“DINER DASH”
progetto per il corso di
“Programmazione ad Oggetti”

Babini Pier Costante,
Giorgi Marco,
Donati Matteo,
Benzi Federico

5 giugno 2023

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il gruppo ha come obiettivo quello di realizzare un gioco molto simile all'iconico Diner Dash. Ne esistono varie versioni, il progetto si concentra sulla prima, quella del 2003. Il gioco consiste nel controllare i movimenti e azioni di Flo, una cameriera intenta nel gestire un ristorante. Durante la partita diversi clienti entreranno nel ristorante e siederanno ai tavoli. Il compito di Flo è quello di raccogliere gli ordini, servire le pietanze e raccogliere gli introiti, tutto questo cercando di non far aspettare troppo i clienti perchè potrebbero spazientirsi ed andarsene.

Requisiti funzionali

- Logica di Clienti, Cameriera, Chef, Bancone e Tavoli.
- Grafica in generale.
- Gestione click del mouse.
- Gestione evento Game Over (scadenza tempo rimanete, raggiunto numero massimo di clienti andati via).
- Gestione stato clienti e relativa rappresentazione grafica (richiesta ordini, arrabbiati, e di pagamento).
- Utilizzo di power-up per alterare le dinamiche di gioco.

Requisiti non funzionali

- Grafica di gioco dinamica in base alla dimensione della finestra.

- Menù di pausa.
- Ordini portati manualmente allo Chef dalla cameriera.
- Clienti ordinano e finiscono di mangiare dopo un lasso di tempo casuale.
- Possibilità di far ricominciare il gioco tramite Restart.

1.2 Analisi e modello del dominio

Il gioco permetterà di controllare la figura di una cameriera all'interno di un ristorante. Alla cameriera sarà possibile indicare dove andare tramite il click del mouse. Durante la partita, ad intervalli di tempo regolari, si presenteranno dei clienti al ristorante. In base alla disponibilità dei tavoli i clienti potranno comportarsi in due modi:

- caso in cui ci sia almeno 1 tavolo libero: il cliente, o gruppo di clienti, si dirigerà verso il tavolo e si siederà, iniziando a pensare all'ordine.
- caso in cui nessun tavolo sia disponibile: il cliente/i si metteranno in fila attendendo che almeno 1 tavolo si liberi. Dopo un certo lasso di tempo se nessun tavolo è disponibile, il cliente si arrabbierà e lascerà il ristorante.

Una volta seduto al tavolo, il cliente inizierà a pensare a cosa ordinare. Da questo momento in poi, il cliente non lascerà il ristorante a meno che il suo ordine non sia completato. Quando avrà deciso mostrerà uno stato che permetterà al giocatore di accorgersi che il cliente è pronto e di conseguenza sarà possibile raccogliere il suo ordine. La camerierà, una volta raggiunto il tavolo, raccoglierà l'ordine del cliente. L'ordine sarà immediatamente disponibile allo chef per iniziarne la preparazione. Quando un ordine sarà pronto verrà aggiunto al bancone. Un piatto sul bancone rappresenta le pietanze per tutti i clienti di un tavolo ed il tavolo sarà identificabile grazie al numero del piatto. La cameriera, potendo trasportare un massimo di 2 piatti alla volta, servirà poi l'ordine al tavolo corretto. Quando i clienti avranno terminato il pasto, saranno pronti per pagare. In questa fase mostreranno nuovamente uno stato che permetterà al giocatore di inviare la cameriera per gestire il pagamento. Una volta terminato il pagamento, l'ordine si concluderà, il tavolo verrà liberato ed il primo cliente in fila potrà occuparlo. Il gioco prevederà anche dei power-up che consentiranno di alterarne il comportamento durante lo svolgimento. La difficoltà primaria consiste nel gestire parallelamente tutte le entità ed i loro diversi comportamenti.

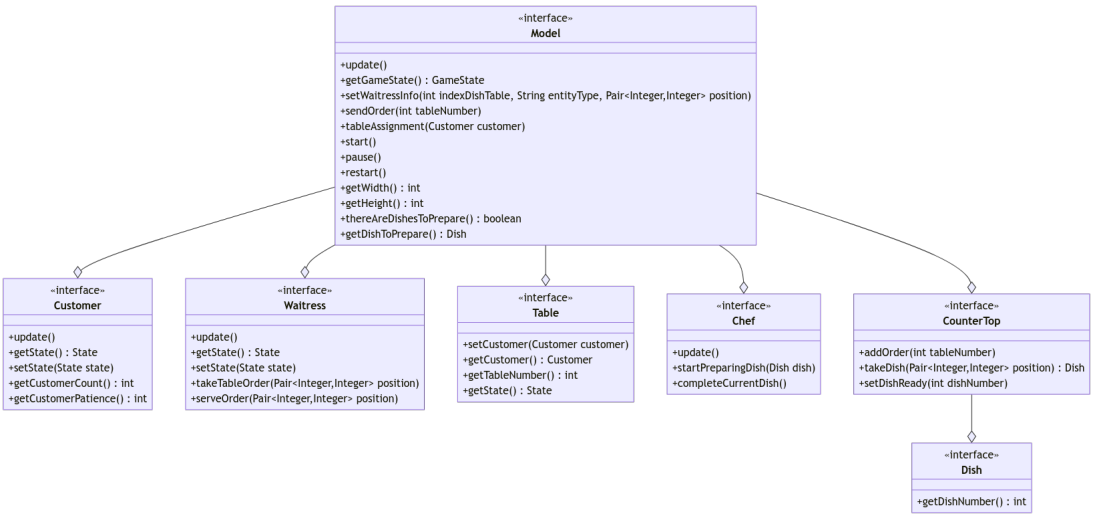


Figura 1.1: Schema UML dell'analisi del problema.

Capitolo 2

Design

2.1 Architettura

In seguito all'analisi è stato scelto l'uso del pattern architetturale MVC, che consente agli elementi che lo costituiscono di avere una certa modularità. Per gestire l'aggiornamento dello stato di ogni entità viene chiamata un sequenza di loro metodi nel Model, in questo modo si evitano problemi di sincronizzazione in caso di accesso contemporaneo a strutture condivise simulando al contempo un'esecuzione parallela.
L'architettura è quindi composta da:

- Model: contiene tutte le entità di gioco ed eventuali strutture necessarie per lo svolgimento del gioco.
- Controller: è il vero e proprio core del sistema. Raccoglie gli input della view e richiama i corretti cambiamenti sul Model. Permette la sincronizzazione fra Model e View.
- Game Loop: gestisce il loop di gioco richiamando ciclicamente il Controller, il quale si occupa di aggiornare correttamente lo stato di gioco.
- View: si occupa di mostrare graficamente lo stato di gioco. E' organizzata modularmente, in modo che ogni elemento che la compone sia visibile al momento opportuno durante il gioco.

L'architettura consente quindi di utilizzare una View qualsiasi, ad esempio GUI o CLI, senza impattare sul resto dei componenti. Durante la partita, come brevemente accennato, il Game Loop richiama il Controller. Quest'ultimo si occupa di richiamare l'aggiornamento allo stato del Model in base ai dati attuali delle entità che lo compongono e di aggiornare la View. Gli unici input del programma sono click nella View e vengono gestiti dal Controller.

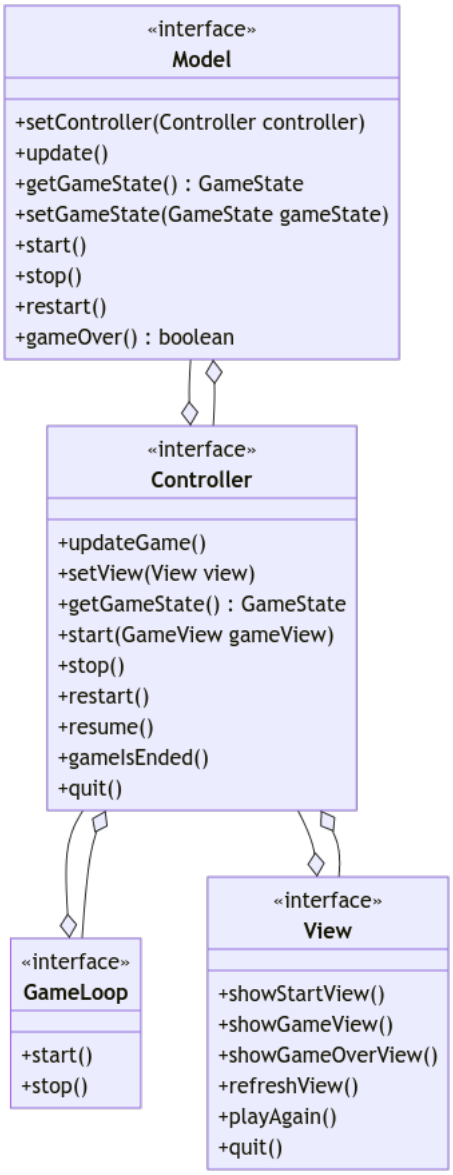
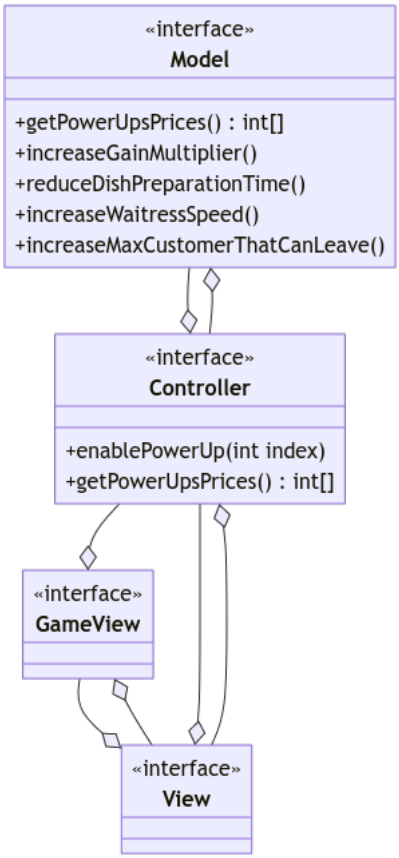


Figura 2.1: Schema UML rappresentante l'architettura.

