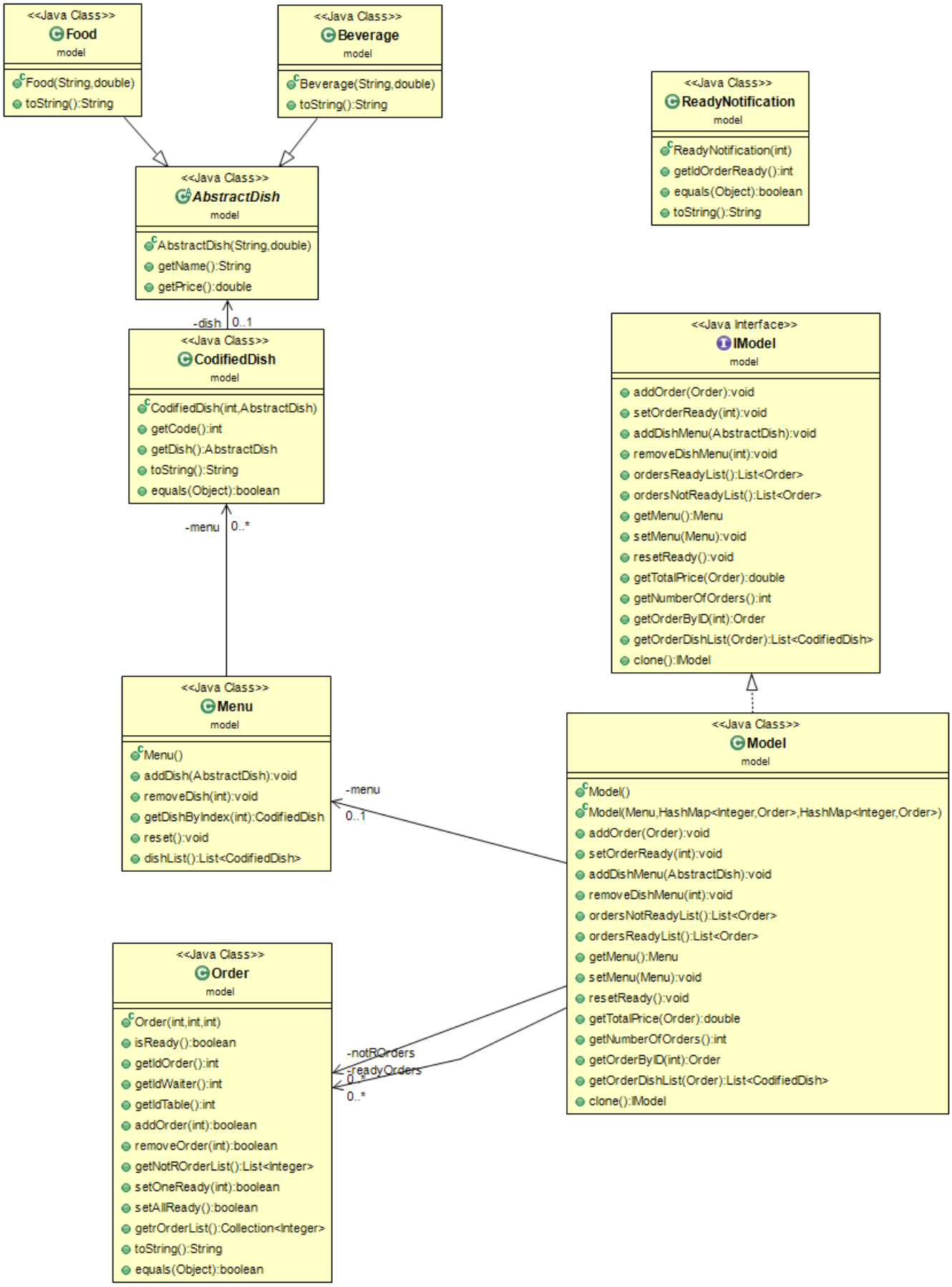
Progettazione di dettaglio: Parte di Marco Nobile

Model

Inizialmente è stato strutturato il modello dell'applicazione, ossia sono state definite le classi base per l'informatizzazione di una semplice attività ristorativa. I concetti base implementati sono stati: -*Pietanza*:generico oggetto che può essere sia cibo che bevanda
-*Ordine*: un insieme di pietanze inizialmente non pronte che devono essere preparate tutte prima che il cameriere possa portarle ai tavoli
-*Menù*:i nsieme di pietanze contraddistinte univocamente da un codice alle quali fare affidamento nella gestione delle comande.

Nello specifico le classi ***Food*** e ***Beverage*** estendono entrame la classe astratta ***AbstractDish***, caratterizzata dal nome e dal prezzo assegnati alla singola pietanza. Per associare un codice ad ogni pietanza è stato deciso di utilizzare semplicemente la classe ***CodifiedDish,*** per ogni pietanza utilizzata nel modello. La classe ***Order*** tiene traccia tramite due liste degli ordini pronti e non pronti. Il Cameriere invierà alla cucina un ordine con tutte le pietanze nella lista dei non pronti, sarà la cucina ad occuparsi, tramite i metodi della classe, a impostarli tutti pronti.
La classe ***Menu*** colleziona ogni pietanza disponibile, bevande e cibi. Il cameriere lo utilizza per scegliere quali piatti inserire in un ordine, la cucina ha la facoltà di modificarlo.
Il modello vero è proprio è implementato nella classe ***Model,*** la quale tiene in memoria sia gli ordini pronti che quelli non pronti, sia il menù attualmente utilizzato. È il medesimo sia per la cucina che per il cameriere.
Ogni qualvolta un ordine è completamente pronto(ovvero ogni pietanza è stata confermata pronta), la cucina manda una notifica al cameriere fornendogli l'ID dell'ordine pronto, questa funzione è implementata tramite la classe ***ReadyNotification***.

In seguito l'UML della struttura descritta.


