Relazione “Tower Jojo”

Abu Ismam, Hu Marco, Mattiussi Vlad, Zhu Zheliang
25 aprile 2019

Indice

Capitolo 1: Analisi

1.1 Requisiti

Il gruppo si pone come obiettivo quello di ricreare un classico videogioco strategico del sottogenere tower-defense.
L'obiettivo del gioco è impedire ai nemici di attraversare la mappa costruendo torri che, sparando autonomamente, colpiscono i nemici in avvicinamento.

Requisiti funzionali:

- Le torri potranno essere posizionate e spareranno autonomamente uccidendo i nemici presenti nel loro range all'interno della mappa in quel momento.
- I nemici si muovono in un senso seguendo il percorso nella mappa
- I nemici verrano generati ad ondate, l'utente avrà la facoltà di decidere quando incominciare una nuova ondata
- Sono presenti tre tipi di torri e due tipi di nemici
- Durante il corso della partita sarà visibile lo stato di avanzamento dell'ondata, vita rimasta, tempo trascorso, monete rimaste ecc.

Requisiti funzionali opzionali:

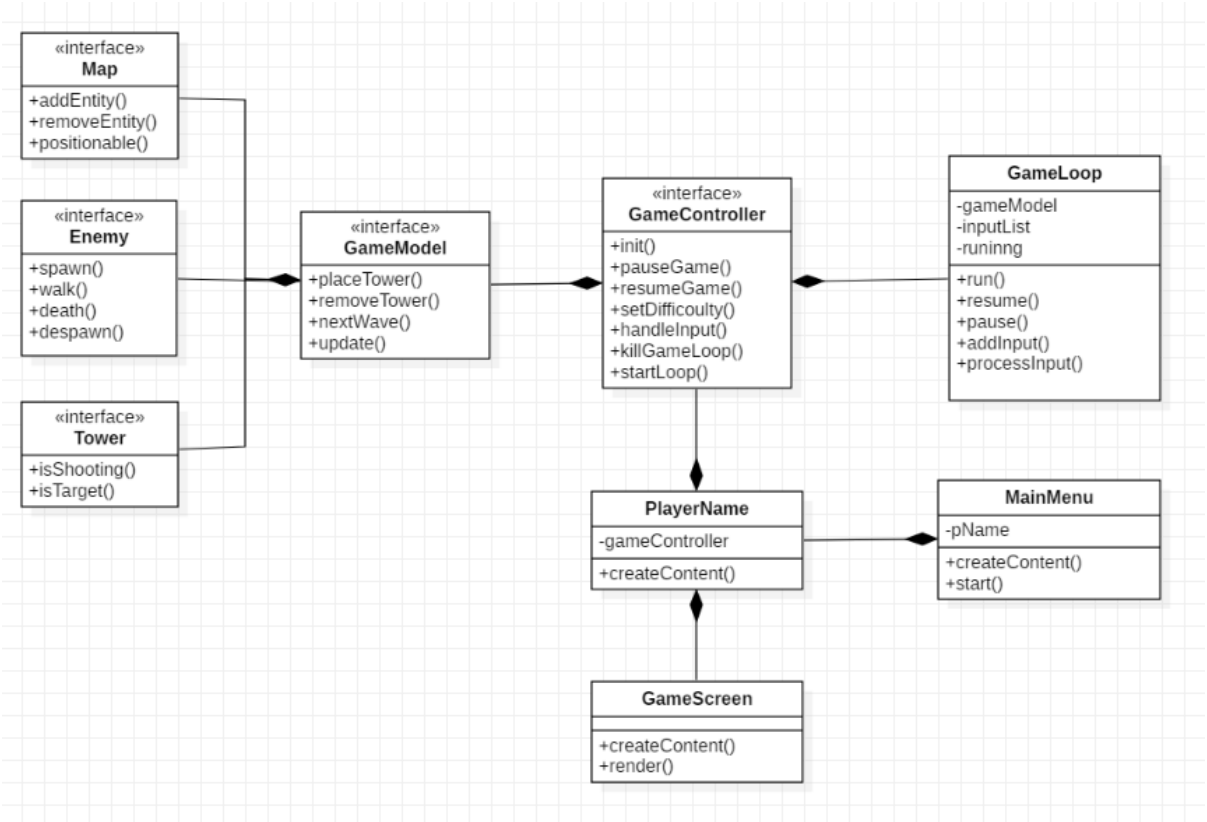
- Creazione di power up alle torri
- Creare più modalità di gioco (survior, arcade)
- Effetti sonori
- Sistema di save/load dalle impostazioni, salvataggio punteggi migliori e sfide
- Easter egg

Requisiti non funzionali opzionali:

- Raggiungere una soddisfacente fluidità di gioco
- Ottenere un'interfaccia il più possibile reattiva all'input dell'utente

1.2 Analisi e modello del dominio

Tower Jojo è un gioco strategico del sottogenere tower-defense, dove il giocatore ha lo scopo di fermare una serie di nemici che devono arrivare alla base che si sta difendendo.
Per infliggere danni ai nemici, l'utente ha la possibilità di posizionare torri; queste ultime sparano autonomamente in un determinato range e con una cadenza di fuoco che dipende dal tipo di torre.
Durante il corso della partita per ogni nemico ucciso e ondata completata vengono assegnate delle monete, con le quali comprare torri o potenziarle.
La vittoria è determinata se dopo una serie di ondate la base è ancora integra, in caso contrario si avrà una sconfitta.



Capitolo 2: Design

2.1 Architettura

Per la realizzazione di Tower Jojo si è scelto di utilizzare il pattern architetturale Model-View-Controller (MVC).
Ciò consente di isolare la logica funzionale che riguarda le tre componenti, le quali non interferiscono tra loro.
Così facendo il progetto è meglio organizzato e la modifica di una delle componenti non comporta cambiamenti nelle altre.
Nella nostra modellazione del pattern MVC gli input vengono catturati dalla view, la quale provvederà a notificarli al Controller e successivamente saranno gestiti comunicando opportunamente con il Model (vedi figura 2.0).

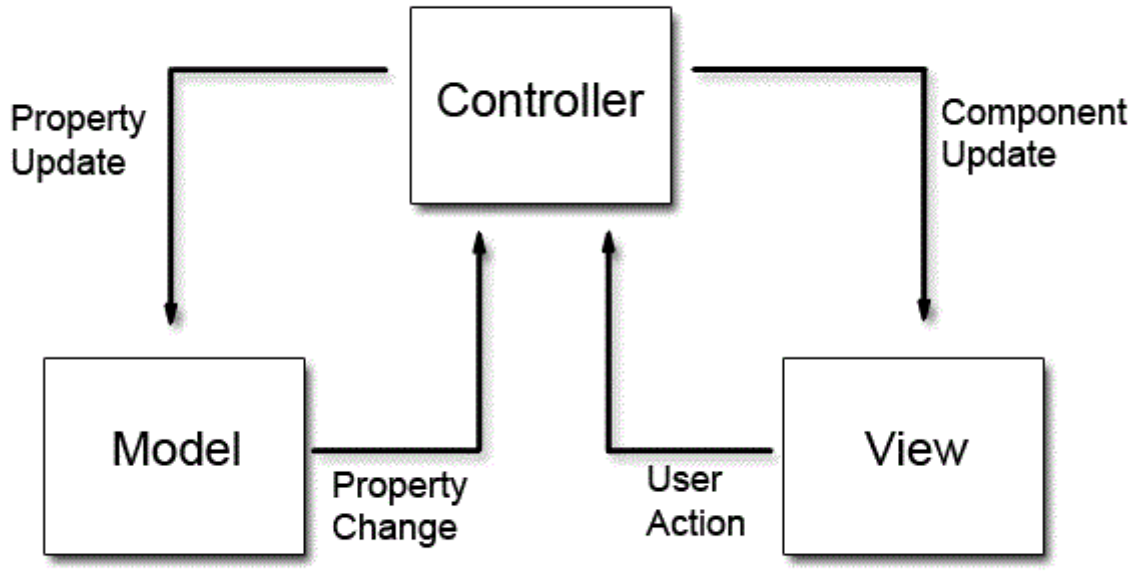


Figura 2.0: Schema sulla gestione del pattern MVC

Il model è il core dell'applicazione e ne contiene l'intera logica di funzionamento.
Dovrà dunque essere aggiornata costantemente tenendo conto anche degli input dell'utente.
Inoltre, il model è l'unica componente passiva dell'architettura e fornisce al suo esterno metodi minimali per l'interazione con i dati, il che garantisce a sua volta una sostituibilità veloce ed efficace della view.
Sarà infine compito della view aggiornare le componenti grafiche.
Nel main dell'applicazione viene lanciata la classe MainMenu che a sua volta istanzierà un GameScreen e un GameController.
Il GameController sucessivamente istanzierà un nuovo GameModel ed un GameLoop (vedi figura 2.1).