2.2 Design dettagliato

Matteo Donati

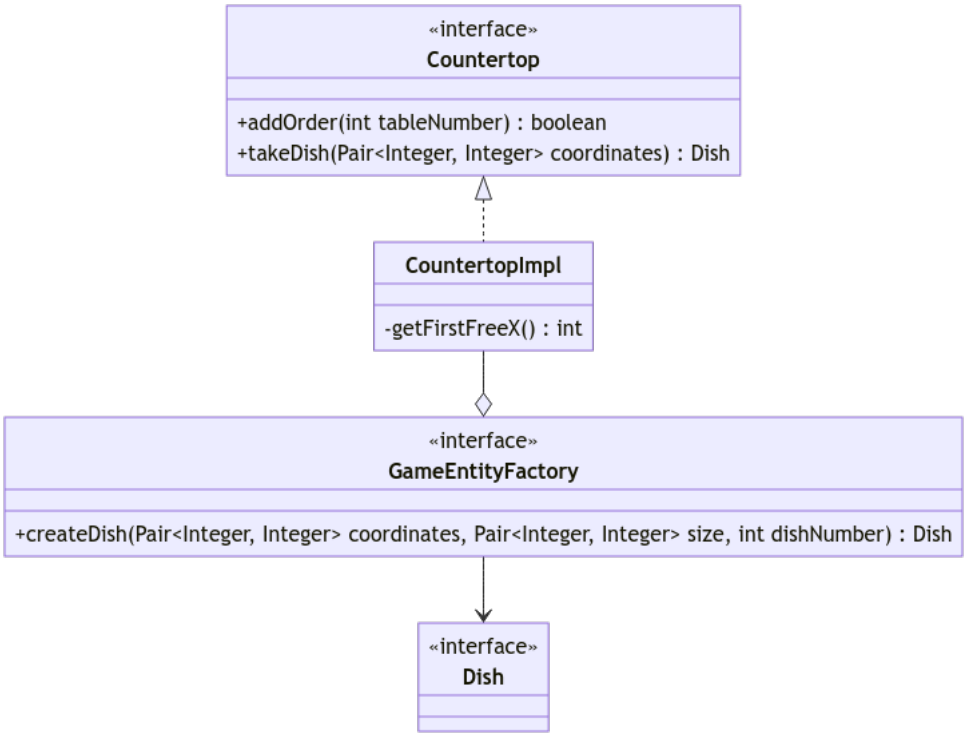
Aggiunta scalabile dei Power-up



Problema: La versione del gioco implementata supporta l'uso di diversi power-up. Si vuole trovare un modo scalabile per poter aggiungerne di nuovi senza dover modificare i componenti che ne fanno uso come la View.

Soluzione: I costi e le relative funzioni di ogni power-up vengono definiti nel Model. Per l'aggiunta di un nuovo power-up è quindi sufficiente modificare il Model ed aggiungere la chiamata al corretto power-up nel Controller.

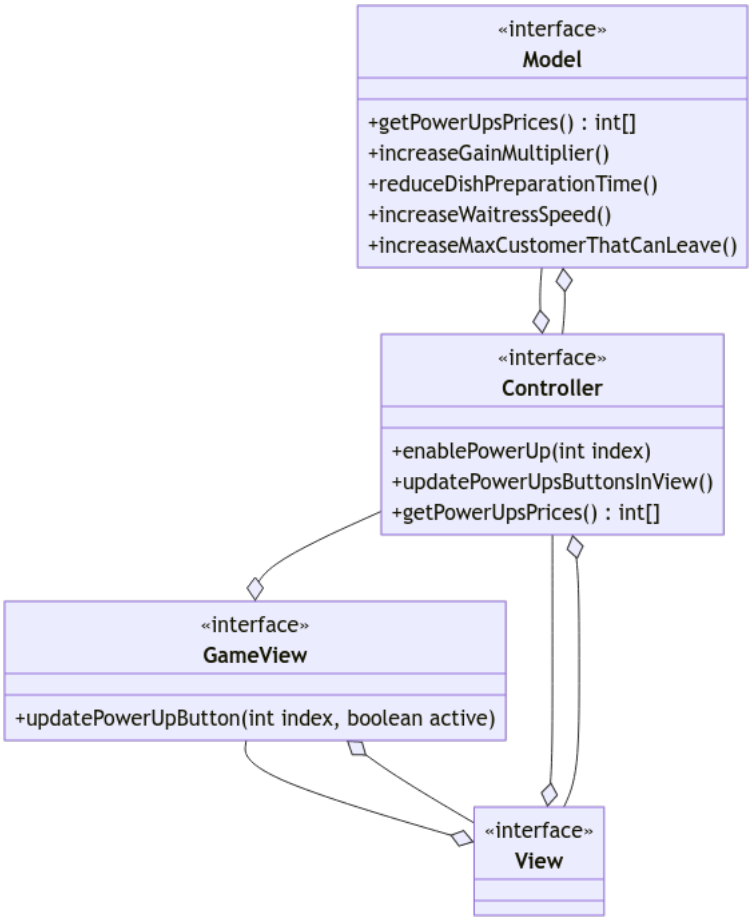
Si devono poter posizionare i piatti sul bancone in modo indipendente dal numero del tavolo



Problema: La cameriera durante il gioco può raccogliere gli ordini di qualsiasi tavolo senza seguire una sequenza specifica. I piatti, una volta pronti, devono quindi essere disponibili sul bancone in maniera indipendente dal numero del tavolo. Inoltre è possibile che la cameriera serva prima un piatto che è stato preparato più di recente.

Soluzione: Il piatto occuperà il primo spazio disponibile a partire da sinistra. Nell'inserire un nuovo ordine viene calcolata la posizione che il piatto occuperà tenendo conto della situazione attuale del bancone.

Aggiornamento interfaccia Power-up in base allo stato del Model

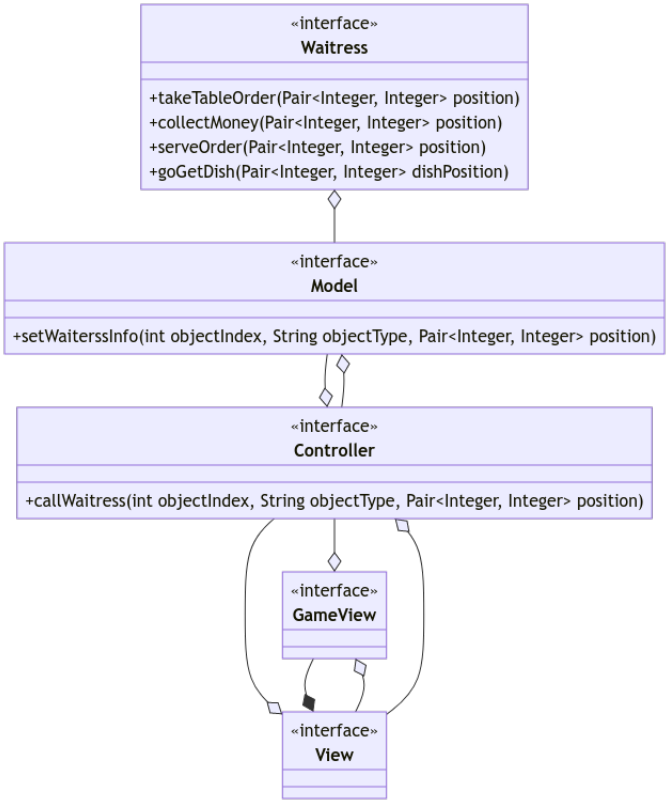


Problema: Il gioco consente di attivare ogni power-up un numero limitato di volte, questo in modo da garantire un certo bilanciamento nel gameplay. Per questo è necessario impedire che ogni tipologia di power-up sia attivabile all'infinito.

Soluzione: L'interfaccia grafica dispone di bottoni per l'attivazione di ogni power-up. Durante l'attivazione di un power-up viene determinato se quest'ultimo sia attivabile ulteriormente e l'interfaccia viene aggiornata di conseguenza, eventualmente disabilitando il bottone corrispondente al power-up.