<<Java Class>>
Food
model
Food(String,double)
toString():String
<<Java Class>>
Beverage
model
Beverage(String,double)
toString():String
<<Java Class>>
ReadyNotification
model
ReadyNotification(int)
getIdOrderReady():int
equals(Object):boolean
toString():String
<<Java Class>>
AbstractDish
model
AbstractDish(String,double)
getName():String
getPrice():double
-dish
0..1
<<Java Class>>
CodifiedDish
model
CodifiedDish(int,AbstractDish)
getCode():int
getDish():AbstractDish
toString():String
equals(Object):boolean
-menu
0..*
<<Java Interface>>
IModel
model
addOrder(Order):void
setOrderReady(int):void
addDishMenu(AbstractDish):void
removeDishMenu(int):void
ordersReadyList():List<Order>
ordersNotReadyList():List<Order>
getMenu():Menu
setMenu(Menu):void
resetReady():void
getTotalPrice(Order):double
getNumberOfOrders():int
getOrderByID(int):Order
getOrderDishList(Order):List<CodifiedDish>
clone():IModel
<<Java Class>>
Menu
model
Menu()
addDish(AbstractDish):void
removeDish(int):void
getDishByIndex(int):CodifiedDish
reset():void
dishList():List<CodifiedDish>
-menu
0..1
<<Java Class>>
Model
model
Model()
Model(Menu,HashMap<Integer,Order>,HashMap<Integer,Order>)
addOrder(Order):void
setOrderReady(int):void
addDishMenu(AbstractDish):void
removeDishMenu(int):void
ordersNotReadyList():List<Order>
ordersReadyList():List<Order>
getMenu():Menu
setMenu(Menu):void
resetReady():void
getTotalPrice(Order):double
getNumberOfOrders():int
getOrderByID(int):Order
getOrderDishList(Order):List<CodifiedDish>
clone():IModel
<<Java Class>>
Order
model
Order(int,int,int)
isReady():boolean
getIdOrder():int
getIdWaiter():int
getIdTable():int
addOrder(int):boolean
removeOrder(int):boolean
getNotROrderList():List<Integer>
setOneReady(int):boolean
setAllReady():boolean
getrOrderList():Collection<Integer>
toString():String
equals(Object):boolean
-notROrders
-readyOrders
0..*
0..*