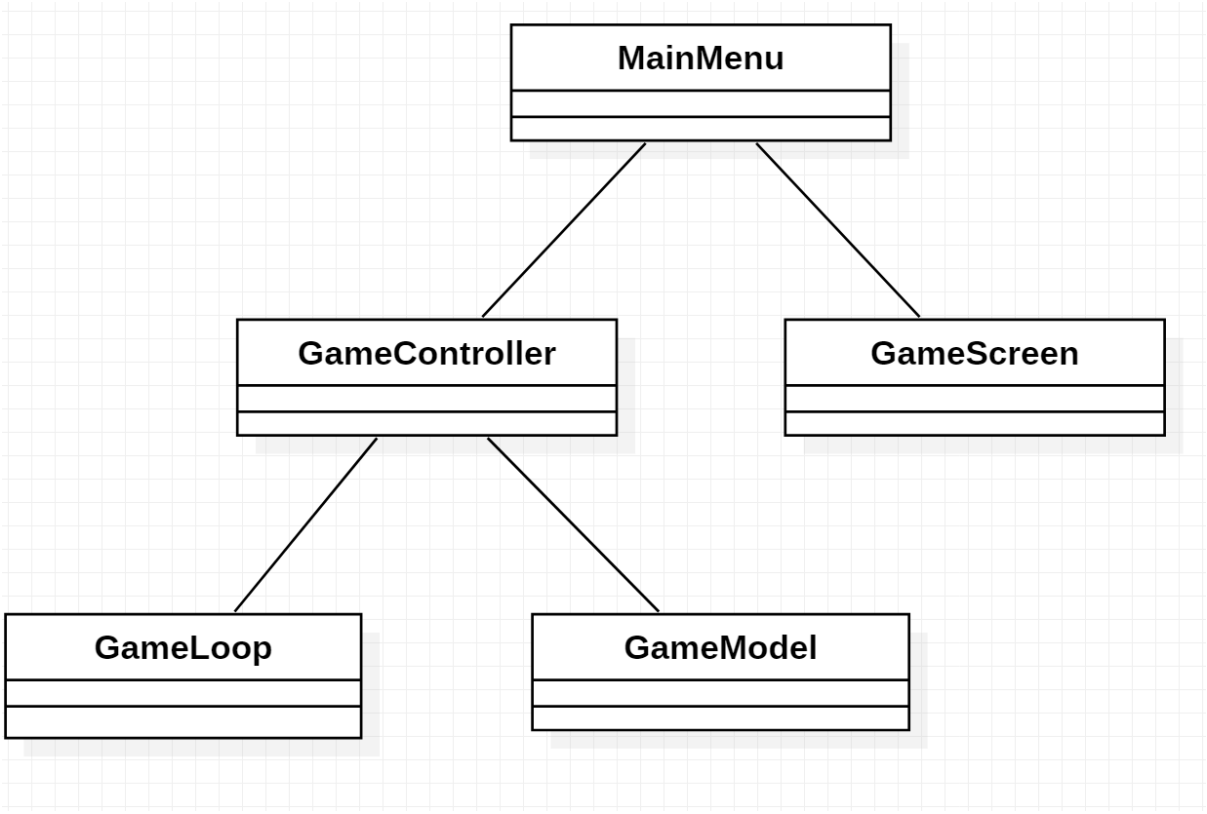


Figura 2.1: Schema UML dell'architettura dell'applicazione

2.2 Design dettagliato

Abu Ismam

La mia parte del progetto prevede la gestione del model della mappa e del player e una parte di view di gioco.

All'interno dell'applicazione Map svolge il compito di immagazzinare le informazioni relative all'andamento del gioco, quindi subirà aggiornamenti dal controller con una certa frequenza.
La griglia di gioco è composta da una serie di oggetti di tipo TileMap che corrisponde ad un quadretto della mappa dove è possibile fare azioni (vedi figura 2.2).
La AbstractMapModel al suo interno possiede come campi un array relativo alla griglia di gioco (composta da oggetti TileMapImpl), il percorso che i nemici devono seguire (TileMapImpl indicizzati seguendo un'ordine) e una lista di entità presenti all'interno di essa (entity può essere una torre o un nemico).
La lista di entità subirà frequentemente degli aggiornamenti in quanto durante il gioco è scontato che si posizionino torri o le si rimuovano.

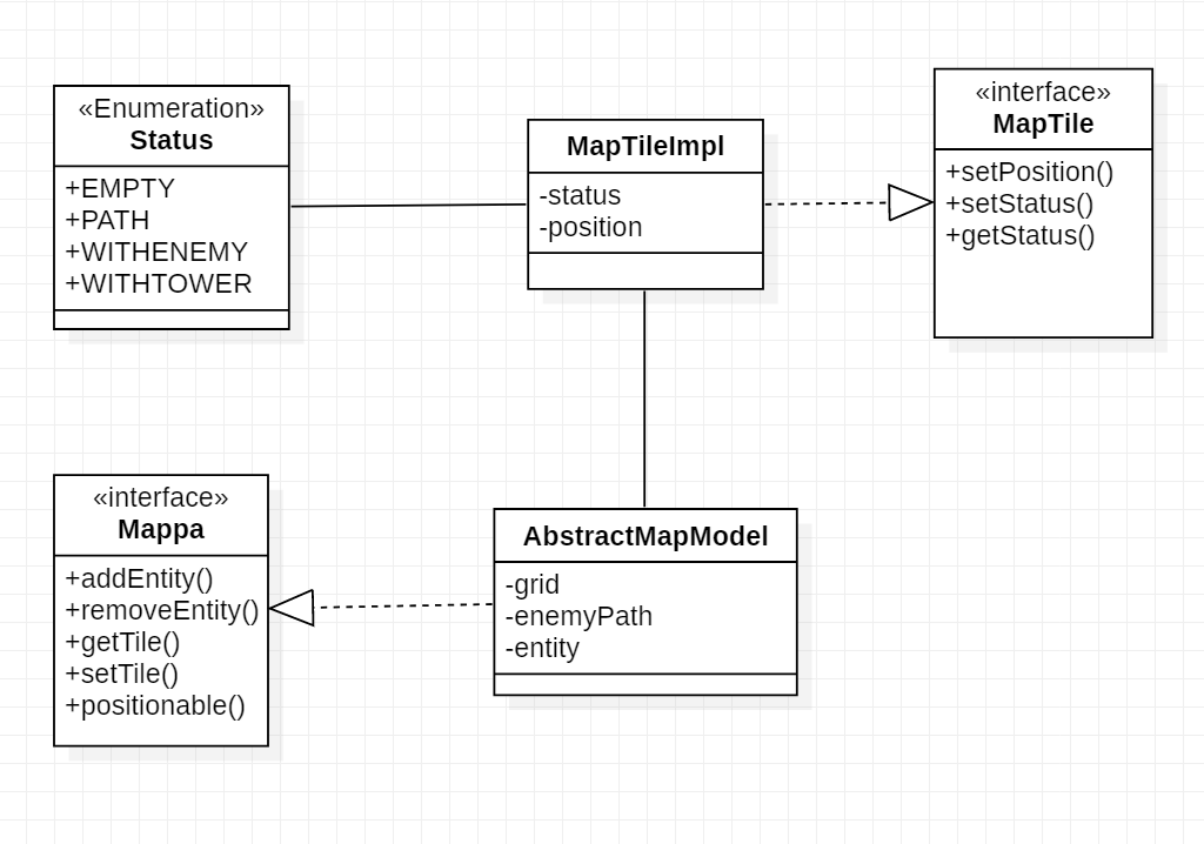


Figura 2.2.1: Schema UML dell'architettura del model della mappa

Il costruttore della mappa richiama due metodi, il primo (privato) per generare la griglia, il secondo (astratto) invece richiama un metodo astratto che genera il percorso.
La classe AbstractMapModel viene estesa da altre 3 classi che a loro volta contengono un metodo per generare il percorso, in base alla difficoltà (vedi Figura 2.3).
È stato utilizzato il pattern template method per generare un algoritmo diverso per ogni livello di difficoltà così da ottenere una fattorizzazione in classi e non dover scrivere più volte lo stesso codice.