Federico Benzi

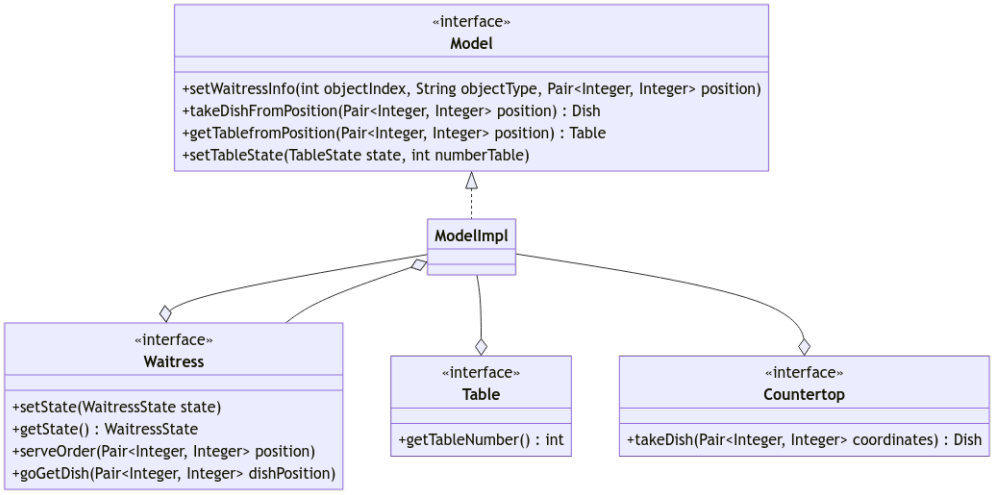
Riconoscimento dell'oggetto selezionato per gestire la corretta azione da parte della Cameriera



Problema: Gli input del gioco consistono in click del mouse che possono verificarsi all'interno di un tavolo o un piatto. E' necessario gestire in maniera diversa questo evento in base all'area del gioco nel quale viene effettuato ed allo stato dell'elemento coinvolto.

Soluzione: Sfruttando un Mouse Listener nel pannello di gioco è possibile identificare le coordinate ed il tipo di oggetto selezionato, implementando correttamente il metodo mouseClicked().

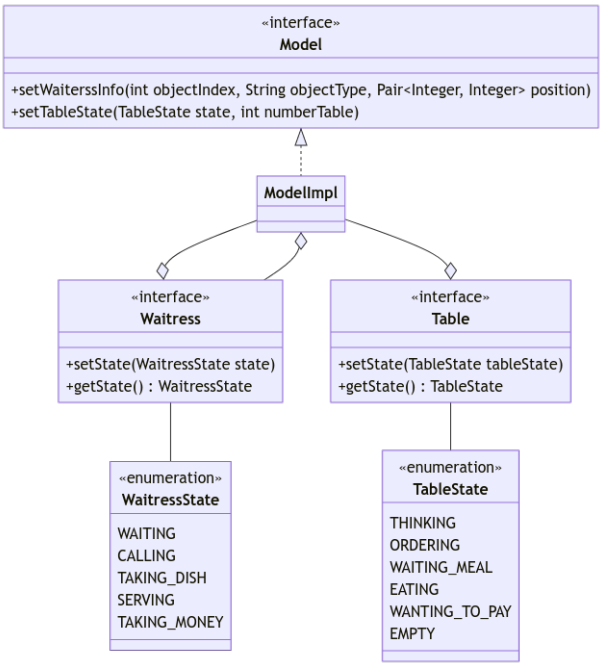
Interazioni tra Bancone e Cameriera per servire un piatto



Problema: E' necessario trovare un metodo condiviso fra Bancone e Cameriera in modo che quest'ultima possa ritirare un piatto quando è pronto. Inoltre occorre gestire le informazioni sui piatti presi dalla cameriera in modo da recapitarli ai tavoli corretti.

Soluzione: Selezionando un piatto, alla cameriera vengono passate le coordinate da raggiungere per poterlo recuperare. La cameriera modifica il suo stato in modo da poter prendere il piatto e successivamente, in base al numero del piatto, è in grado di determinare se giunta ad un tavolo, ad esso corrisponda uno dei piatti che trasporta.

Rappresentazione dei diversi comportamenti di Cameriera e Tavoli in base al loro stato

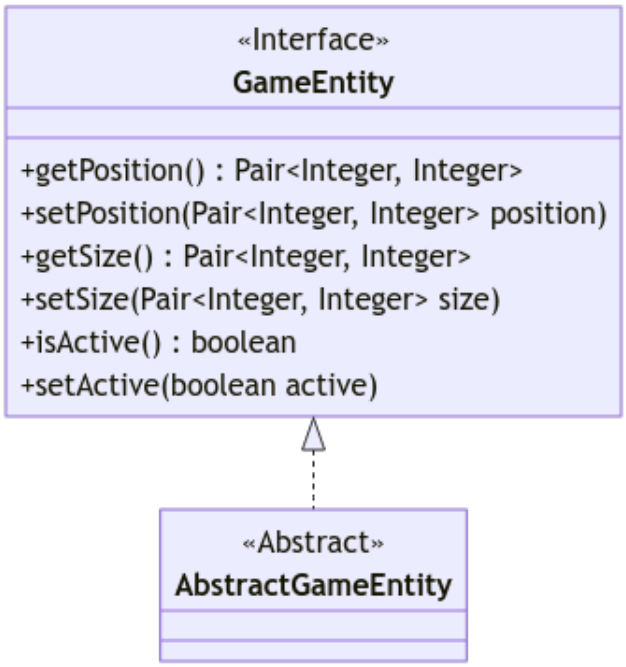


Problema: Durante l'esecuzione del gioco tavoli e cameriera possono avere comportamenti diversi in base al loro stato. E' necessario poter tracciare lo stato di queste entità in modo da definire il comportamento atteso.

Soluzione: Vengono introdotti gli stati TableState per i tavoli e WaitressState per la cameriera. Modificando lo stato di queste entità durante il gioco sarà possibile alterare il loro comportamento. Di conseguenza, queste entità avranno un comportamento specfico in baso allo stato in uso.

Pier Costante Babini

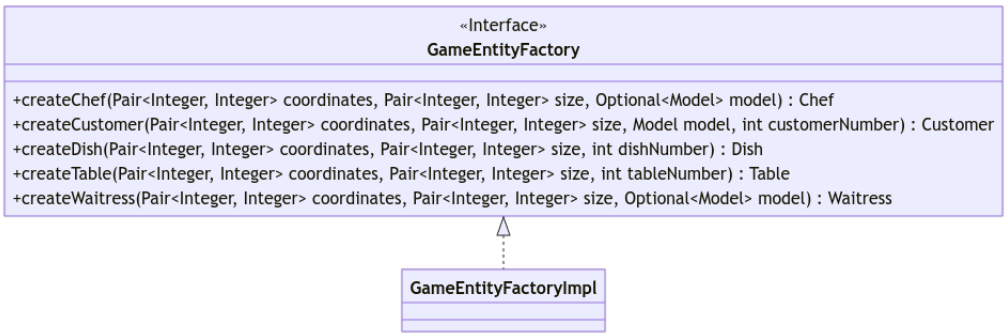
Definizione di entità riutilizzabili con caratteristiche comuni



Problema: Il gioco è caratterizzato da diverse entità, alcune più semplici ed altre con funzioni aggiuntive ma tutte con una struttura di base condivisa. E' necessario trovare un modo per rappresentare le entità evitando ripetizione di codice.

Soluzione: Per questo aspetto ho valutato diverse soluzioni, come l'uso del pattern Decorator che però non è risultato ottimale, in quanto ogni entità ha un comportamento specifico non in comune con le altre, per questo sarebbe risultato difficile costruire le entità come insieme di decoratori. Lo stesso per altri pattern come Builder. Per questo ho optato per l'ereditarietà.
Partendo da un'interfaccia GameEntity di base è stata creata la classe astratta AbstractGameEntity, quest'ultima definisce la struttura base di ogni GameEntity. E' quindi possibile definire altre entità con comportamenti specifici a partire da AbstractGameEntity.

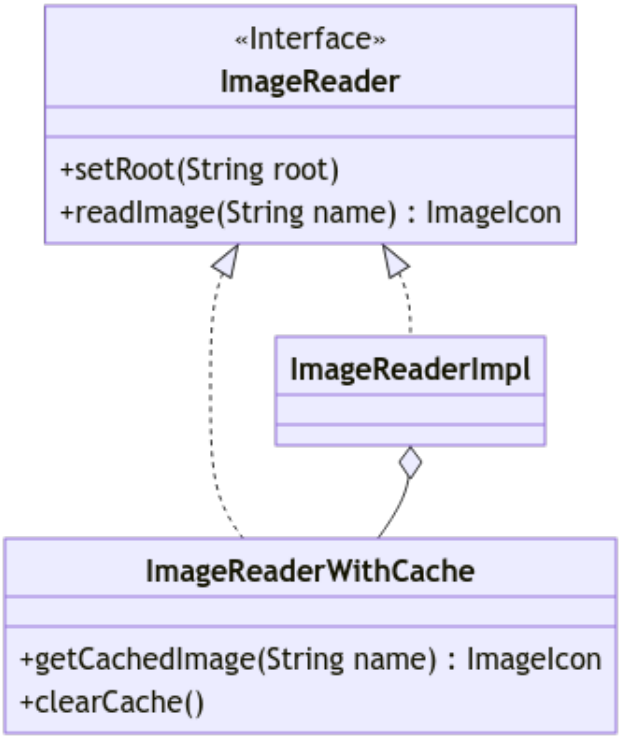
Creazione delle entità



Problema: Creare entità senza esporre direttamente la logica interna di creazione e l'implementazione effettiva.