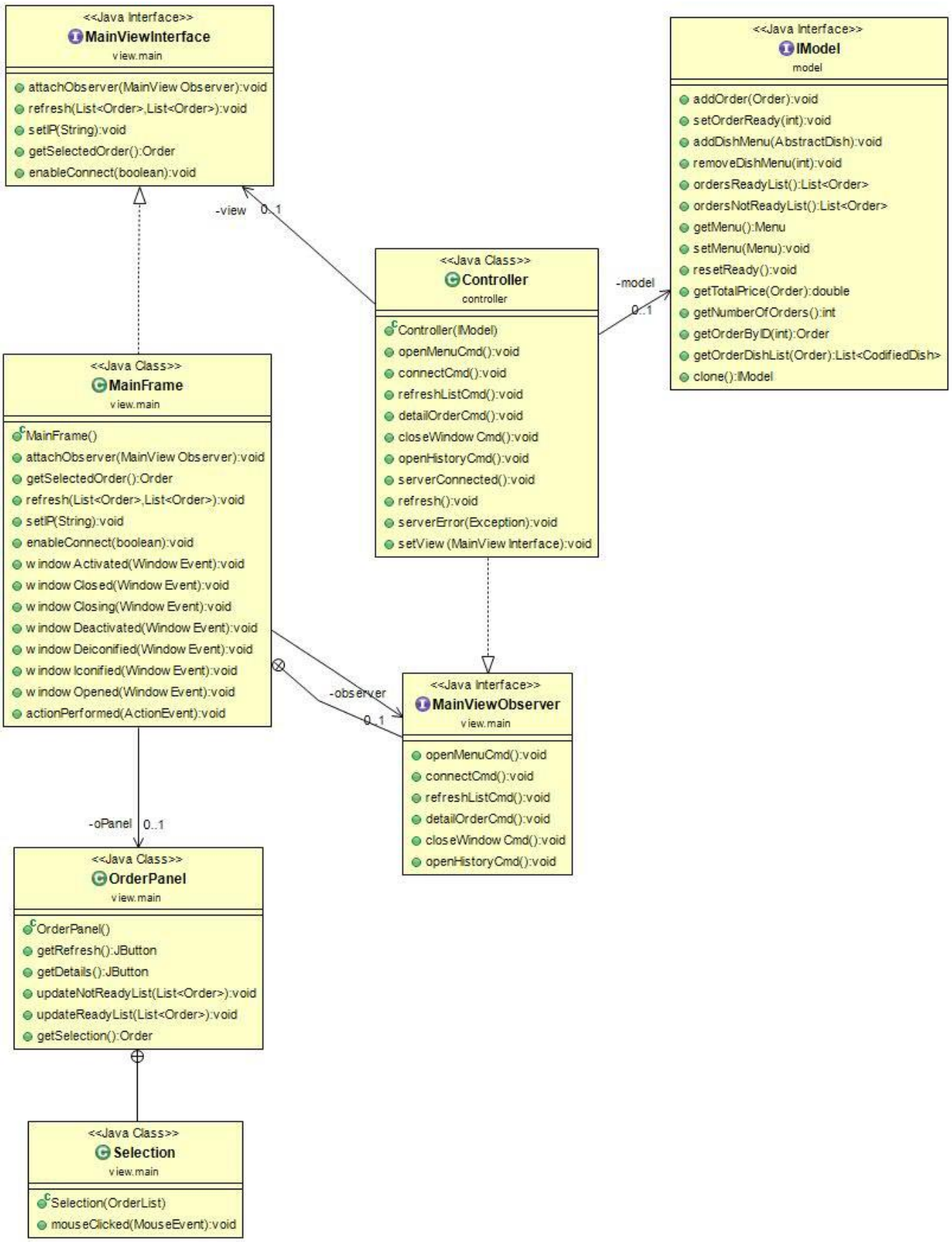
Interfaccia grafica Cucina

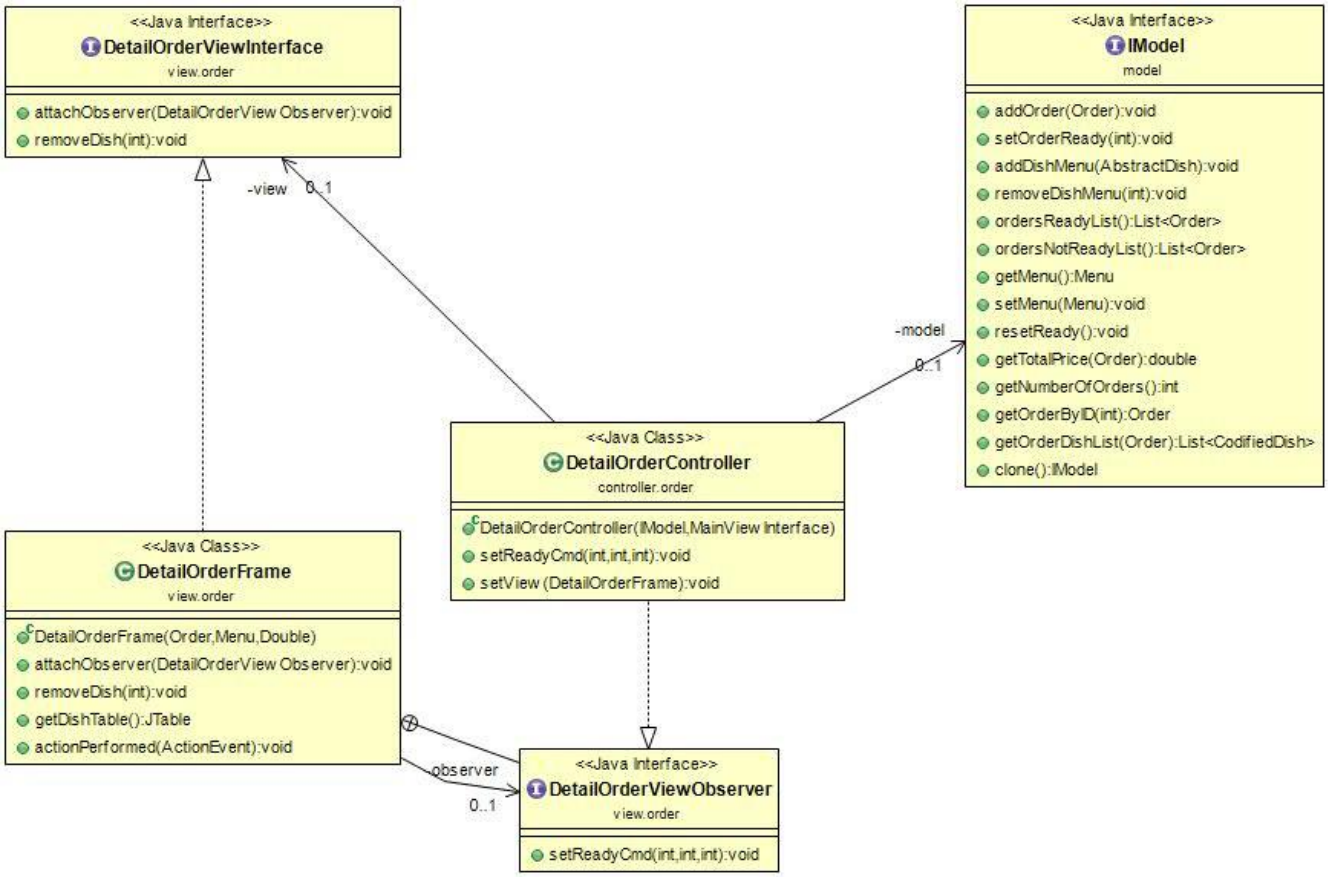
La GUI della cucina consta di 3 Frame fondamentali:

- *MainFrame*: finestra principale, in cui è possibile visualizzare gli ordini pronti e gli ordini non pronti
- *DetailOrderFrame:* finestra di riepilogo di un ordine. Viene utilizzata per selezionare i piatti da confermare come pronti. in caso di ordine già pronto, visualizza semplicemente l'elenco completo della comanda.