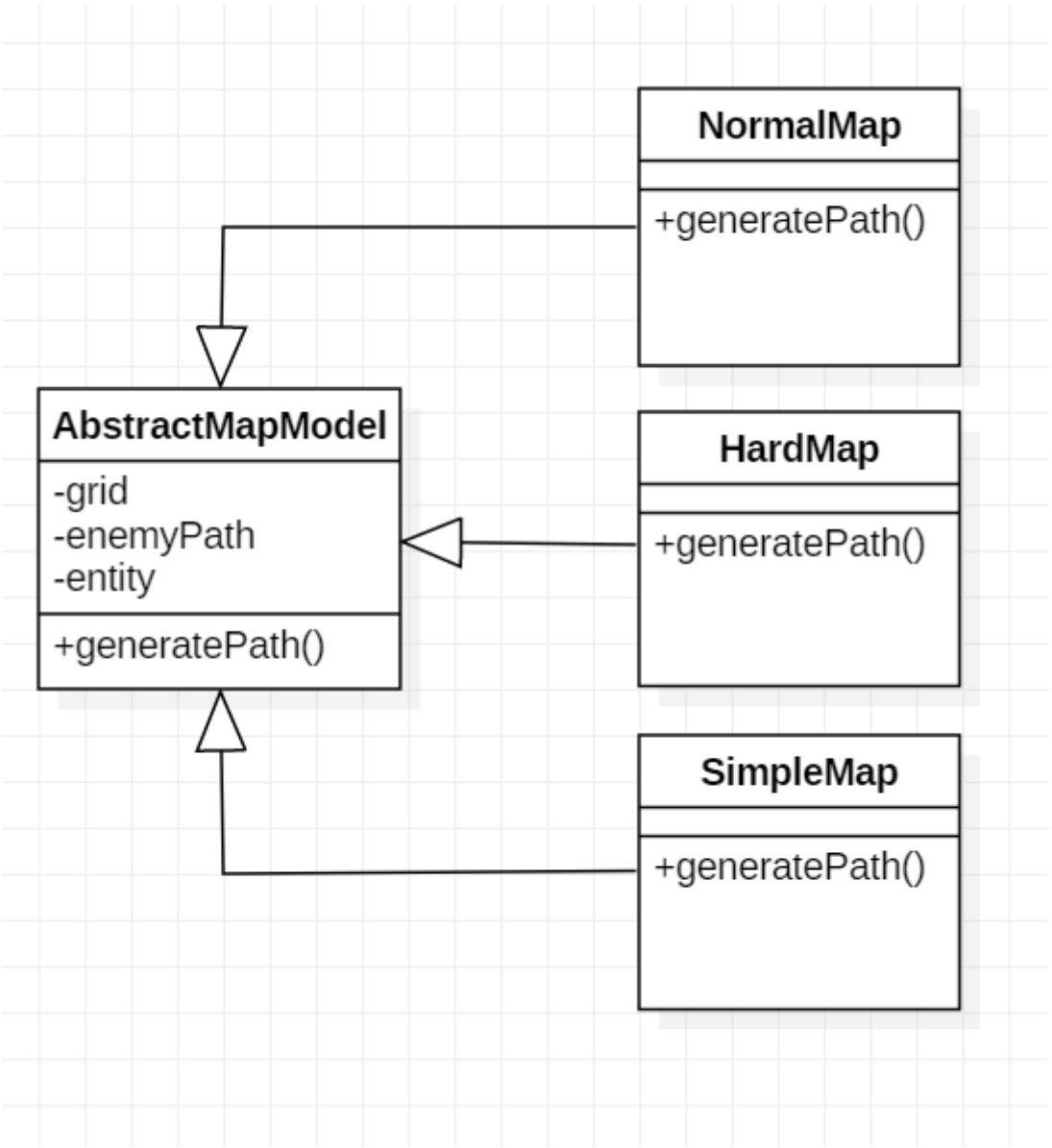


Figura 2.2.2: Schema UML dei percorsi della mappa

La classe PlayerImpl contiene informazioni utili riguardo il giocatore come: nome, monete, vita, danni, ondata corrente e ondate fatte.
Viene aggiornato dal controller ogniqualvolta venga ucciso un nemico aumentando le monete, venga superata un'ondata o un nemico raggiunga la base decrementando gli hp del giocatore (vedi figura 2.4).

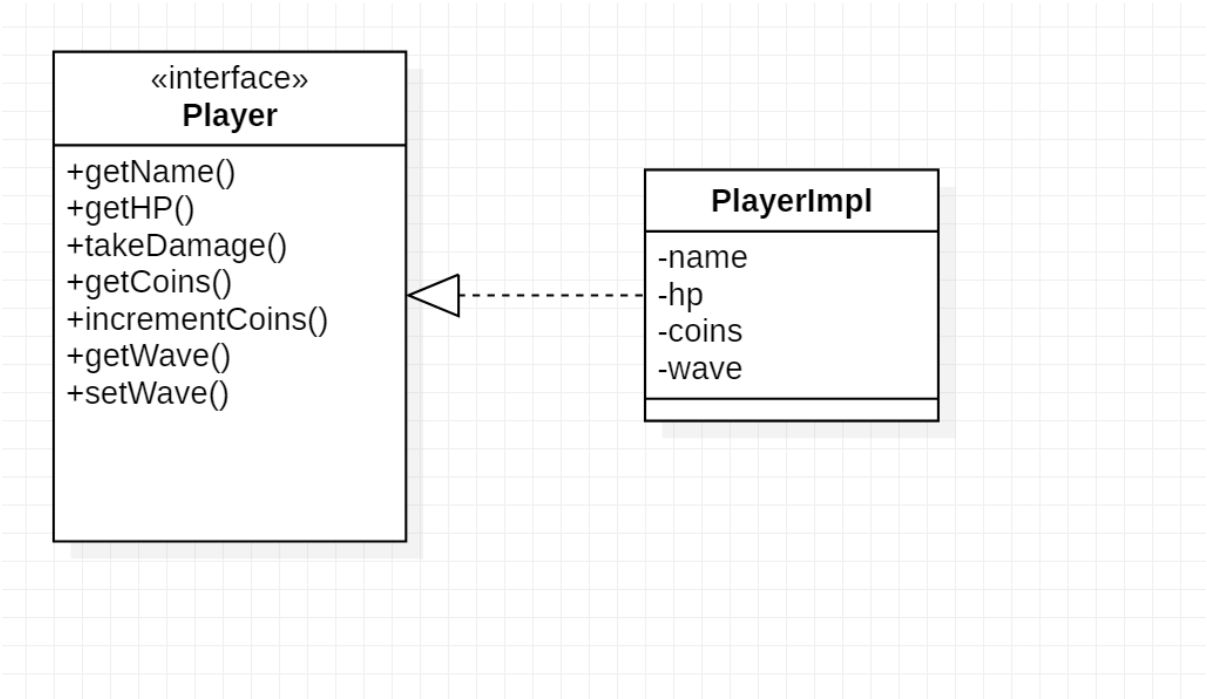


Figura 2.2.3: Schema UML del model del player

Per la gestione degli Input All'interno abbiamo l'enumerazione InputType, all'interno della quale sono definiti i due tipo di input che possiamo avere (figura 2.5).
Questa classe verrà richiamata per generare una lista di richieste di input che verranno prese in carico secondo la logica dettata dal controller.

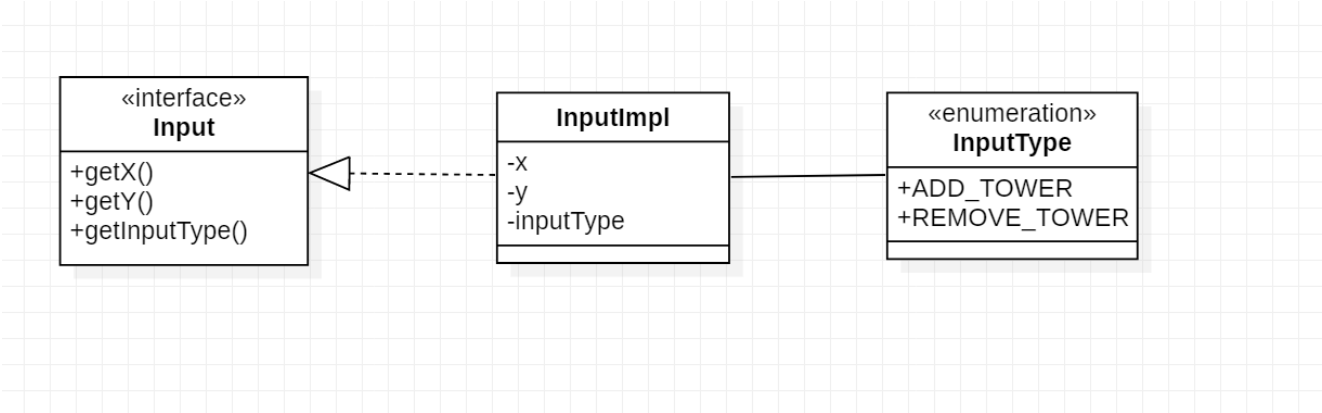


Figura 2.2.4: Schema UML dell'input.

Hu Marco

La mia parte prevede l'implementazione del GameModel, Controller e GameLoop del gioco.

Il GameModel è la classe che coordina i vari model e definisce la logica e il funzionamento del gioco (vedi figura 2.2.5).
Esso infatti contiene la mappa, il player, l'ondata corrente e lo status del gioco.
È dotato di metodi come PlaceTower, RemoveTower, nextWave e getStatus (che permette al gioco di funzionare e controllare lo stato).