Soluzione: Per gestire la creazione delle entità in modo efficiente ho scelto di utilizzare il pattern Factory Method. In questo modo viene nascosta la logica di creazione interna, garantendo così la modularità e flessibilità del codice. Ciò permette inoltre di sostituire l'implementazione sottostante di un'entità senza modificare il codice che ne fa uso.

Gestione efficiente delle immagini



Problema: Un'operazione onerosa nel contesto di un programma è la lettura dei file da disco, per questo è necessario trovare un modo per ripeterla il meno possibile, leggendo però i file necessari al corretto funzionamento.

Soluzione: Questa problematica è stata affrontata introducendo una classe di utilità ImageReader, responsabile della lettura delle immagini, gli unici assets di gioco. Tuttavia, oltre alla lettura, era necessario memorizzarle in modo da non dover ripetere questa operazione ed avere sempre un riferimento disponibile a runtime. Per questo ho creato la classe ImageReaderWithCache sfruttando il pattern Proxy.
Questa classe mantiene la struttura di ImageReader, introducendo una Mappa per il caching delle immagini. Ogni immagine può essere comodamente richiamata durante il gioco tramite il suo nome senza estensione. Di seguito vengono riportati alcuni esempi:

- per "./chef.gif" si usa "chef"

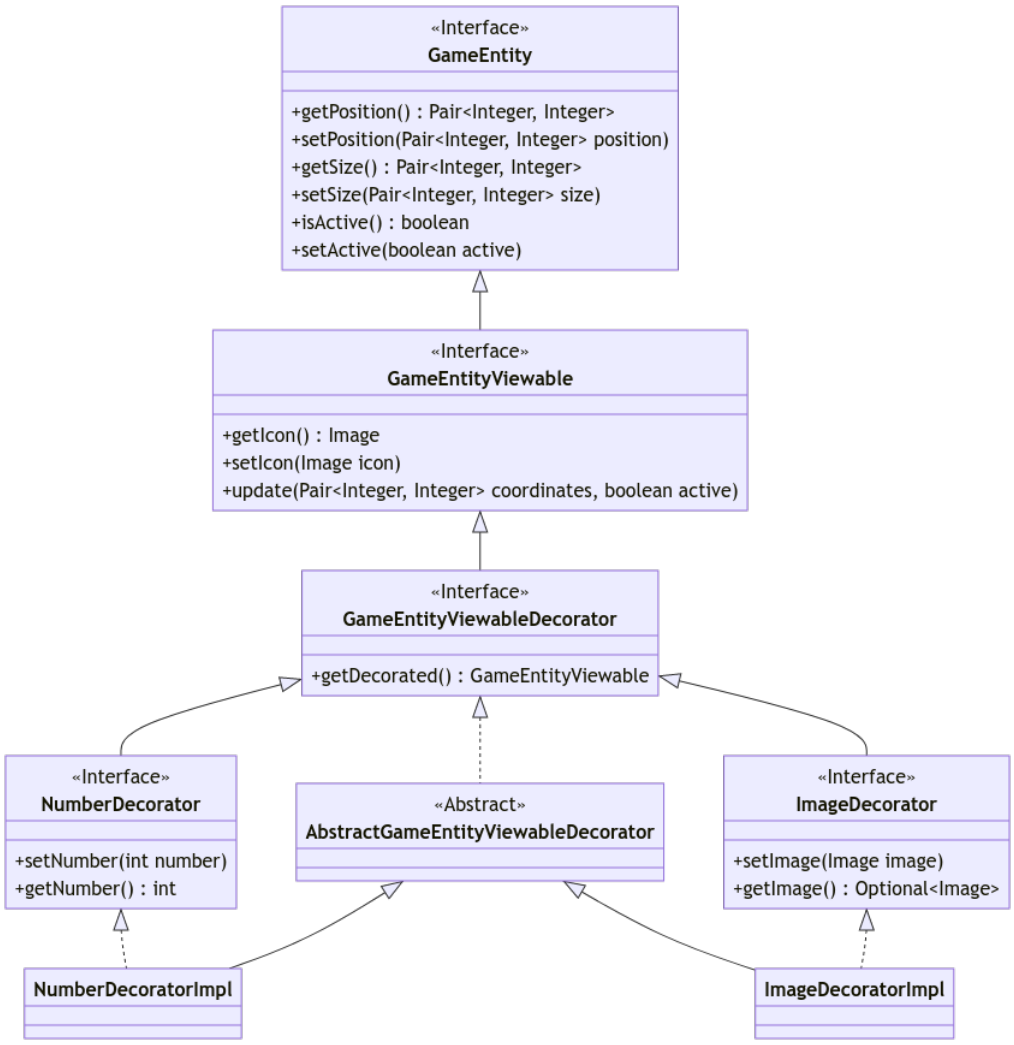
- per “./tables/withDish/tableWithDish1.png” si usa “tableWithDish1”

Questo grazie ad una funzione di parsing interna che genera il nome per il riferimento a partire dal percorso dell’immagine. Inoltre, avendo diverse immagini con una certa molteplicità (esempio tavolo con 1 persona, tavolo con 2 persone, gruppo di 3 clienti e così via) ho suggerito la memorizzazione utilizzando il formato:

`<nome_immagine><numero>.<estensione>`

In questo modo si evitando ripetizioni durante la lettura e viene definito uno standard condiviso sul nome delle immagini per il loro successivo utilizzo.

Rappresentare correttamente alcune entità della view con informazioni aggiuntive

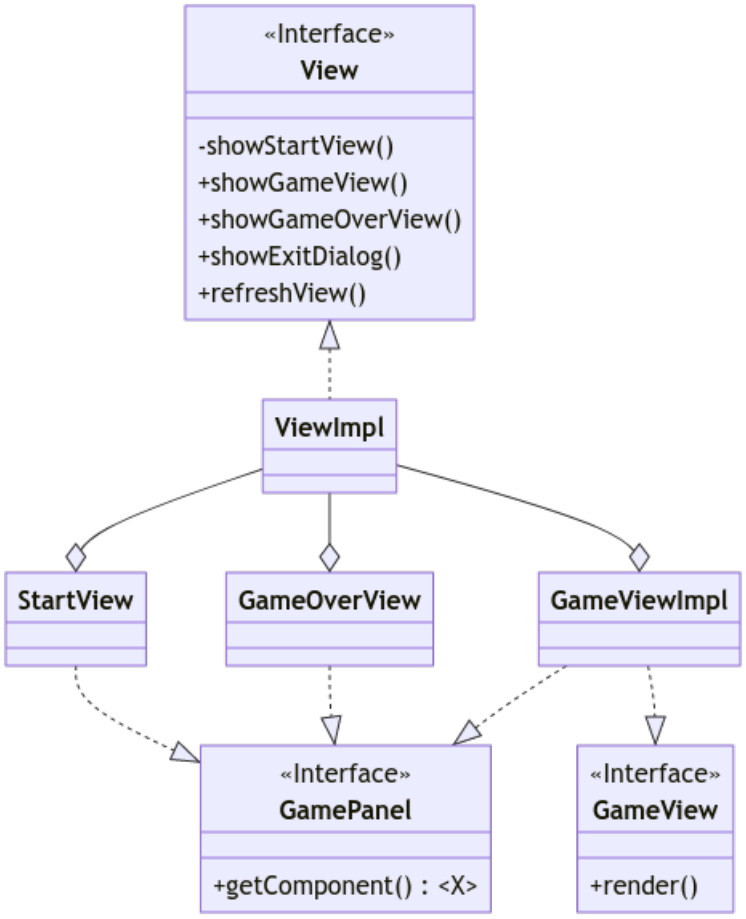


Problema: Le entità di gioco vengono rappresentate nella View come GameEntityViewable, classe che estende una GameEntity aggiungendo il supporto ad un'icona. Tuttavia per alcune di esse è necessario rappresentare immagini o dati aggiuntivi.

Soluzione: L'utilizzo del pattern Decorator mi è sembrata la scelta migliore in questo caso. Infatti garantisce una certa flessibilità nell'aggiungere funzionalità ad un'entità, senza modificarne la struttura di base. Per alcune

entità era richiesta l'aggiunta di una sola funzionalità, mentre per altre di più funzionalità; il Decorator permette di modellare questa richiesta, componendo diversi decoratori per ottenere combinazioni di funzionalità personalizzate. Nello schema è mostrata la sola struttura implementativa del Decorator, tralasciando gli utilizzi nello specifico.