- *MainMenuFrame:* finestra in cui è possibile visualizzare l'elenco di tutte le pietanze in menu, rimuoverle o aggiungerne di nuove.

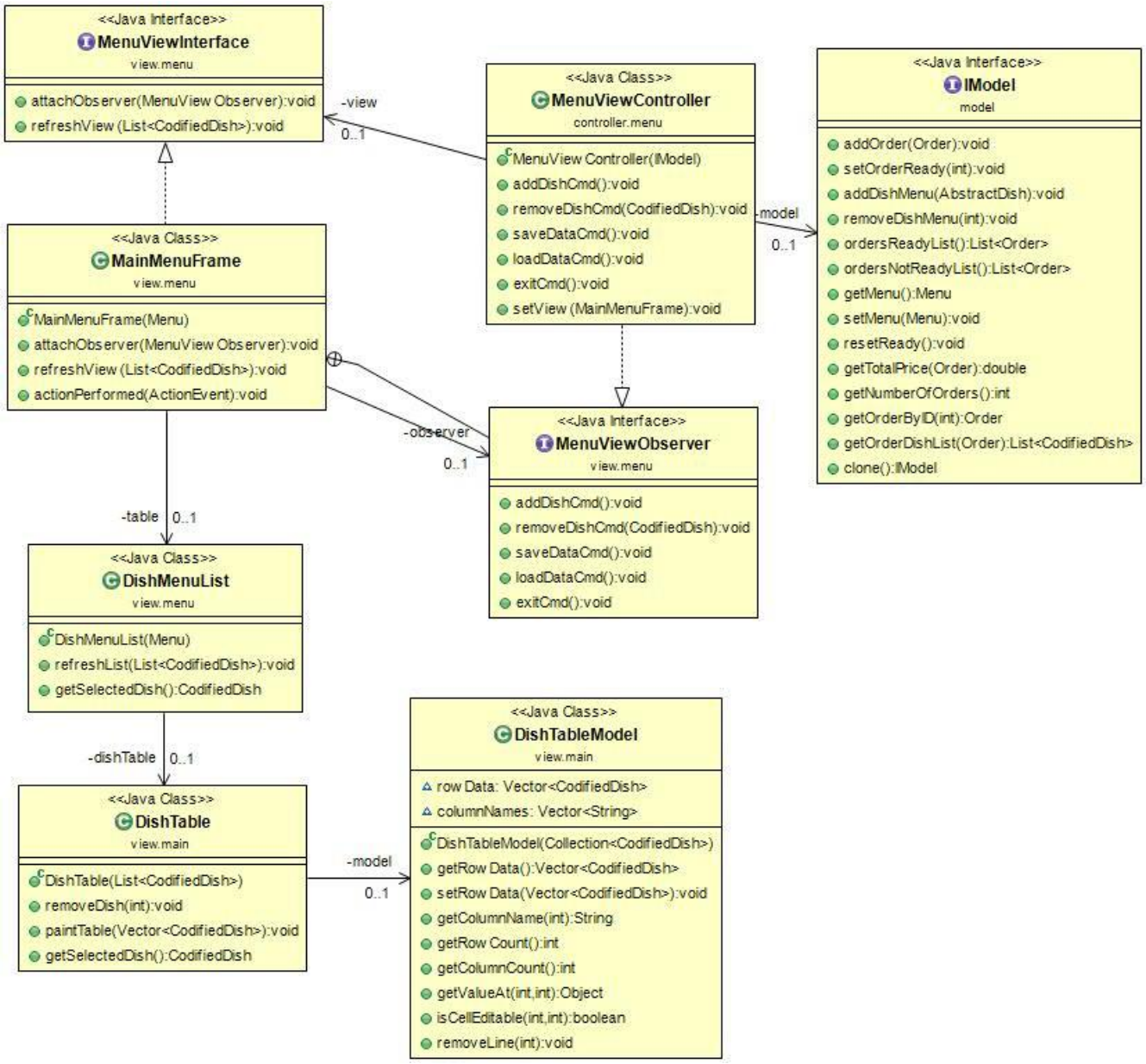
Tutte e tre le finestre sono stato realizzate implementando il pattern *Model-View-Controller* e *Observer*. Ad ogni frame è quindi associato un proprio controller, che potrà operare su di essa prelevando i dati dal model. Il controller è inoltre *Observer* della finestra, in quanto estende le relative interfacce realizzate all'interno della frame o del panel stesso.