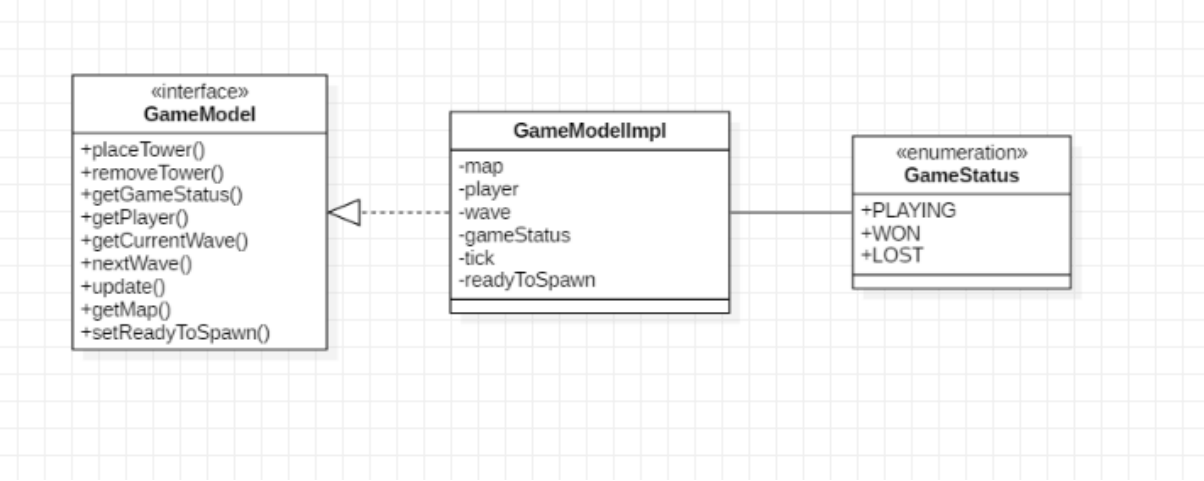


Figura 2.2.5: Schema UML del GameModel

Poiché in runtime il gioco deve conoscere e gestire alcune azioni eseguite dai singoli modelli, si è pensato di utilizzare il pattern observer.
Ogni volta che gli observable (torre e nemici) vorranno comunicare con il GameModel (observer) chiameranno il metodo notifyObservers().
In risposta a questa chiamata, l'Observer (GameModel) eseguirà una certa azione update (Observable subject), diverso dall'update() che serve per mandare avanti il gioco di un tick.
Quindi abbiamo Observable per le torri e due per i nemici, uno per quando muore arrivando a fine percorso e uno quando muore per via di una torre.

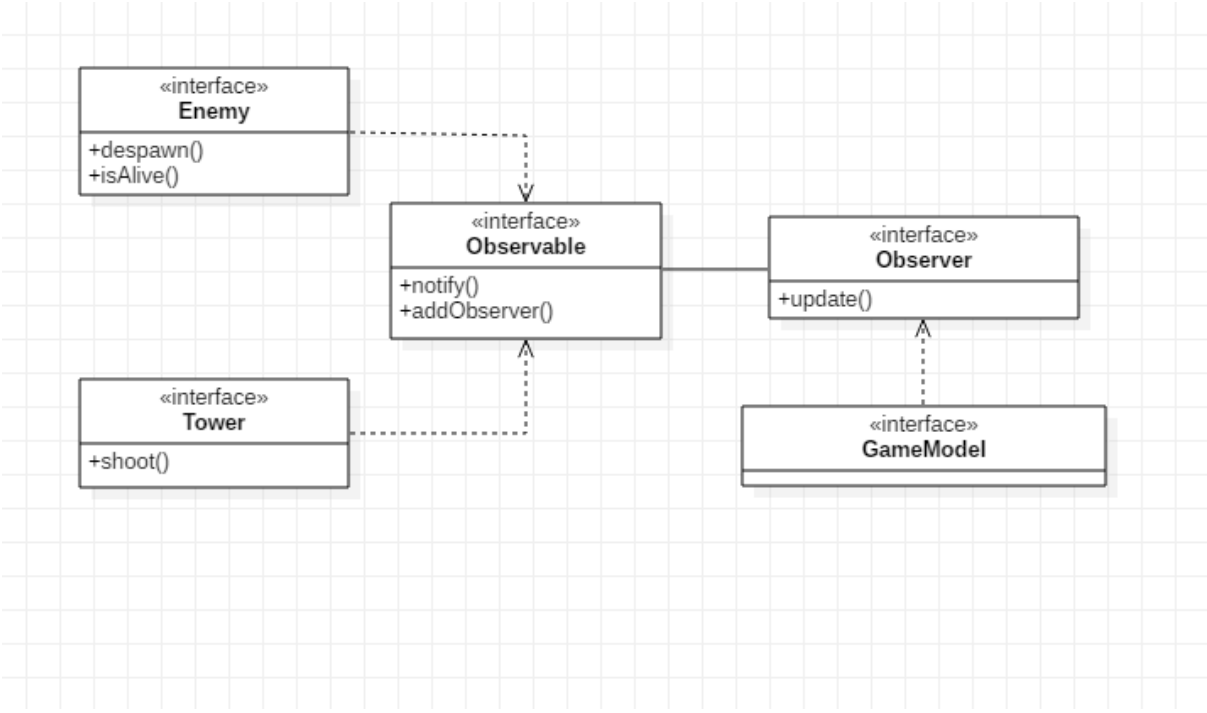


Figura 2.2.6: Schema UML del pattern Observer

Il controller (vedi figura 2.2.7) si occupa di gestire il gameloop, in particolare:

- iniziare/terminare il gioco (StartGame(), killGameLoop());
- mettere in pausa/ riprendere (pauseGame(), resumeGame());
- gestire gli input (handleInput);

Esso infatti funziona da 'ponte' tra Model e View.

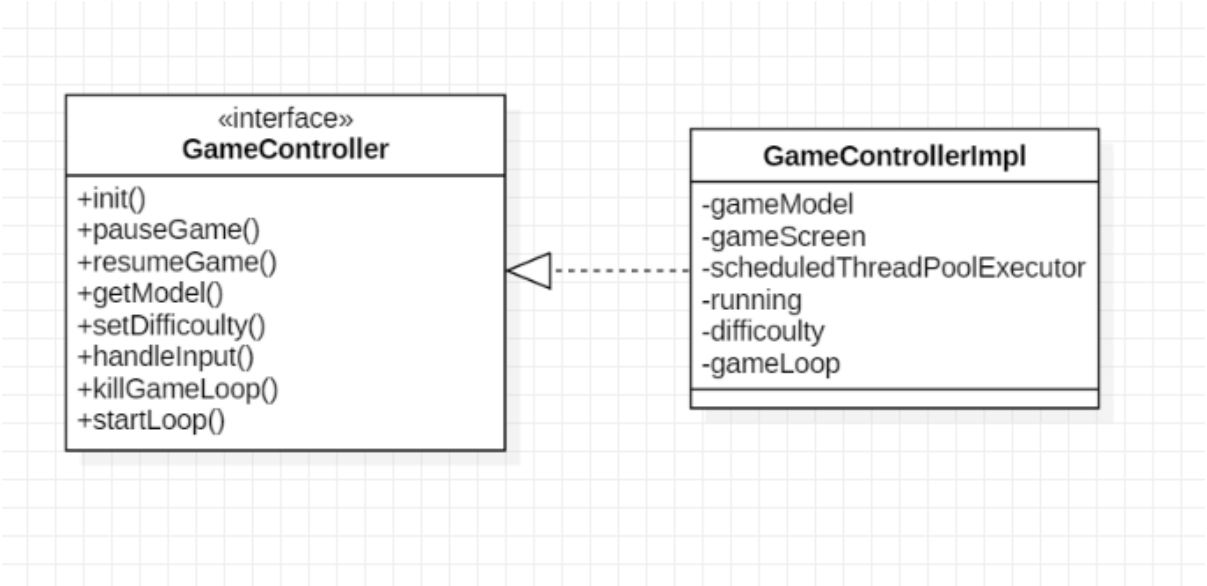


Figura 2.2.7: Schema UML del GameController

Il gameloop (vedi figura 2.2.7) si occupa di eseguire periodicamente (ogni tick del gioco) le seguenti azioni:

- inputProcess, che legge e traduce gli input utente istruendo il model;

- update, aggiorna le entità del gioco;
- render, disegna le entità del gioco sulla GUI permettendo all'utente di visualizzarle.