Gestione flessibile e modulare della GUI



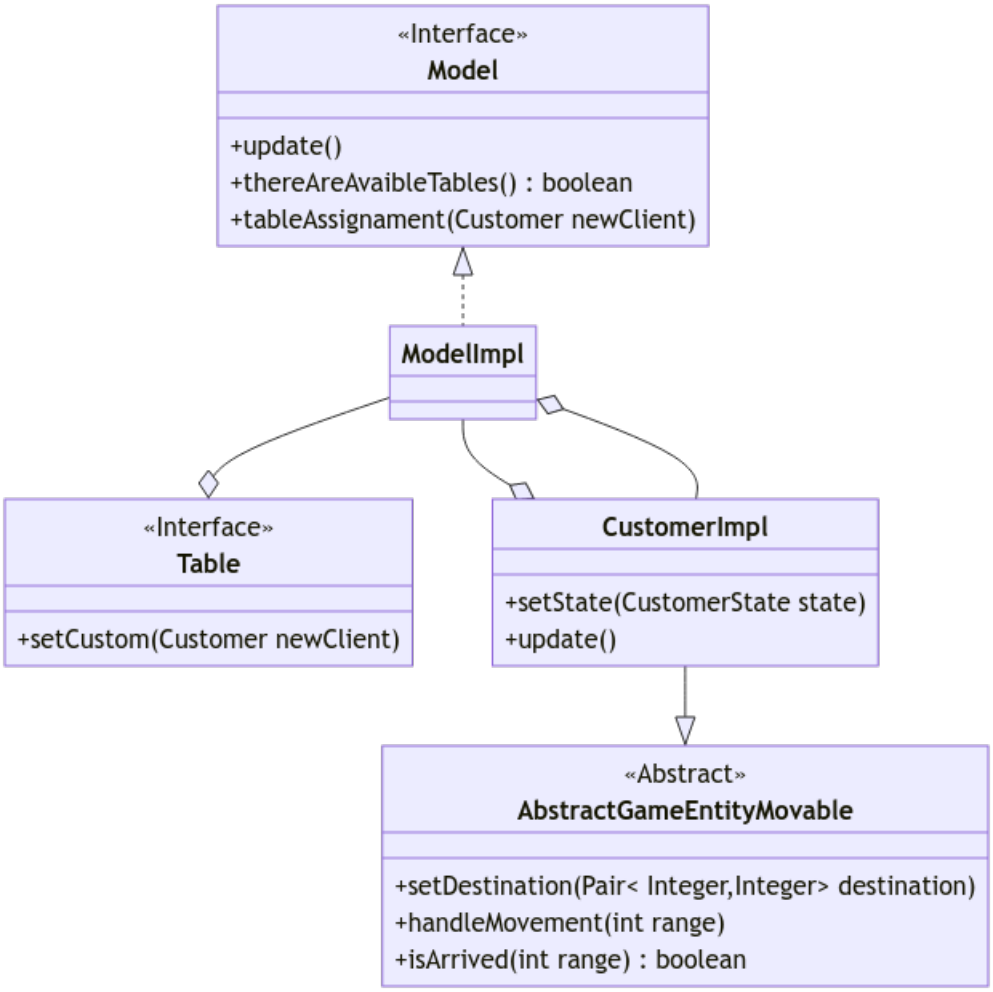
Problema: L'interfaccia grafica di gioco prevede più view, ognuna delle quali con elementi e caratteristiche specifiche. E' necessario trovare un modo per implementare in maniera ordinata e modulare la gestione delle view, affinchè risulti semplice la modifica di una determinata view e l'aggiunta di nuove, senza impattare su quelle esistenti.

Soluzione: Per risolvere questo problema, partendo dalla libreria scelta per lo sviluppo della grafica, ho inizialmente pensato ad un JFrame "madre" in grado di ospitare tanti JPanel "figli". Per esporre correttamente il comportamento di ogni componente ho quindi creato un'interfaccia View ed un GamePanel utilizzabile come base per la creazione di pannelli di gioco personalizzati.

Tuttavia, ho trovato questa soluzione altamente specifica; pertanto, ho optato per implementare la View principale come una classe ordinaria, in cui viene creato il JFrame senza imporne l'uso. Lo stesso è stato fatto per i pannelli di gioco, che sono stati sviluppati partendo da un'interfaccia generica. In questo modo la View principale è in grado di aggiungere, aggiornare e rimuovere ogni pannello in modo ottimale, inoltre ogni pannello mantiene un'implementazione separata dagli altri e la sua presenza (o mancanza) non li influenza.

Marco Giorgi

Creazione e assegnazione tavolo o posto in fila al Cliente

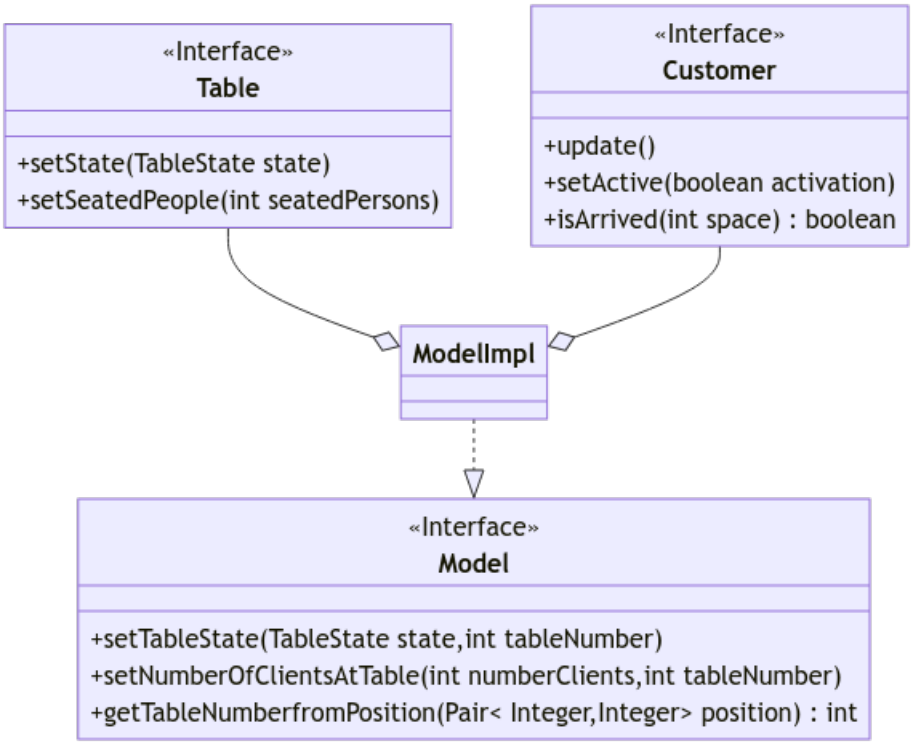


Problema: Ad intervalli regolari, un cliente, o un gruppo, entreranno nel ristorante. E' necessario che il sistema assegni a quest'ultimi un tavolo libero, oppure un posto in fondo alla fila, nel caso siano tutti occupati.

Soluzione: I Clienti estendono la classe AbstractGameEntityMovable che garantisce la gestione dei movimenti delle figure che lo richiedono. In seguito alla creazione di un nuovo cliente, il sistema controlla se ad almeno uno dei tavoli non sia già assegnato il corrispondente cliente che lo occupa. In caso

affermativo imposta le coordinate del tavolo al Cliente, che sarà in grado di muoversi autonomamente verso di esso. Viene anche reso "occupato" il tavolo, in modo che, anche riducento il tempo di creazione dei clienti, non si rischi che 2 clienti diversi vogliano occupare lo stesso tavolo. Nel caso non sia disponibile, al cliente sarà assegnata una posizione in fila, calcolandola tramite la lunghezza di quest'ultima.

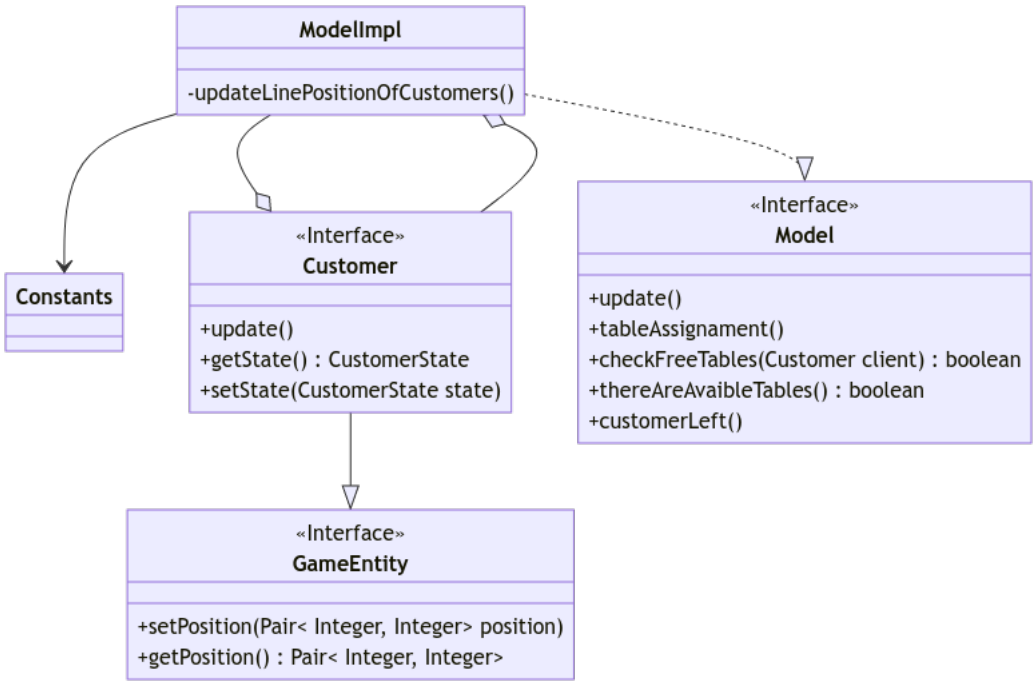
Gestione delle interazioni tra Cliente/Tavolo



Problema: E' necessario realizzare la gestione ed i metodi con cui clienti ed tavoli comunicano tra di loro, facendo in modo che una modifica in uno non influenzi l'altro.