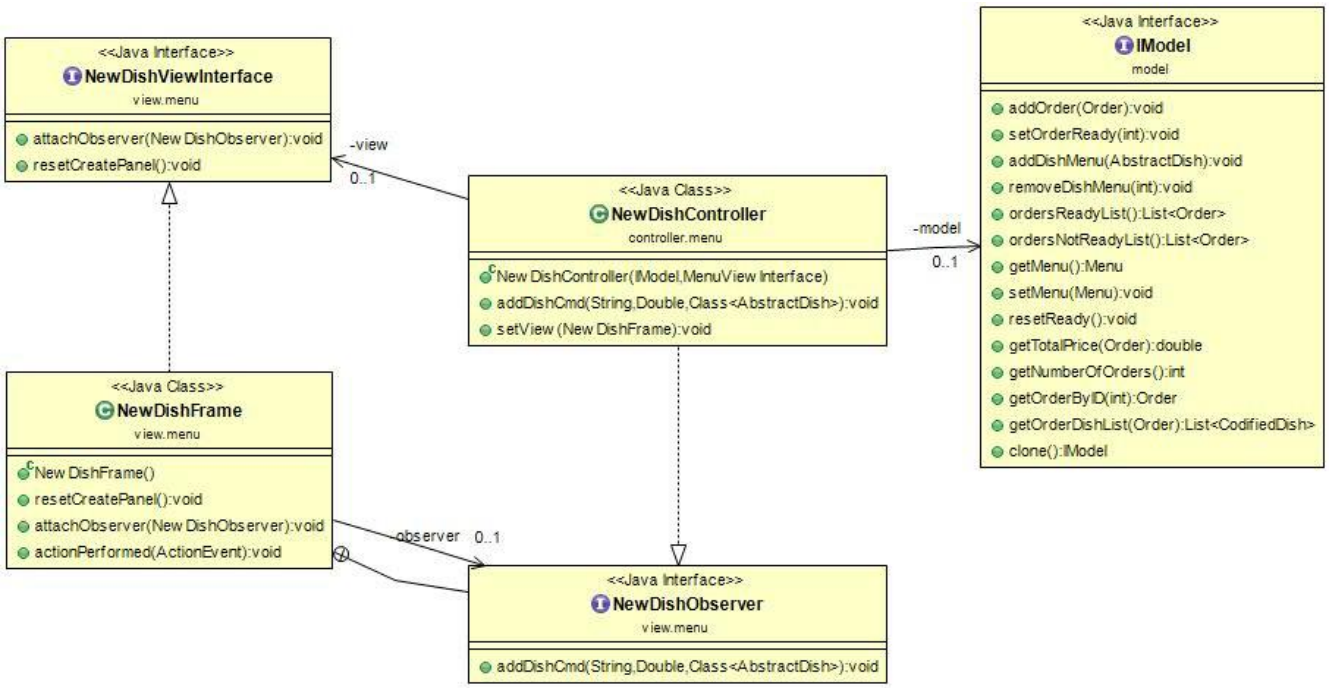
Di seguito l'UML delle prime due finestre, con relativi observer e controller:


<<Java Interface>>
MainViewInterface
view.main
attachObserver(MainView Observer):void
refresh(List<Order>,List<Order>):void
setIP(String):void
getSelectedOrder():Order
enableConnect(boolean):void
<<Java Interface>>
IModel
model
addOrder(Order):void
setOrderReady(int):void
addDishMenu(AbstractDish):void
removeDishMenu(int):void
ordersReadyList():List<Order>
ordersNotReadyList():List<Order>
getMenu():Menu
setMenu(Menu):void
resetReady():void
getTotalPrice(Order):double
getNumberOfOrders():int
getOrderByID(int):Order
getOrderDishList(Order):List<CodifiedDish>
clone():IModel
-view
0..1
<<Java Class>>
Controller
controller
Controller(IModel)
openMenuCmd():void
connectCmd():void
refreshListCmd():void
detailOrderCmd():void
closeWindow Cmd():void
openHistoryCmd():void
serverConnected():void
refresh():void
serverError(Exception):void
setView (MainView Interface):void
-model
0..1
<<Java Class>>
MainFrame
view.main
MainFrame()
attachObserver(MainView Observer):void
getSelectedOrder():Order
refresh(List<Order>,List<Order>):void
setIP(String):void
enableConnect(boolean):void
w indow Activated(Window Event):void
w indow Closed(Window Event):void
w indow Closing(Window Event):void
w indow Deactivated(Window Event):void
w indow Deiconified(Window Event):void
w indow Iconified(Window Event):void
w indow Opened(Window Event):void
actionPerformed(ActionEvent):void
-observer
0..1
<<Java Interface>>
MainViewObserver
view.main
openMenuCmd():void
connectCmd():void
refreshListCmd():void
detailOrderCmd():void
closeWindow Cmd():void
openHistoryCmd():void
-oPanel
0..1
<<Java Class>>
OrderPanel
view.main
OrderPanel()
getRefresh():JButton
getDetails():JButton
updateNotReadyList(List<Order>):void
updateReadyList(List<Order>):void
getSelection():Order
<<Java Class>>
Selection
view.main
Selection(OrderList)
mouseClicked(MouseEvent):void