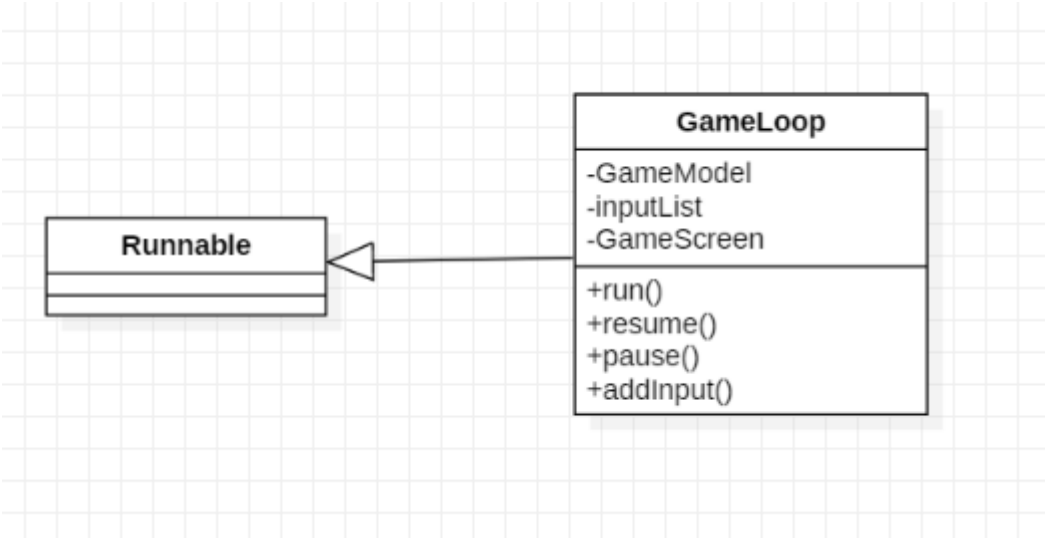


Figura 2.2.8: Schema UML del GameLoop

Mattiussi Vlad

La mia parte all'interno dell'applicazione prevede la gestione del model dei nemici, delle ondate e di una parte della view e parte del controller.

Il nemico è un'entità del gioco e la sua funzione principale è di arrivare alla fine di un percorso seguendo una via prestabilita.
Il nemico è caratterizzato da un numero di punti ferita e una velocità di movimento.
Quando i punti ferita raggiungono lo zero (morte per mano di una torre) il nemico "rilascia" una certa quantità di moneta del gioco, la quale servirà al giocatore per comprare e potenziare le torri.
Per permettere a un nemico di seguire un percorso si è utilizzato un array contenente tutte le posizioni del percorso che devono essere raggiunte.
È possibile creare più tipi di nemici (modificando i diversi parametri di vita, velocità, ecc.).

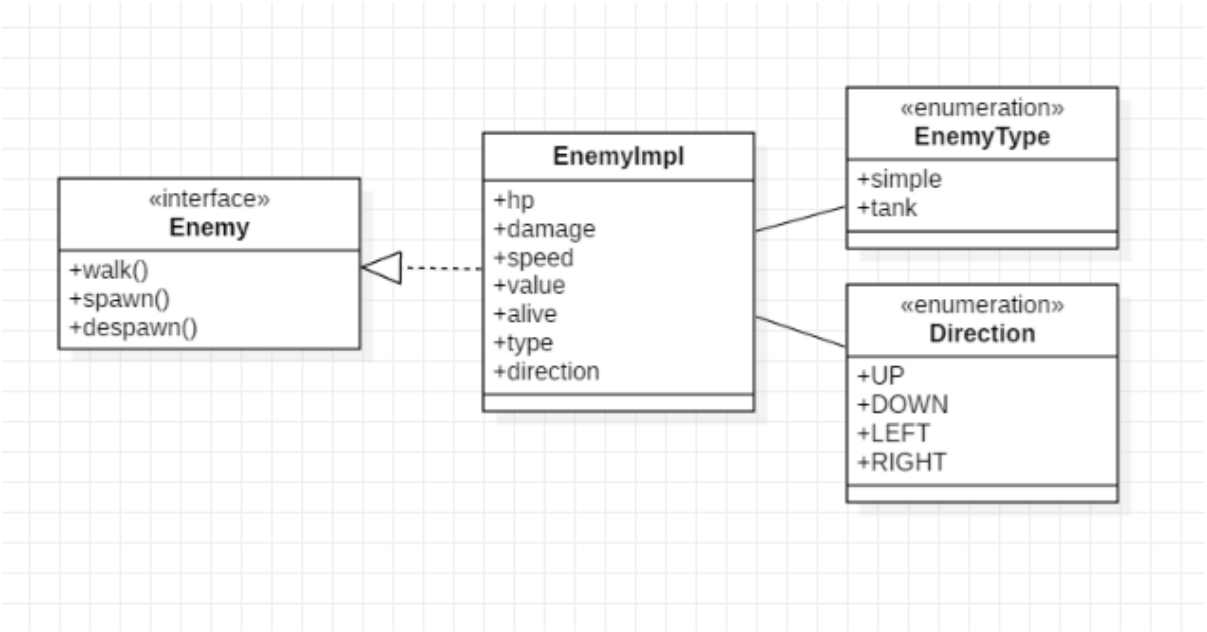


Figura 2.2.9: Schema UML della gestione dei nemici

Ogni ondata (wave) contiene un array di nemici pronti per essere “spawnati” nella mappa di gioco. Il costruttore è formato da un int e un enum “type” per sapere la quantità e il tipo di nemico da generare. Ogni ondata contiene un metodo per creare e passare alla prossima wave.

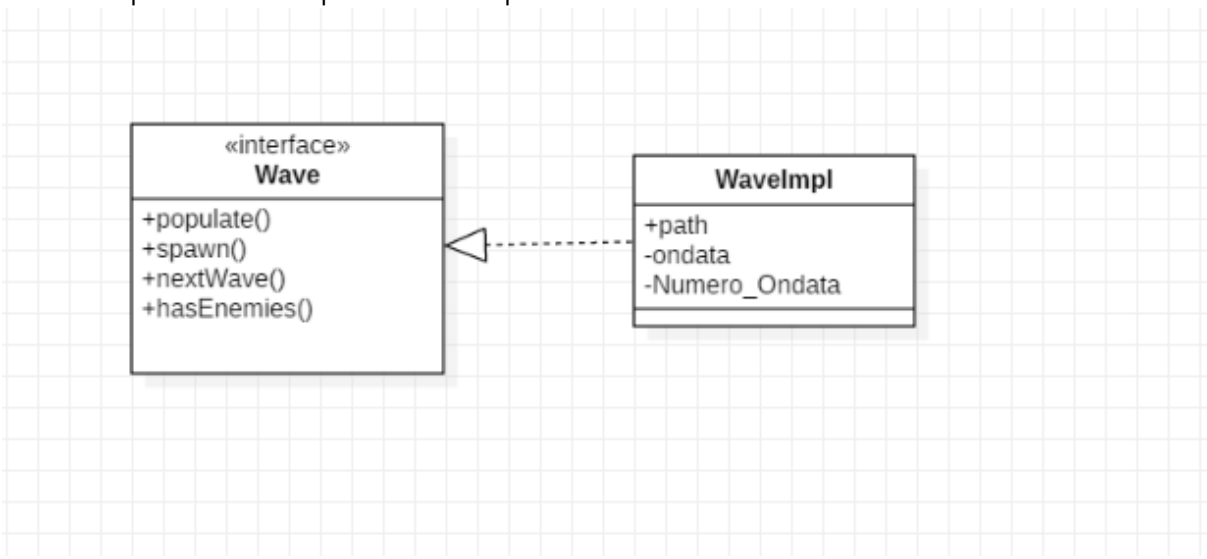


Figura 2.2.10: Schema UML della gestione delle Wave

Zhu Zheliang

La mia parte consiste nell'implementazione del model delle torri, del proiettile e le collisioni.
Sia le torri sia i proiettili fanno parte dell'entità del gioco.
Le torri sono dotate di un range (quadrato) per mirare, il loro obiettivo è sparare al primo nemico presente nel range e passare al successivo quando quello corrente sia morto o uscito dal range (vedi Figura 2.2.11).
Ci sono 3 tipi di torri, ognuno di essi con un certo range; infliggono tipo di danni diversi ai nemici, hanno un costo diverso e questi sono differenziati anche dall'enumerazione TowerType.
Le torri possono essere collocate in un punto qualsiasi della mappa, tranne che sul percorso dei nemici.
La parte essenziale dell'implementazione del model è verificare la presenza dei nemici nel range, dove ho utilizzato due ArrayList per salvare i Tile della griglia della mappa.
Il range di ogni torre è stata trovata tramite il metodo setRange .
Con findEnemy si trovano i nemici dentro un certo range.
Infine, si chiama il metodo shoot per sparare.
Il costruttore della classe di proiettile è formato da un Pair per la posizione del proiettile, un nemico e i danni creati.