Soluzione: Per questa parte ho pensato di utilizzare il pattern Mediator, in modo da rendere il più indipendenti possibili Tavoli e Clienti. Così facendo si rendere più semplice il loro riutilizzo, inoltre nel caso si richieda di modificare come interagiscono tra loro, sarà sufficente modificare solo il Mediator.

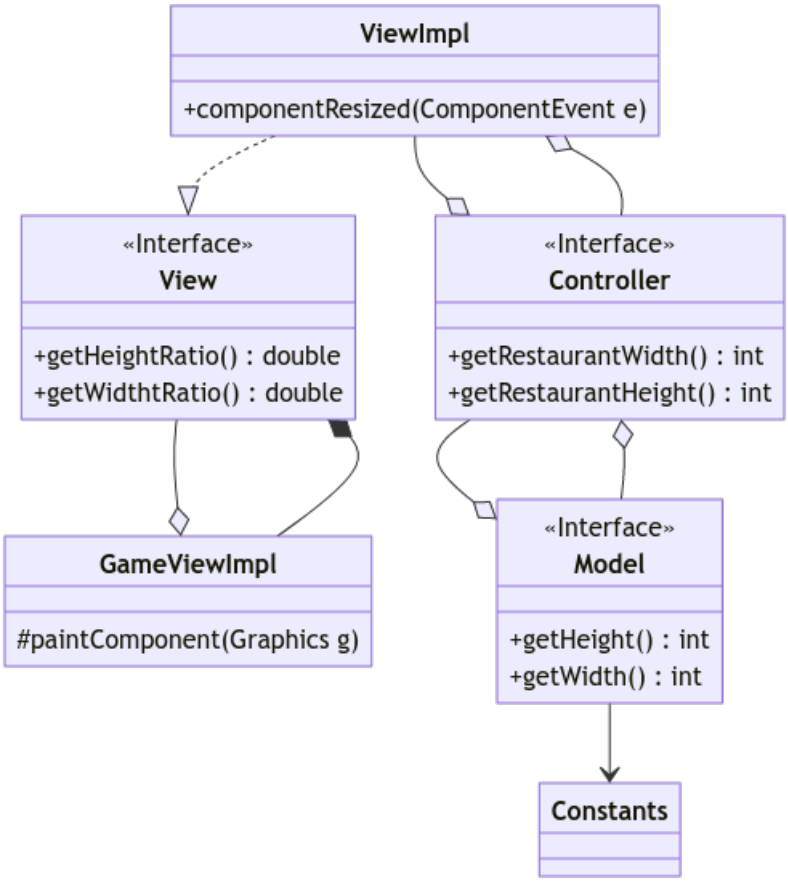
Gestione della coda e dei clienti arrabbiati



Problema: I clienti una volta passato un certo tempo in fila, si arrabbiano e lasciano il ristorante. E' necessario gestire quindi il loro livello di pazienza prima di andarsene, e l'aggiornamento delle posizioni in fila nel caso in cui un cliente se ne vada o si sieda.

Soluzione: Ogni cliente in fila controlla se si è liberato un tavolo e, se si trova nella prima posizione in fila, si dirige verso di esso per occuparlo. Cosi facendo i restanti clienti in fila potranno avanzare di una posizione. Quest'ultima azione si verifica anche nel caso un cliente arrabbiato decida di andarsene. La velocità con la quale un cliente diventa arrabbiato corrisponde ad una variabile facilmente modificabile, nel caso si voglia rendere il gioco più difficile. Questa soluzione adotta sempre il pattern Mediator visto in precedenza, così facendo si rendono facilemte modificabili sia il numero di tavoli presenti nel ristorante, che il numero di clienti che possono entrare.

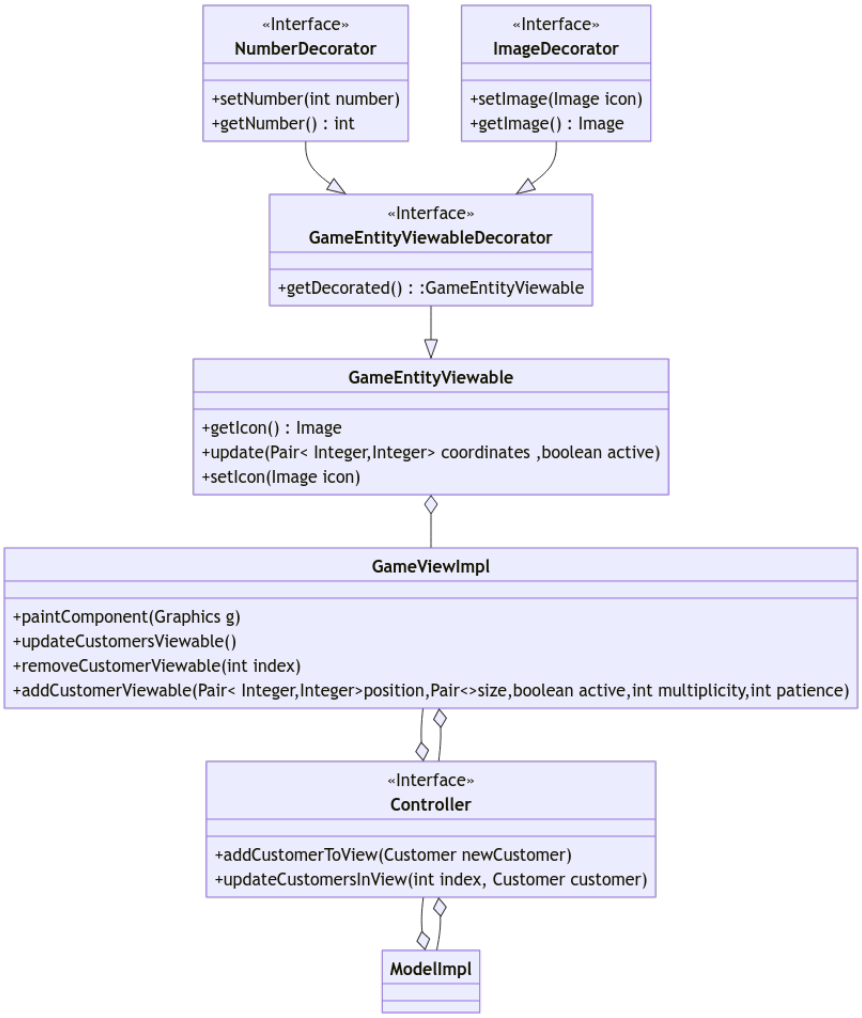
Implementazione di una finestra di gioca dinamica



Problema: La presenza in commercio di monitor di dimensioni diverse, rendono necessaria la realizzazione di una finestra di gioco dinamica in grado di adattarsi ad ogni schermo.

Soluzione: L'iniziale idea era di implementare una classe dedita al ridimensionamento delle immagini, e al ricalcolo delle posizioni. Si è dimostrata inefficente e dispendiosa di risorse da parte del sistema. Ragion per cui si è optato per la realizzazione di una dimensione "Logica" e fissa, del modello, e di un sistema in grado di calcolare il rapporto tra l'effettiva dimensione della finestra e quella logica. Il rapporto viene calcolato ogni qual volta si ridimensioni la finestra, ed i dati ricavati vengono utilizzati dalla View durante il repaint(). Con questa soluzione le entità vengono proiettate correttamente nella View a prescindere dalla dimensione della finestra.

Aggiornamento e stampa grafica dei Clienti



Problema: La simulazione del movimento dei clienti, e la rappresentazione, sotto formato grafico, delle icone che suggeriscono al giocatore come comportarsi, richiedono una costante comunicazione per le richieste di aggiornamento tra la logica del gioco e l'interfaccia grafica.

Soluzione: Data la presenza di 2 tipologie di Clienti: una che utilizza un'icona aggiuntiva per mostrare all'utente il proprio livello di pazienza, ed una priva di quest'ultima; E dalla presenza nel gioco di figure con un comportamento simile, si è pensato di utilizzare il pattern Decorator, dato dal fatto

che le 2 tipologie di clienti sono di base uguali, e contenenti le stesse informazioni. L'unica differenza consiste nel livello di pazienza, gestito sotto forma numerica tramite NumberDecorator, e dalla rispettiva rappresentazione in forma grafica, gestita dall'ImageDecorator. L'aggiornamento delle informazioni dei clienti lato model, vengono trasmesse attraverso il controller alla view, facendo si che il Model rimanga all'oscuro e distaccato dalla parte grafica.