<<Java Interface>>
DetailOrderViewInterface
view.order
attachObserver(DetailOrderView Observer):void
removeDish(int):void
-view
0..1
<<Java Interface>>
IModel
model
addOrder(Order):void
setOrderReady(int):void
addDishMenu(AbstractDish):void
removeDishMenu(int):void
ordersReadyList():List<Order>
ordersNotReadyList():List<Order>
getMenu():Menu
setMenu(Menu):void
resetReady():void
getTotalPrice(Order):double
getNumberOfOrders():int
getOrderByID(int):Order
getOrderDishList(Order):List<CodifiedDish>
clone():IModel
-model
0..1
<<Java Class>>
DetailOrderController
controller.order
DetailOrderController(IModel,MainView Interface)
setReadyCmd(int,int,int):void
setView (DetailOrderFrame):void
<<Java Class>>
DetailOrderFrame
view.order
DetailOrderFrame(Order,Menu,Double)
attachObserver(DetailOrderView Observer):void
removeDish(int):void
getDishTable():JTable
actionPerformed(ActionEvent):void
-observer
0..1
<<Java Interface>>
DetailOrderViewObserver
view.order
setReadyCmd(int,int,int):void

Nell'interfaccia del Menu, ogni qualvolta si desidera aggiungere un nuovo piatto in lista, si apre un frame per la scelta dei parametri da dare alla nuova pietanza: il *NewDishFrame,* anch'esso dotato di un suo controller. Di seguito l'UML completo:





In ultimo vi è la possibilità di mostrare a video il contenuto di un file testuale che tiene traccia dello storico degli ordini catalogandoli uno dopo l'altro specificando per ognuno la data, si chiama *HistoryFrame.* Quest'ultimo non ha bisogno di un osservatore in quanto funge semplicemente da area di testo da visualizzare.

Progettazione di dettaglio: Parte di Piero Biagini

Network

La prima parte ad essere stata realizzata è la parte legata alla comunicazione delle due applicazioni tramite reti.

Entrambe le classi, ***Client*** e ***Server***, ovvero le uniche classi che si occupano di questo compito, sono due thread, realizzati secondo il pattern *singleton* e il pattern *observer*. L'uso del pattern *singleton* ha permesso di evitare di dover passare l'istanza del thread da un controller all'altro, rendendo il codice più snello e leggibile. Essi vengono inizializzati e creati all'inizio del programma, e rimangono in esecuzione fino alla sua chiusura, a meno di eventuali errori, notificati tramite il relativo Observer (**ClientObserver** e **ServerObserver**) e due eccezioni create ad-hoc: ***ConnectionImpossibleException*** e ***DisconnectedException***.

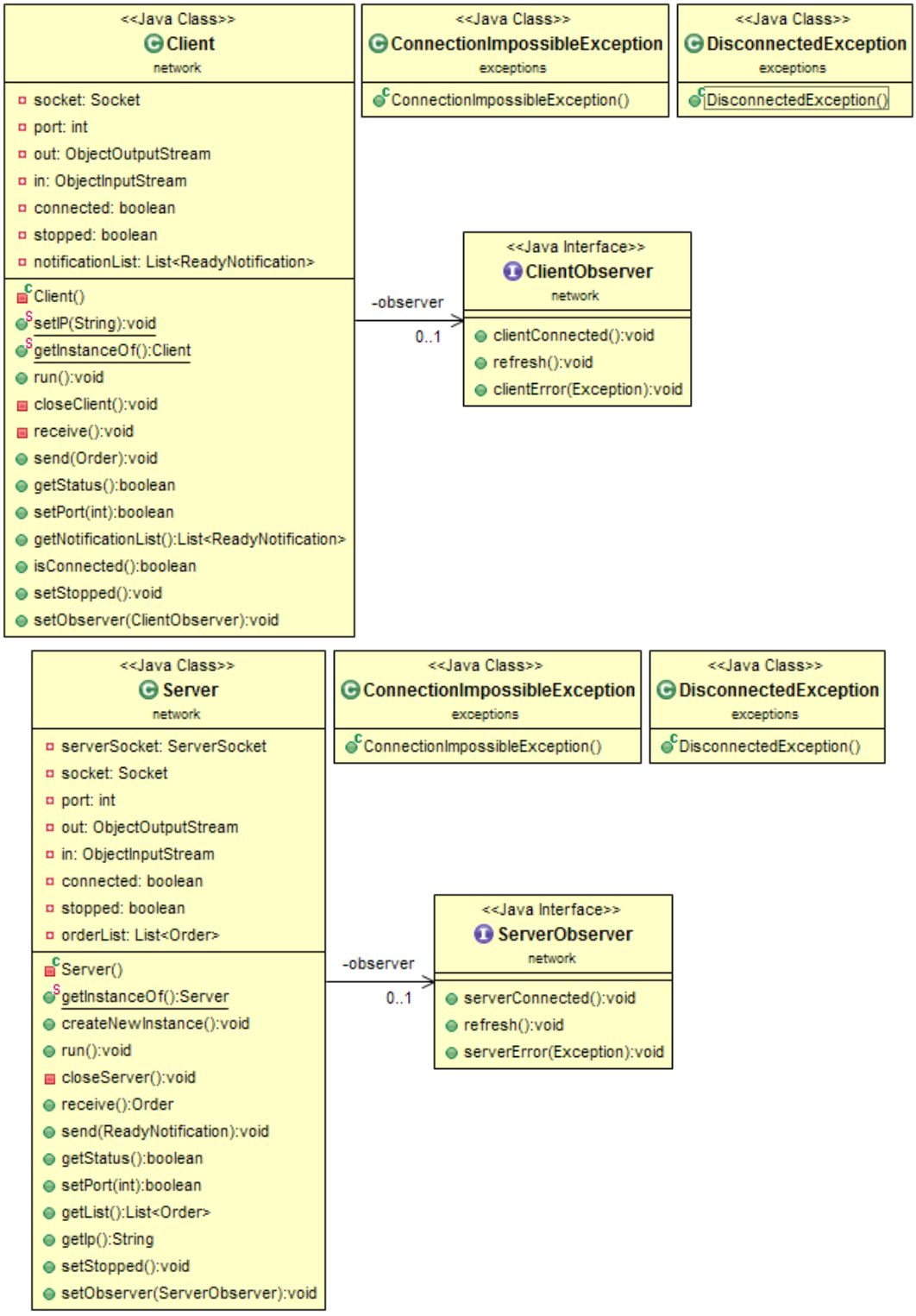
Il comportamento in caso di errori è differente, a seconda che si tratti del programma *Kitchen* o *Waiter*:

- *Kitchen:*
  Se per qualche motivo il thread dovesse lanciare un'eccezione terminare, è possibile ottenere una nuova istanza del *Server* tramite l'apposito metodo (in sostanza, è stata leggermente alterata la vera natura del pattern Singleton per garantire maggior flessibilità). In questo modo è possibile effettuare una riconnessione tramite l'apposito bottone *File>Connetti*.
- *Waiter:*
  Il *Client* è realizzato con un pattern Singleton puro, quindi in caso di terminazione in seguito ad un errore è necessario riavviare il programma (come specificato dalla finestra che compare per notificare l'evento all'utente). In ogni caso alla riapertura del programma verrà chiesto di

ricaricare un backup che viene salvato ogni volta che vengono inviati o creati nuovi ordini, quindi costantemente aggiornato.

L'**Observer** permette inoltre di effettuare alcune operazioni nel momento in cui il *Client* o il *Server* eseguono operazioni di lettura o scrittura e nel momento in cui si connettono l'uno all'altro.

Di seguito l'UML delle classi appena descritte:



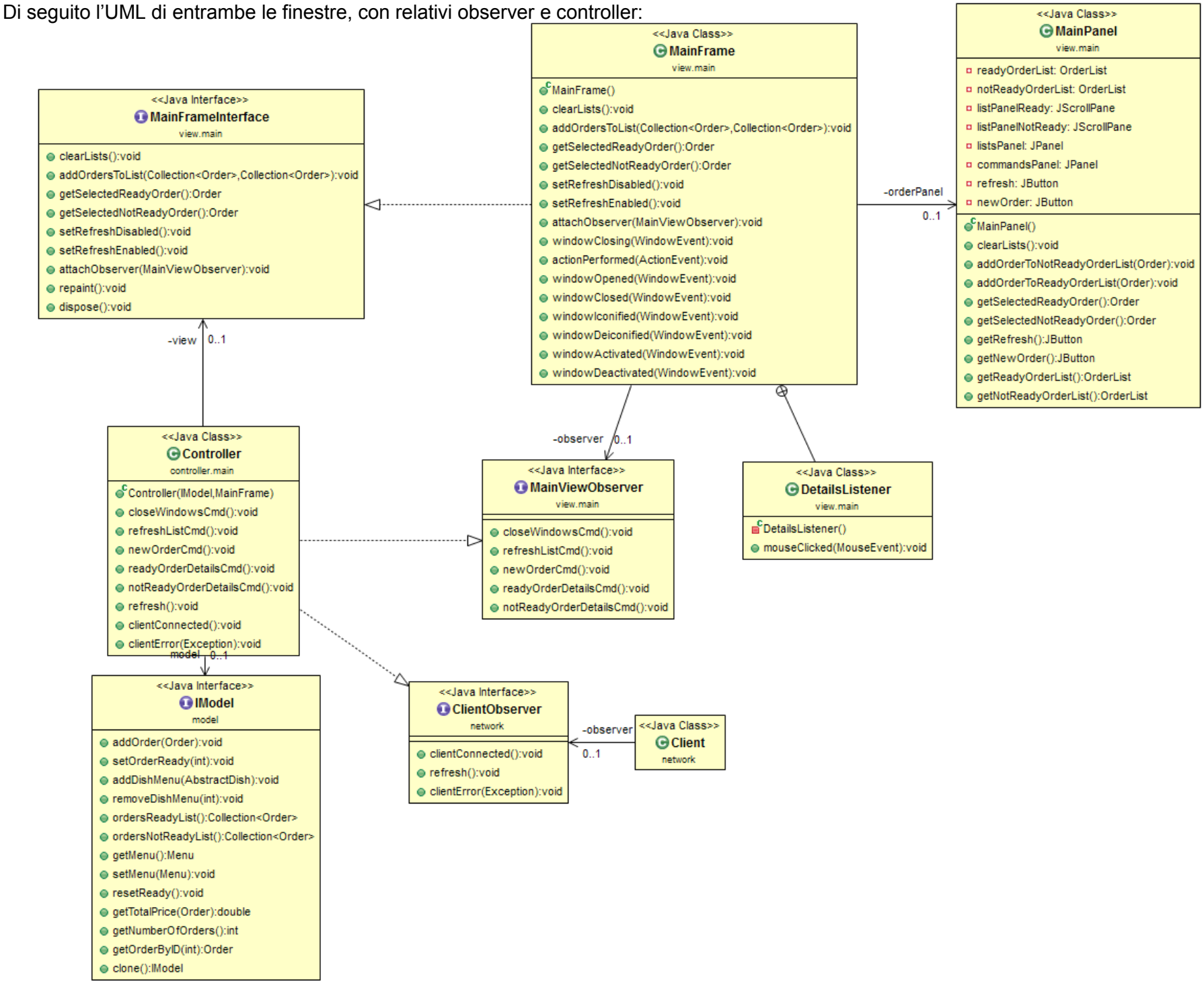
Interfaccia grafica

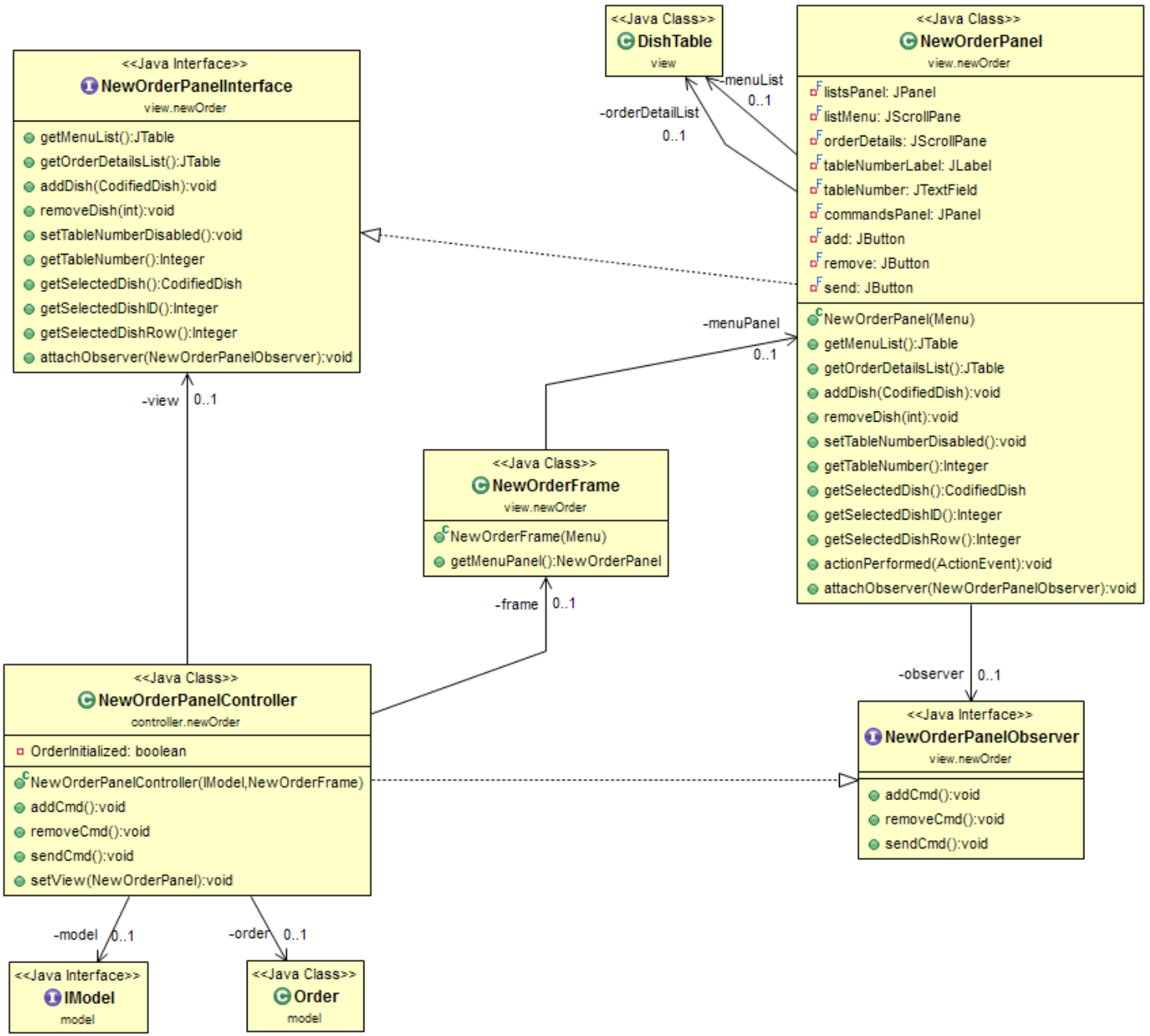
Il programma consta di 3 finestre:

- *MainFrame*: finestra principale, in cui è possibile visualizzare gli ordini pronti e gli ordini non pronti
- *NewOrderFrame:* finestra in cui è possibile creare ed inviare un nuovo ordine. Può essere aperta cliccando sul bottone “Nuovo Ordine” della *MainFrame*
- *DetailsFrame:* finestra di riepilogo di un ordine. Può essere aperta cliccando su un qualunque ordine, pronto o meno, visualizzato nella *MainFrame*.

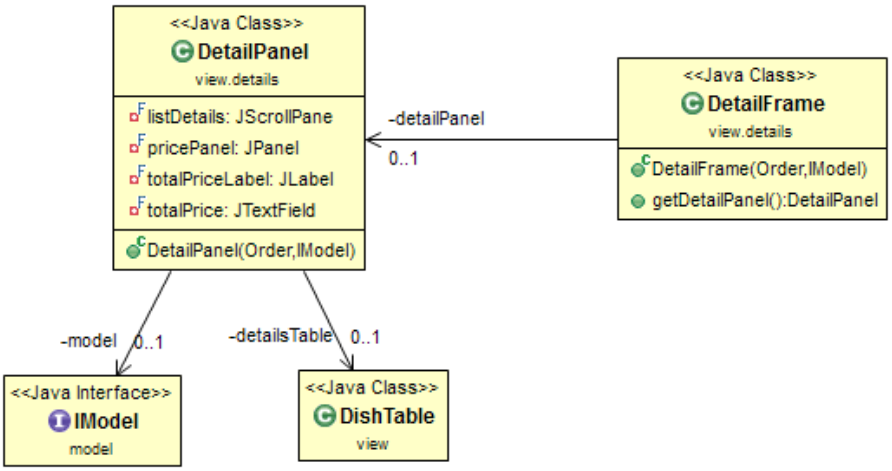
Le prime due finestre sono realizzate utilizzando i due pattern *Model-View-Controller* e *Observer*. Ad ogni frame è quindi associato un proprio controller, che potrà operare su di essa prelevando i dati dal model. Il controller è inoltre *Observer* della finestra, in quanto estende le relative interfacce realizzate all’interno della frame o del panel stesso.

Di seguito l'UML di entrambe le finestre, con relativi observer e controller:




<<Java Interface>>
NewOrderPanelInterface
view.newOrder
getMenuList():JTable
getOrderDetailsList():JTable
addDish(CodifiedDish):void
removeDish(int):void
setTableNumberDisabled():void
getTableNumber():Integer
getSelectedDish():CodifiedDish
getSelectedDishID():Integer
getSelectedDishRow():Integer
attachObserver(NewOrderPanelObserver):void
<<Java Class>>
DishTable
view
-menuList
0..1
-orderDetailList
0..1
<<Java Class>>
NewOrderPanel
view.newOrder
listsPanel: JPanel
listMenu: JScrollPane
orderDetails: JScrollPane
tableNumberLabel: JLabel
tableNumber: JTextField
commandsPanel: JPanel
add: JButton
remove: JButton
send: JButton
NewOrderPanel(Menu)
getMenuList():JTable
getOrderDetailsList():JTable
addDish(CodifiedDish):void
removeDish(int):void
setTableNumberDisabled():void
getTableNumber():Integer
getSelectedDish():CodifiedDish
getSelectedDishID():Integer
getSelectedDishRow():Integer
actionPerformed(ActionEvent):void
attachObserver(NewOrderPanelObserver):void
-menuPanel
0..1
-view
0..1
<<Java Class>>
NewOrderFrame
view.newOrder
NewOrderFrame(Menu)
getMenuPanel():NewOrderPanel
-frame
0..1
-observer
0..1
<<Java Class>>
NewOrderPanelController
controller.newOrder
OrderInitialized: boolean
NewOrderPanelController(IModel,NewOrderFrame)
addCmd():void
removeCmd():void
sendCmd():void
setView(NewOrderPanel):void
<<Java Interface>>
NewOrderPanelObserver
view.newOrder
addCmd():void
removeCmd():void
sendCmd():void
-model
0..1
-order
0..1
<<Java Interface>>
IModel
model
<<Java Class>>
Order
model

Per la terza finestra è stato adottato, invece, un metodo più semplificato, in quanto essa non deve operare su dati, ma semplicemente li deve visualizzare. L'ordine che dovrà mostrare le viene semplicemente passato dal controller della finestra principale, in seguito verrà trasformato in lista e aggiunto alla tabella.

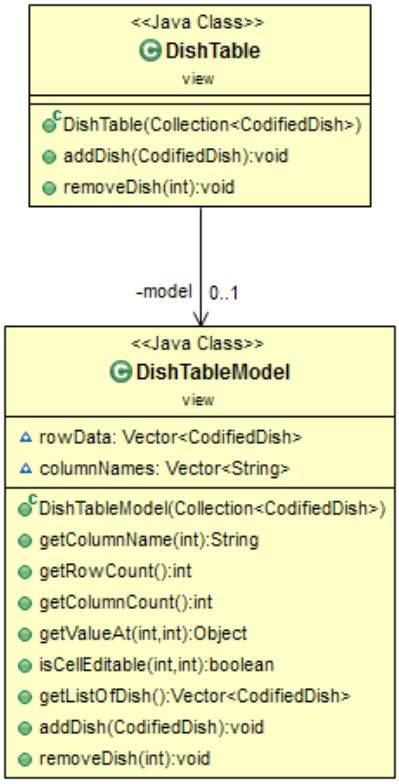


DishTable

Per la visualizzazione dei dettagli di un ordine oppure del menu (in generale, liste di ordini) è stata utilizzata una JTable modificata.

Come si può notare dall'UML presente qui sotto, sono stati aggiunti oltre ai metodi principali due metodi per permettere l'aggiunta o la rimozione di piatti.

Chiaramente oltre alla JTable è stato modificato anche il suo modello, creandone uno ad-hoc, sia per permettere l'aggiunta/rimozione di elementi che per impedire la possibilità di modificare gli elementi della tabella.



Test

I test sono stati realizzati tramite la libreria JUnit e sono descritti in maniera dettagliata nella sezione della relazione relativa al Testing.

Testing

Sono state testate le classi principali del progetto tramite JUnit. In particolare:

- **TableServiceWaiter.test.model:**
  - **TestMenu:** si occupa di testare le funzionalità della classe “Menu”:
    - Aggiunta di nuovi piatti (e relativa gestione delle eccezioni in caso vengano aggiunti troppi piatti)
    - Rimozione di piatti (e relativa gestione delle eccezioni in caso il piatto non sia presente)
    - Reset del menu
  - **TestOrder:** si occupa di testare le funzinalità della classe “Order”
    - Aggiunta di nuovi piatti all’ordine
    - Verifica dello stato dell’ordine (“Ready” o “Not Ready”)
    - Impostazione di un piatto come “Ready”
    - Impostazione di tutti i piatti come “Ready”
  - **TestModel:** si occupa di testare le funzionalità della classe “Model”:
    - Aggiunta di un menu (classe già testata in precedenza)
    - Aggiunta di nuovi ordini (classe già testata in precedenza)
    - Impostazione di un ordine come “Ready”
    - Recupero di un ordine tramite ID
    - Calcolo del costo totale di un ordine
- **TableServiceWaiter.test.network** e **TableServiceKitchen.test.network:**
  - **TestClientConnection** (Waiter) e **TestServerConnection** (Kitchen): le due classi stabiliscono una connessione di rete e si scambiano alcuni oggetti, verificando che siano uguali. In particolare il Client invia degli ordini, mentre il Server delle ReadyNotification.
  - **TestClientExceptions:** viene verificato che, a seguito dell’utilizzo di un indirizzo IP non corretto (quindi alla mancanza di un server a cui connettersi) il Client generi l’eccezione adatta.
  - **TestServerExceptions:** viene verificato che, a seguito di una mancata connessione al Client, il Server generi un’eccezione adatta.

Oltre ai test tramite JUnit sono stati eseguiti alcuni test manuali, in modo da verificare il corretto comportamento dell’interfaccia del programma. Questi test sono stati svolti personalmente dai partecipanti al progetto testando ognuno la parte dell’altro, in modo da evidenziare potenziali difetti non previsti da chi aveva scritto il codice.
Le funzionalità testate sono, chiaramente, tutte le funzionalità richieste (descritte nell’introduzione del programma).

Note finali

Il flusso di lavoro è stato abbastanza lineare: l'idea di cosa sviluppare era piuttosto chiara sin dall'inizio. Sono stati tuttavia aggiunti numerosi dettagli, la cui importanza è stata scoperta durante la realizzazione del programma stesso.

I cambiamenti più importanti non sono stati fatti riguardo al contenuto del programma, quando alla scelta di come scriverlo, e hanno portato più volte ad una rifattorizzazione del codice.

Le parti sono state sviluppate quasi completamente in autoonmia, a parte alcuni momenti di check in cui è stata richiesta l'aggiunta o la modifica di funzionalità particolari da parte di uno o dell'altro componente del gruppo. In ogni caso la collaborazione è stata piuttosto tranquilla e produttiva.

Le criticità più importanti sono state rilevate nella parte di comunicazione di rete che, sebbene funzionasse a grandi linee (gli oggetti venivano effettivamente scambiati), presentava bug più o meno importanti dovuti alla difficoltà iniziale di catturare ed interpretare le eccezioni provenienti dal thread.