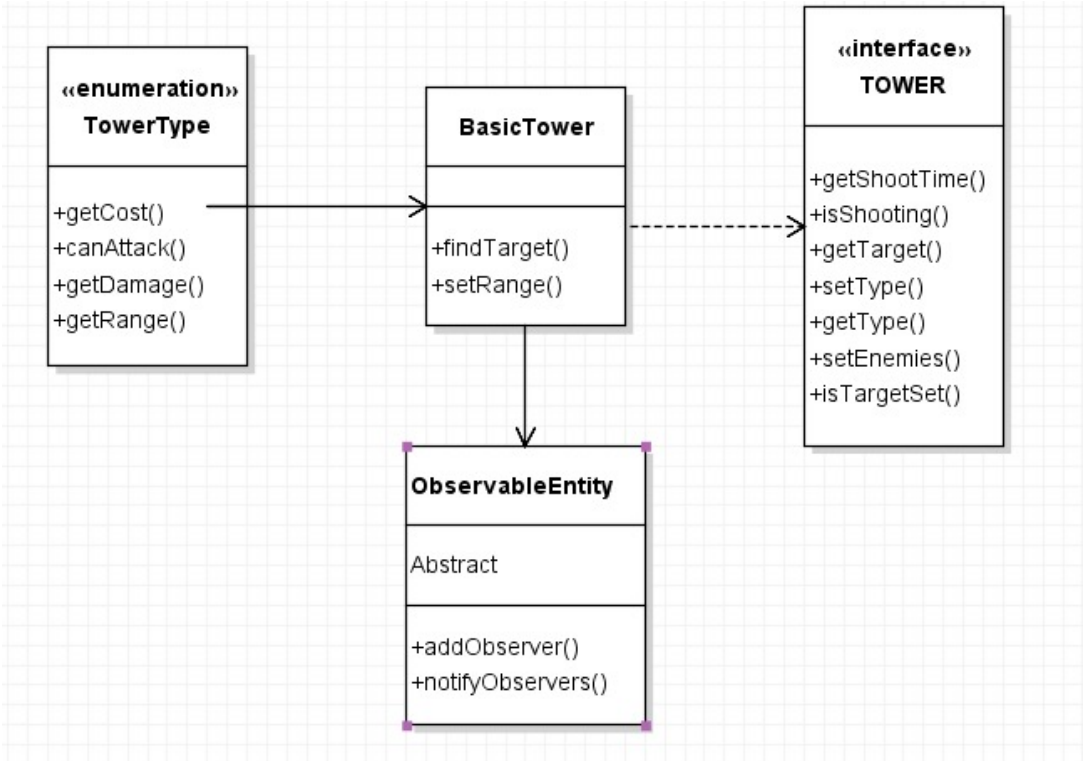


Figura 2.2.11: Schema UML della gestione del model delle torri

È stata creata una variabile “tick” utilizzata nel metodo update per fare collidere il proiettile con il nemico creando danni resettandosi dopo.
Per facilitare e avvisare la mappa la presenza del proiettile si è utilizzato un’interfaccia Observer e una classe astratta Observarble che viene estesa dalla classe BasicTower.

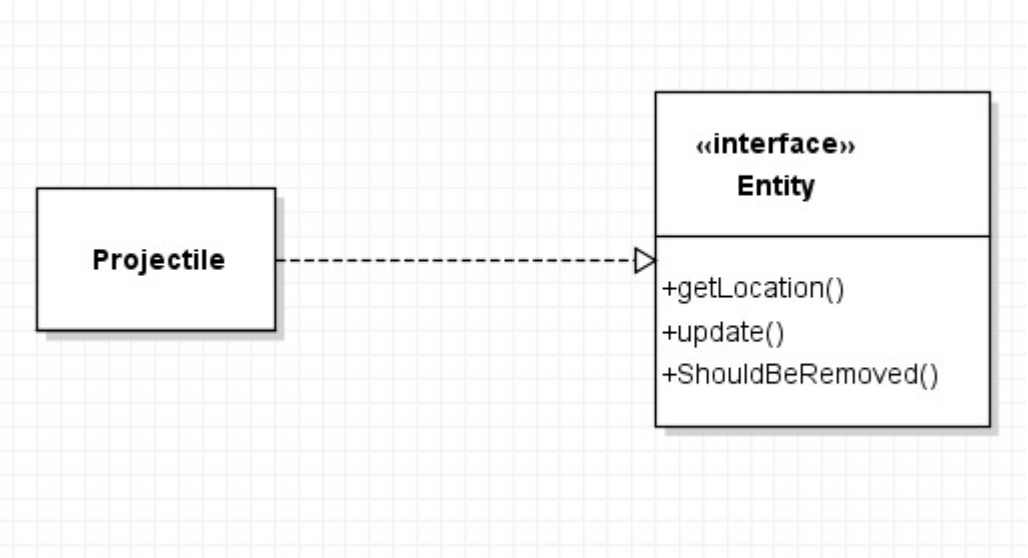


Figura 2.2.12: Schema UML dei proiettili

Abu e Mattiussi

Per realizzare le componenti grafiche dell’applicazione si è deciso di utilizzare JavaFx, una libreria grafica molto versatile resa disponibile dalla Oracle.
Per le dimensioni della finestra di gioco, si è deciso di stabilire un’unità di misura tramite cui si dimensionano tutti i vari bottoni, finestre, griglie.
Le costanti per definire l’unità di misura si trovano nella classe GameConstants.
La scelta di non realizzare il gioco full screen è motivata dal fatto
che in questo modo l'esperienza di gioco è sia piacevole, sia adattabile ai vari ambienti e situazioni di lavoro (nel caso l'utente avesse la necessità di mantenere varie applicazioni attive nello schermo).

Per gestire il MainMenu (figura 2.2.13) si è sfruttata la classe di JavaFX Application.
Tramite questo menu si possono scegliere le difficoltà di gioco, aumentare/abbassare il volume.
Nel costruttore è presente un Pane che contiene al suo interno le varie interfacce.
Abbiamo una serie di schermate per ogni menu:

- Menu principale, contiene i classici bottoni start, option ed exit
- Menu Option, contiene i bottoni back, sound e difficulty
- Menu Sound, contiene I bottoni back, low, medium e high, da questo menu si gestisce il volume di gioco
- Menu Difficulty, contiene I bottony back, simple, medium e hard, da questo menu è possibile cambiare la difficoltà in base alla mappa

Ogni menu/bottoni ha animazioni e sfondi personalizzati realizzati tramite classi apposite di JavaFx.

Per gestire le immagini si richiama la classe LayoutImages, che contiene al suo interno tutte le immagini delle varie schermate.
Questa classe a sua volta per caricare immagini richiama la classe ImageLoader, che carica dalla cartella res le immagini richieste.
Le classi MenuButton e VolumeButton realizzano bottoni con funzioni specifiche.
La classe Difficulty crea una nuova finestra per avvisare del cambio di difficoltá.
La classe PlayerName è una scena di mezzo tra lo start del MainMenu e il GameScreen, qui viene istanziato sia il GameController che il GameScreen.

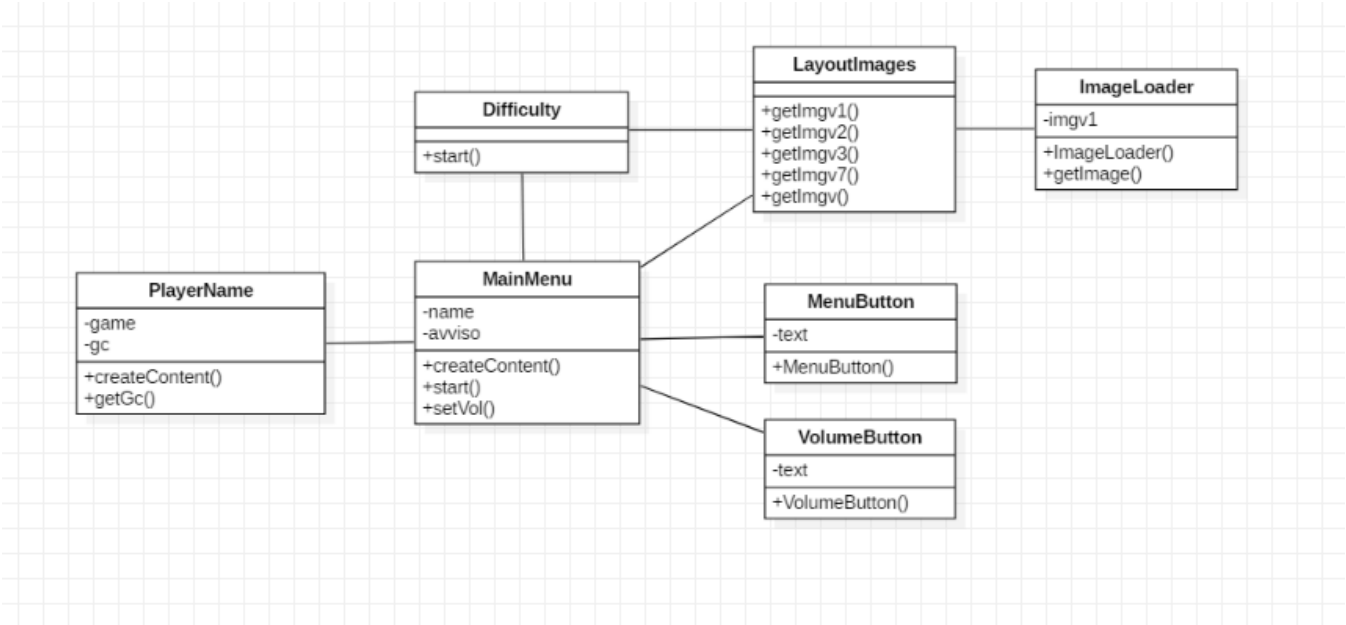


Figura 2.2.13: Schema UML delle schermate del MainMenu

Il GameScreen è l'interfaccia di gioco dell'applicazione (vedi Figura 2.2.14).
L'utente, quindi, quando giocherà interagirà direttamente tramite essa.
Nel costruttore è presente un Pane principale composto da una FlowPane (a destra) che contiene la sidebar e un GridPane (a sinistra), che contiene la griglia di gioco.