NOTA: il Pattern Decorator è stato sviluppato e pensato con Pier Costante Babini.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare correttamente il testing del software abbiamo utilizzato la suite JUnit 5. La parte di View è stata testata manualmente acccertandosi che l'applicazione rispondesse correttamente a prescindere dalla dimensione dello schermo in uso e controllando che i comportamenti che ci aspettavamo non venissero disattesi durante l'esecuzione.
Per quanto riguarda le entità di gioco, abbiamo implementato i seguenti test:

- ChefTest: in questa classe vengono testati i principali comportamenti e stati dello Chef inclusa l'attivazione del power-up per la riduzione del tempo di preparazione dei piatti.
- CustomerTest: consente di testare i comportamenti dei Clienti, e le interazioni di quest'ultimi con le entità Tavolo. Vengono testate le azioni di clienti in base al loro stato, ovvero mentre sono in fila, e mentre vanno a sedersi.
- DishTest: la classe verifica la corretta creazione di un piatto, verificando che l'oggetto creato ed il suo numero corrispondano a quanto atteso.
- TableTest: questa classe testa la creazione di un tavolo, verificando anche la corretta assegnazione di un cliente. Inoltre viene testato il corretto cambiamento di stato.
- WaitressTest: la classe verifica il corretto funzionamento della cameriera, testando tutti i comportamenti e funzioni previste come recarsi ad un tavolo per prendere un ordine, servirlo, far pagare i clienti e attivare il power-up per l'aumento della velocità di movimento.

Mentre per le classi rimanenti:

- CountertopTest: questa classe verifica il corretto funzionamento del Countertop provando ad aggiungere un ordine per la preparazione, verificando che sia il prossimo ordine da preparare ed impostandolo come pronto. Viene inoltre testato il metodo per fornire un piatto alla Waitress rimuovendolo dal bancone.

- DirectionTest: la classe testa il funzionamento dell'Enum Direction, classe di utilità per la gestione del movimento delle entità.

- GameTimerTest: testa il funzionamento del timer di gioco, classe il cui compito è richiamare ogni secondo un metodo passato all'atto della creazione, nel gioco un metodo per decrementare il tempo rimanente. Vengono testati i metodi per far partire il timer, metterlo in pausa, resettarlo, farlo riprendere e stopparlo.

- ImageReaderTest: testa il funzionamento della classe che implementa la lettura delle immagini da disco.

- ImageReaderWithCacheTest: testa il funzionamento della classe per memorizzare le immagini in cache. Nello specifico vengono testati i metodi per leggere e aggiungere un'immagine alla cache, ottenere il riferimento ad un'immagine letta e per svuotare la cache.

- ModelTest: in questa classe sono raccolti diversi test che che permettono alle entità del gioco di scambiarsi informazioni e comunicare tra loro. Sono contenuti anche test che verificano la corretta inizializzazione dello stato logico del gioco.

Nell'effettuare determinati test, in mancanza di strumenti già disponibili per verificarne il corretto svolgimento, sono stati implementati appositi getters.

3.2 Metodologia di lavoro

In seguito alla fase di analisi abbiamo pensato a quella che poteva essere una struttura scheletro del progetto, definendo alcune interfacce e implementando qualche classe in modo da avere più chiara la struttura del codice.
Successivamente abbiamo perfezionato il tutto con ulteriori incontri e suddiviso il lavoro. Abbiamo lavorato per gran parte del tempo in maniera autonoma, facendo ogni tanto il punto sullo stato del progetto e su come far comunicare fra di loro le classi sviluppate. Per quanto riguada l'uso del

DVCS, abbiamo scelto la versione semplificata, creando un repository centrale clonato da ogni membro. Le modifiche successive sono state apportate tramite gli appositi comandi di *push*, *pull*, *merge* e così via.

Matteo Donati

In autonomia mi sono occupato di:

- Implementazione enablePowerUp(), getPowerUpsPrices(), updatePowerUpsButtonsInView() in ControllerImpl
  (package it.unibo.dinerdash.controller.impl)
- Implementazione di CountertopImpl
  (package it.unibo.dinerdash.model.impl)
- Implementazione di DishImpl (package it.unibo.dinerdash.model.impl)
- Implementazione di strutture e metodi per la gestione dei Power-up nel Model (package it.unibo.dinerdash.model.impl)
- Generazione dei bottoni dei Power-up in GameViewImpl
  (package it.unibo.dinerdash.view.impl)
- Gestione della rappresentazione dei Power-up in GameViewImpl (package it.unibo.dinerdash.view.impl)
- Ricerca degli assets di gioco

In collaborazione mi sono occupato di:

- Definizione di alcuni metodi in Controller con Pier Costante Babini e Marco Giorgi (package it.unibo.dinerdash.controller.api)
- Definizione dell'interfaccia di Countertop con Pier Costante Babini (package it.unibo.dinerdash.model.api)
- Definizione dell'interfaccia di Model con Pier Costante Babini e Marco Giorgi (package it.unibo.dinerdash.model.api)
- Definizione della struttura per memorizzare in modo efficiente gli assets con Pier Costante Babini (package resources)

Federico Benzi

In autonomia mi sono occupato di:

- Definizione degli stati di un tabolo in TableStates (package it.unibo.dinerdash.model.api.states)
- Definizione degli stati della cameriera in WaitressStates (package it.unibo.dinerdash.model.api.states)
- Implementazione di WairessImpl (package it.unibo.dinerdash.model.impl)
- Implementazione di TableImpl (package it.unibo.dinerdash.model.impl)

In collaborazione mi sono occupato di:

- Rappresentazione grafica dei tavoli e della cameriera in GameViewImpl con Pier Costante Babini (package it.unibo.dinerdash.view.impl)
- Risoluzione di diversi errori PMD e Checkstyle (package all)
- Implementazione di un modo per far comunicare la Cameriera con il Controller in GameViewImpl con Marco Giorgi (package it.unibo.dinerdash.view.impl)

Pier Costante Babini

In autonomia mi sono occupato di:

- Definizione ed implementazione del GameLoop (package it.unibo.dinerdash.engine)
- Implementazione di AbstractGameEntity (package it.unibo.dinerdash.model.api.gameentities)
- Definizione ed implementazione di Chef (package it.unibo.dinerdash.model)
- Definizione di GameEntity (package it.unibo.dinerdash.model.api.gameentities)
- Definizione ed implementazione di GameEntityFactory (package it.unibo.dinerdash.model.api.gameentities)
- Definizione di GameState (package it.unibo.dinerdash.model.api.states)

- Implementazione dei metodi setController(), init(), clear(), start(), pause(), stop(), gameOver(), restart(), decrementRemainingTime(), getters e setters per Coins e GameState, earnMoneyFromTable(), getDishToPrepare(), completeDishPreparation() in ModelImpl (package it.unibo.dinerdash.model.impl)
- Definizione ed Implementazione di tutto il pacakge Utility (package it.unibo.dinerdash.utility)
- Definizione di AbstractGameEntityViewableDecorator, GameEntityViewable, GameEntityViewableDecorator, GamePanel, ImageDecorator, NumberDecorator, OutlinedLabel (package it.unibo.dinerdash.view.api)
- Implementazione di ImageDecoratorImpl, NumberDecoratorImpl, GameEntityViewableImpl (package it.unibo.dinerdash.view.api)
- Implementazione di GameOverView, StartView (package it.unibo.dinerdash.view.impl)
- Definizione di MainApplication (package it.unibo.dinerdash)

In collaborazione mi sono occupato di:

- Definizione dell'interfaccia del Model con Marco Giorgi e Matteo Donati (package it.unibo.dinerdash.model.api)
- Definizione dell'interfaccia del Controller con Marco Giorgi (package it.unibo.dinerdash.controller.api)
- Implementazione del Controller con Marco Giorgi, Matteo Donati (package it.unibo.dinerdash.controller.impl)
- Implementazione di AbstractGameEntityMovable con Marco Giorgi (package it.unibo.dinerdash.model.api.gameentities)
- Definizione iniziale dell'interfaccia Dish e Countertop con Matteo Donati (package it.unibo.dinerdash.model.api)
- Implementazione di GameEntityMovable con Marco Giorgi (package it.unibo.dinerdash.model.api.gameentities)
- Definizione delle interfacce Table e Waitress con Marco Giorgi (package it.unibo.dinerdash.model.api.gameentities)
- Definizione di Constants con Marco Giorgi (package it.unibo.dinerdash.model.api)

- Implementazione di DishImpl con Matteo Donati
(package it.unibo.dinerdash.model.impl)
- Implementazione di update() in ModelImpl con Marco Giorgi (package it.unibo.dinerdash.model.impl)
- Definizione di GameView con Marco Giorgi e Matteo Donati (package it.unibo.dinerdash.view.api)
- Definizione di View con Marco Giorgi (package it.unibo.dinerdash.view.api)
- Implementazione di ViewImpl con Marco Giorgi
(package it.unibo.dinerdash.view.impl)
- Implementazione di GameViewImpl con Marco Giorgi e Matteo Donati
(package it.unibo.dinerdash.view.impl)

Marco Giorgi

In autonomia mi sono occupato di:

- Implementazione di CustomerImpl
(package it.unibo.dinerdash.model.impl)
- Implementazione di una finestra di gioco Dinamica
(package it.unibo.dinerdash.view.impl)
- Implementazione di ViewImpl.componentResized()
(package it.unibo.dinerdash.view.impl)
- Implementazione di ModelImpl.addCustomer() per la creazione di clienti (package it.unibo.dinerdash.model.impl)
- Gestione dell'assegnamento di un tavolo o di un posto in fila ad un cliente (package it.unibo.dinerdash.model.impl.ModelImpl)
- Gestione aggiornamento delle posizioni in fila dei clienti
(package it.unibo.dinerdash.model.impl.ModelImpl)
- Gestione per la rimozione Logica e Grafica di un cliente arrabbiato, tramite ModelImpl.removeAngryCustomers()
(package it.unibo.dinerdash.model.impl)
- ModelImpl.removeCustomerViewable()
(package it.unibo.dinerdash.view.impl)