Il FlowPane si suddivide ancora in:

- HBox per i bottoni start wave, pause e resume
- VBox per le informazioni riguardanti l'andamento del gioco come hp, coins, wave e nome giocatore
- FlowPane per il menu della selezione delle torri disponibili

Il metodo render serve ad aggiornare la sidebar e disegnare gli oggetti presenti sulla mappa (nemici e proiettili, nota bene, non le torri).

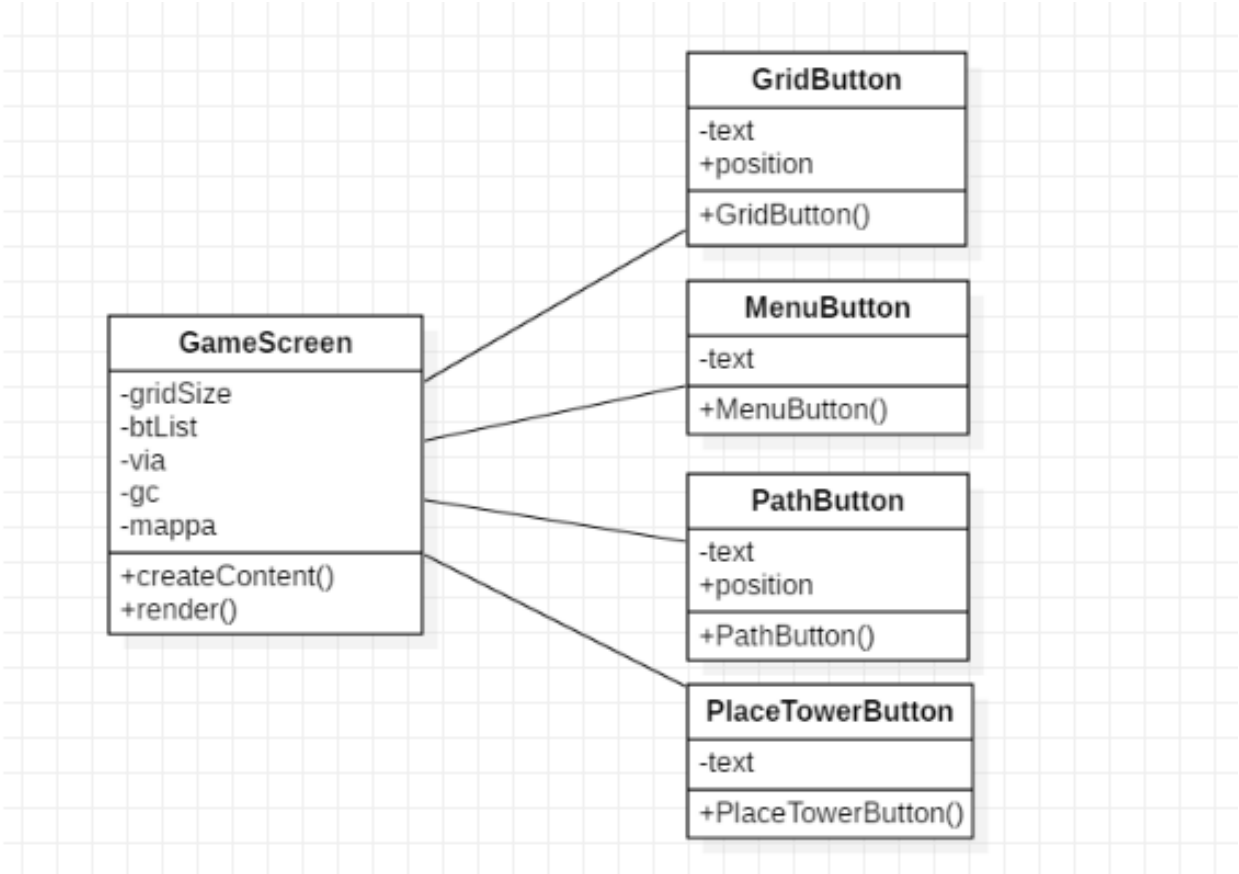


Figura 2.2.14: Schema UML del GameScreen.

Capitolo 3: Sviluppo

3.1 Testing automatizzato

Durante la fase di sviluppo dell'applicazione si è deciso di avvalerci della libreria JUnit per testare in modo automatizzato alcune funzionalità di base.
Altre invece, data la loro impossibilità ad essere testate automaticamente, sono state verificate manualmente.
Esponiamo ora le componenti del progetto testate automaticamente.

Model:

- Mappa: Test sulle diverse funzionalità della mappa quali creazione percorso, posizionamento torre, interrogazioni varie
- Nemico: Test sulle funzioni dei nemici, creazione, rimozione e movimentazione
- Torre: Test sulla creazione dell'oggetto, sulle sue coordinate, se stia sparando o meno, verificare il rateo di fuoco
- GameModel: Test sulla logica di funzionamento, posizionamento e rimozione torre, spawn enmmy test, go to next wave, enemy walk test, game win and lost condition

Tra le funzionalità che sono invece state testate manualmente abbiamo:

- View: Tutta la parte grafica del gioco è stata testata visivamente avvalendosi di più monitor con risoluzioni e grandezze diverse, la finestra di gioco si adatta alla risoluzione del monitor sulla quale si sta giocando
- Audio: Corretta attivazione/disattivazione dell'audio a seguito di input dell'utente nel menu delle opzioni e durante la partita
- Input: L'effettiva risposta dell'applicazione ai vari input non presenta bug

3.2 Metologia di lavoro

A seguito della fase di sviluppo, possiamo asserire che la ripartizione dei compiti da noi concordata e confermata nelle fasi precedenti è stata parzialmente rispettata, siccome la ripartizione non era particolarmente bilanciata abbiamo dovuto apportare leggere modifiche.
Di seguito elenchiamo le funzionalità implementate dai singoli elementi del team:

- Abu Ismam: I model della mappa e del player, gestione degli input, gestione schermate grafiche
- Hu Marco: GameController, GameModel
- Mattiussi Vlad: I model di Enemy e Wave, gestione schermate grafiche
- Zhu Zheliang: I model di torre e Projectile

Altre parti sono invece state sviluppate da più componenti del team.
View.GameView e View.MainMenu, i package che rappresentano graficamente le finestre del gioco vero e proprio, sono state realizzate da Abu Ismam e Vlad Mattiussi.
Per il collegamento tra le parti dell'applicazione tramite classi del controller hanno collaborato Hu Marco e Vlad Mattiussi.

Si è cercato di fare buon uso del DVCS: il lavoro è stato articolato su più branch, ognuno con il compito di realizzare un package.
Il branch master principale è stato utilizzato principalmente come punto di sincronizzazione fra le parti.

3.3 Note di sviluppo

3.3.1 Abu Ismam

Utilizzate occasionalmente lambda durante la scrittura delle classi del model della mappa, questa funzionalità del linguaggio sono state utili per eseguire operazioni sulle entità presenti nella mappa stessa.
Utilizzato il template method per la gestione di una parte del model della mappa
Per la parte che concerne la view ho utilizzato la libreria JavaFx che non conoscevo, dunque ho dovuto usare spesso la documentazione fornita da Oracle per comprenderne il funzionamento e l'utilizzo delle varie componenti adoperate.

3.3.2 Hu Marco

Durante l'implementazione del model si è cercato di mantenere chiaro e semplice il codice facendo uso di nomi di variabili semplici ma significative.
Utilizzo di lambda e stream per filtrare oggetti, ove era necessario.
Per implementare il gameloop ho fatto delle ricerche online per capirne il funzionamento e alla fine ho deciso di usare delle librerie di java util, in particolare dei ScheduledThreadPoolExecutor che permette a un thread di eseguire cicli ogni tot tempo fissato.

3.3.3 Mattiussi Vlad

Ho utilizzato JavaFx, una libreria che non conoscevo, dunque ho dovuto consultare spesso la documentazione per capire il funzionamento di vari componenti utilizzati.
Ho sfruttato le lambda per velocizzare l'implementazione di alcune parti di codice
Per la fase implementativa, per ridurre il lag durante il rendering ho caricato le immagini della mappa, del percorso, dei nemici e delle torri una solva volta e usate come campi nella mia classe.

3.3.4 Zhu Zheliang

Ho utilizzato lambda per fare un filtro dei nemici dalle entità e trovare un target per la torre.
È stato utilizzato anche un'interfaccia Observer e una classe astratta ObservableEntity per notificare altre entità.

Capitolo 4: Commenti finali

4.1 Autovalutazione e lavori futuri

Abu Ismam

È stata una bella esperienza realizzare un gioco partendo dalle sole conoscenze forniteci, sono parzialmente soddisfatto del risultato.
Sono convinto si possa ancora migliorare su molti aspetti, soprattutto implementativi.
Mi sono trovato bene con gli altri compagni, abbiamo superato vari momenti di difficoltà soprattutto in fase di analisi dovuti alla nostra poca (se non nulla) esperienza in game programming, ma con calma e collaborazione siamo sempre riusciti a cavarcela.
La collaborazione durante la fase di analisi è stata di vitale importanza.
Ho migliorato la mia capacità di programmazione, principalmente grazie ai design patterns dei quali, prima di questo corso, ne ignoravo l'esistenza, e in generale di tutto ciò che concerne un codice di più alta qualità.
A causa del poco tempo a disposizione non credo di voler portare avanti il progetto.
Sicuramente è stata un'ottima esperienza che mi sarà d'aiuto per progetti futuri.

Hu Marco

Posso ritenermi soddisfatto del progetto che abbiamo realizzato.
Il mio ruolo nel gruppo è stato quello di coordinare il lavoro comune.
Lo sviluppo della mia parte non richiedeva particolari tecniche implementative e/o algoritmi complessi, per questo ho provato ad utilizzare design pattern e cercando sempre di organizzare il codice facendo del mio meglio.
Riconosco che alcune parti avrei potuto progettarle meglio anche se ho sempre cercato di rendere più comodo l'utilizzo delle mie interfacce ai miei colleghi tramite l'utilizzo di pattern di progettazione.

Mattiussi Vlad

Complessivamente mi ritengo abbastanza soddisfatto del lavoro svolto finora.
È stata un'esperienza che ha ampliato le mie conoscenze nel campo della programmazione java; in particolare ho appreso la varietà delle librerie di javafx con le quali ho fatto una parte di grafica.
Inoltre, è stato interessante poter lavorare in gruppo per confrontarci e aiutarci l'uno con l'altro.
Riconosco che in alcune parti ho scritto del codice non proprio ottimale, ma l'importante e capire i propri errori per non ripeterli.

Zhu Zheliang

È stata un'esperienza molto utile per me nell'approfondimento del linguaggio java e mi sono trovato bene con il gruppo, ognuno di noi è sempre disponibile per aiutare gli altri e non solamente concentrarsi sulla propria parte.
Mi ritengo abbastanza soddisfatto del progetto anche se poteva essere migliorato, ma ha alzato la mia capacità di programmare in java.
Durante l'implementazione della mia parte ho avuto difficoltà e grazie all'aiuto dei miei compagni sono state superate.
Molti aspetti del codice potevano essere migliorati, ma a causa del tempo e altri problemi ho cercato di renderla più semplice.

4.2 Difficoltà incontrate e commenti per i docenti

Tutti abbiamo avvertito una certa confusione nel disegnare in modo accurato i class diagram UML, visto che non conoscevamo a fondo la sintassi del linguaggio. Potrebbe essere utile, a nostro avviso, fornire agli studenti maggiori dettagli a riguardo per facilitare loro la modellazione grafica.
È stato complesso anche strutturare bene la gestione degli svariati elementi grafici di una schermata e mantenerne l'allineamento in tutte le tipologie di schermo utilizzate.
Riscontrati problemi anche sulla gestione delle risorse che devono essere accedute molto frequentemente come getEntityList nella classe AbstractMapModel per il quale abbiamo avuto bisogno di creare una copia momentanea.

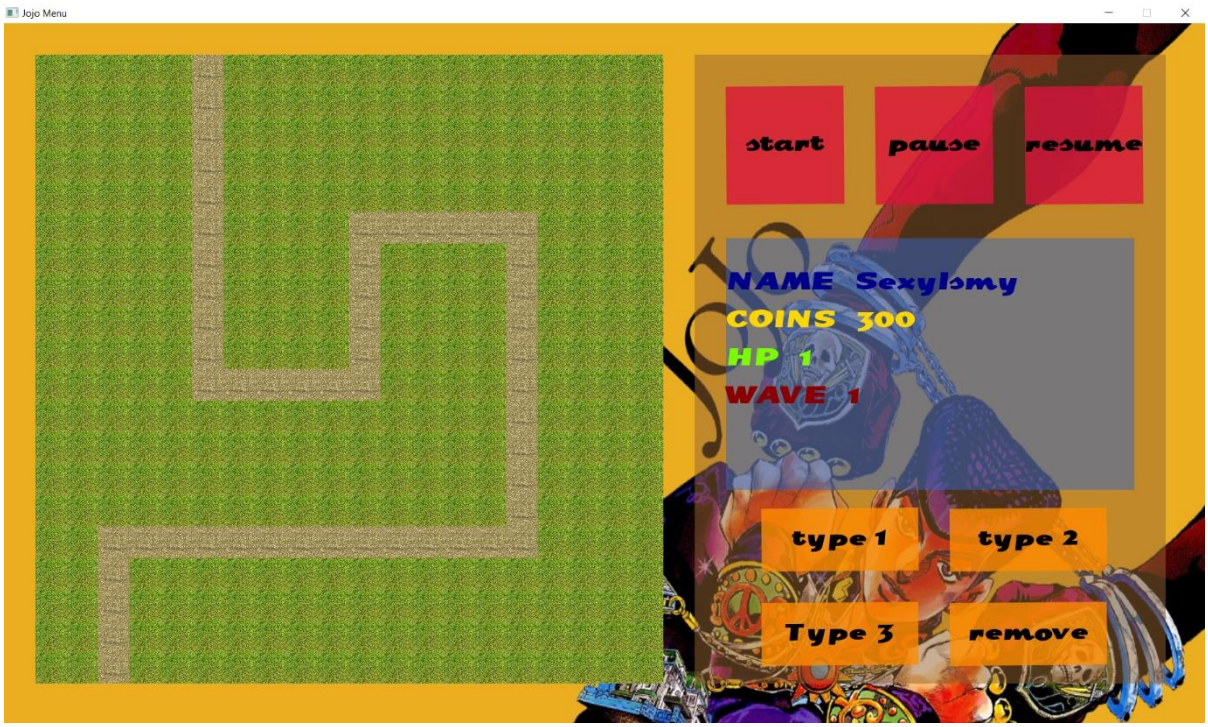
Capitolo 5: Guida utente

Appena avviata l'applicazione si aprirà il menu principale dalla quale si potrà scegliere tra le seguenti opzioni:

- New game: per iniziare una nuova partita
    - Segue la schermata in cui si inserisce il nome del giocatore e si inizia effettivamente a giocare premendo start.
- Settings: in questa schermata è possibile regolare a proprio piacimento le impostazioni di gioco.
    - Sound: questa schermata permette di modificare il volume in gioco
    - Difficoulty: Permette di selezionare la difficoltà di gioco, ci sono 3 livelli di difficoltà
- Exit: per uscire dall'applicazione

Scena iniziale dell'applicazione

Una volta avviata la partita, la schermata di gioco si presente come in figura.



Inizialmente non ci saranno nemici e si avrà il tempo di posizionare un certo numero di torri nella mappa.
Per iniziare un'ondata è necessario premere il bottone start wave, questo fa comparire una serie di nemici che dipendono dal numero dell'ondata, più si va avanti più ne aumenta il numero.
Attenzione alla mappa semplice, ai nemici piace muoversi senza logica teletrasportandosi.
L'obiettivo del giocatore è impedire ai nemici di attraversare la mappa per arrivare alla base.
I bottoni pause e resume servono per fermare momentaneamente il gioco, in questo lasso di tempo però si possono posizionare torri e rimuoverle.
Prima di eseguire l'applicazione consigliamo vivamente di modificare il ridimensionamento e layout dello schermo intorno al 100% in qualunque risoluzione (a maggior ragione a risoluzioni come e oltre il 2160p) per godere di una buona visuale di gioco.
Quando si inizia il gioco è fortemente consigliato posizionare attentamente le torri, in maniera che riescano a colpire i nemici nella maniera più efficiente possibile.
Inoltre è consigliabile evitare di posizionare torri durante il corso dell'ondata anche se effettivamente possibile e funzionante.