- ModelImpl.getTableNumberfromPosition()
  (package it.unibo.dinerdash.model.impl)
- isArrived() (package it.unibo.dinerdash.model.api.gameentities)
- ModelImpl.getWidth() e ModelImpl.getHeight() per ottenere le dimensioni logiche del gioco (package it.unibo.dinerdash.model.impl)
- Aggiornamento dei clienti e comunicazione di quest'ultimi con la controparte grafica, tramite i metodi presenti in ModelImpl, ControllerImpl, GameViewImpl
- implementazione di addCustomerToView(), removeCustomerInView(), updateCustomerInView(),getRestaurantWidth() e getRestaurantHeight() in ControllerImpl (package it.unibo.dinerdash.controller.impl)
- GameViewImpl.addCustomerViewable()
  (package it.unibo.dinerdash.view.impl)
- GameViewImpl.updateCustomersViewable()
- Stampa grafica dei clienti e implementazione della dinamicità per le altre entità di gioco durante la stampa con GameViewImpl.paintComponent()
  (package it.unibo.dinerdash.view.impl)

In collaborazione mi sono occupato di:

- Implementazione del Controller con Pier Costante Babini, Matteo Donati (package it.unibo.dinerdash.controller.impl)
- Definizione dell'interfaccia del Model con Pier Costante Babini e Matteo Donati (package it.unibo.dinerdash.model.api)
- Implementazione di AbstractGameEntityMovable con Pier Costante Babini (package it.unibo.dinerdash.model.api.gameentities)
- Implementazione di Decoratori per le GameEntityViewable,assieme a Pier Costante Babini () (package it.unibo.dinerdash.view.api)
- Gestione delle interazioni tra Tavoli e Clienti, con Federico Benzi (package it.unibo.dinerdash.model.impl)
- Ottimizzazione del codice tramite GameEntityMovable. GameEntityMovable.handleMovement ed Constants, con Pier Costante Babini (package it.unibo.dinerdash.model.api.gameentities) e (package it.unibo.dinerdash.model.api)

- Definizione delle interfacce Table e Waitress con Pier Costante Babini (package it.unibo.dinerdash.model.api.gameentities)

3.3 Note di sviluppo

Federico Benzi

- Utilizzo di Optional:
  - Permalink di esempio
- Utilizzo di Lambda Expressions:
  - Permalink di esempio
- Utilizzo di Stream:
  - Permalink di esempio

Matteo Donati

- Utilizzo di Optional:
  - Permalink di esempio
- Utilizzo di Stream:
  - Permalink di esempio
- Utilizzo di Lambda Expressions:
  - Permalink di esempio

Pier Costante Babini

- Utilizzo di Lambda Expressions (alcuni esempi):
  - Permalink di esempio 1
  - Permalink di esempio 2
  - Permalink di esempio 3
  - Permalink di esempio 4
- Utilizzo di Stream:

    - Permalink di esempio
- Utilizzo di Generics:
    - Permalink di esempio
- Utilizzo di Functional Interfaces:
    - Permalink di esempio
- Utilizzo di Method Reference:
    - Permalink di esempio 1
    - Permalink di esempio 2
    - Permalink di esempio 3
    - Permalink di esempio 4
- Utilizzo di Optional (alcuni esempi):
    - Permalink di esempio 1
    - Permalink di esempio 2
    - Permalink di esempio 3
    - Permalink di esempio 4

Marco Giorgi

- Utilizzo di Stream:
    - Permalink di esempio 1
    - Permalink di esempio 2
- Utilizzo di Lambda Expressions:
    - Permalink di esempio
- Utilizzo di Functional Interfaces:
    - Permalink di esempio
- Utilizzo di Optional:
    - Permalink di esempio 1
    - Permalink di esempio 2

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Federico Benzi

Il progetto è stato sviluppato con l'idea di essere la versione migliore di se stesso, evitando codice base e ripetizioni, questo mi ha portato a scoprire che la mia conoscenza del linguaggio era inferiore a quella dei miei compagni. Per lo sviluppo del progetto mi sono ritrovato più volte a controllare e ricontrollare le slide proposte in classe, rimarcando anche più volte gli stessi argomenti. Grazie a questo ho potuto capire meglio concetti e strutture.

Pier Costante Babini

Mi ritengo abbastanza soddisfatto di questo progetto, infatti mi ha permesso di conoscere ed approfondire un modus operandi che mi tornerà molto utile in progetti ancora più complessi. Nel progetto ho approfondito elementi di Java come lambda/streams la cui semplicità e potenza mi ha aiutato a scrivere codice pulito e che ho apprezzato molto. All'interno del gruppo ho avuto il compito di preparare la struttura iniziale del repository e di coadiuvare gli sforzi collettivi per raggiungere l'obiettivo. Lavorare in gruppo mi ha permesso inoltre di confrontarmi e conoscere altri punti di vista nella stesura delle classi. Il risultato del nostro lavoro è sicuramente qualcosa di cui vado fiero e che manterrò sul mio profilo GitHub. Complessivamente il bilancio di questa esperienza è positivo, tuttavia una partecipazione più attiva da parte di alcuni membri del gruppo, sia in fase di progettazione che in fase implementativa, avrebbero sicuramente contribuito ad una migliore analisi del dominio, ad un'implementazione più efficiente di determinati aspetti ed in generale una qualità ancora più alta del progetto.

Matteo Donati

Sono estremamente soddisfatto del completamento del progetto di programmazione ad oggetti. All'inizio del mio progetto ero un po' scettico a riguardo poichè le mie conoscenze non erano cosi solide e ampie per poter affrontare un progetto di questo tipo, però devo ammettere che con qualche aiuto da parte del team sono riuscito ad affrontare qualsiasi problematica mi si sia posta davanti. Sicuramente dovrò porre più attenzione nella gestione della scalabilità del mio codice in futuro, infatti più volte mi son ritrovato con dei codici che non permettevano una facile espansione. Il progetto è stato fondamentale per scoprire il lavoro di squadra in questa tipologia di lavoro, e cosa significhi lavorare ad un progetto facendo parte di un team. Non penso di continuare il progetto in questo momento, causa impegni universitari e lavorativi, nonostante mi sia trovato estremamente bene con il team.

Marco Giorgi

Valuto il mio lavoro in maniera abbastanza positiva. Essendo la prima volta che ho lavorato ad un progetto in gruppo, penso di essermi impegnato proponendo soluzioni a problemi o idee sulle modalità di lavoro. Seppur le parti da me sviluppate possano essere migliorate e ottimizzate, ritengo che contengano le giuste basi, utili per sviluppare progetti di una miglior qualità in futuro. La presenza di lacune teoriche da parte di alcuni membri ha reso il progetto più lungo in termini di tempistiche, e la necessità della presenza di altri membri per aiutare a colmarle. I punti più dolenti rimangono: mancanza di esperienza nell'applicazione di pattern, codice in alcune parti non ben indentato o poco leggibile, e applicazione di tecniche per ridurre la ripetizione di codice.

Appendice A

Guida utente

La schermata iniziale del gioco presenta un semplice menù dal quale è possibile scegliere se iniziare una partita o uscire dal programma. Per avviare una partita sarà quindi necessario premere “Start game”. Una volta dentro al gioco, si visualizzerà la seguente schermata:

Figura A.1: Schermata di gioco.

Durante la partita si potranno controllare le azioni ed i movimenti della cameriera. Come anticipato i clienti entreranno nel locale in cerca di un posto che, nel caso fose disponibile, andranno ad occupare, altrimenti resteranno in fila. Compito della cameriera è gestire parallelamente le richieste di tutti i clienti cercando di non far perdere la pazienza a quelli in fila, visto che ciò

comporterà prima o poi il termine della partita. Gli unici comandi disponibili consitono in click che possono essere effettuati su:

- "Tavolo": per raccogliere un'ordine, servirlo e far pagare i clienti.
- "Piatto": per prenderlo dal bancone e successivamente servirlo al tavolo corrispondente.
- "Power-up": per attivare i power-up, quando disponibili, cliccando sui bottoni a destra. Le icone semplificate ne suggeriscono le funzioni che, ad ogni modo, sono:
  - Riduzione del tempo di preparazione dei piatti.
  - Aumento velocità di movimento della cameriera.
  - Aumento numero di clienti che possono andarsene dalla fila prima di perdere.
  - Moltiplicatore monete guadagnate da ogni ordine.

E' inoltre possibile mettere in pausa il gioco in qualsiasi momento cliccando sul tasto "Pause". Da qui si aprirà un menù secondario che permetterà di continuare la partita, ricominciarla oppure uscire dal gioco.
Una volta terminata la partita la schermata di Game Over mostrerà il risultato e consentirà di giocare nuovamente oppure uscire dal programma.

Bibliografia

[1] Wikipedia. Pagina di Diner Dash.
Link: `https://it.wikipedia.org/wiki/Diner_Dash`.

[2] Fonti e attribuzioni: Le immagini ed icone del gioco provengono da Diner Dash 2003, originariamente sviluppato da GameLab e pubblicato da PlayFirst. Il software ad oggi appartiene ad Electronic Arts (EA) che ne detiene i diritti, pertanto per l'utilizzo degli assets in questo progetto è stata richiesta l'autorizzazione all'uso che, che per "educational purposes", ci è stata concessa.