RELAZIONE PROGETTO OOP

KILL TO SURVIVE

BESHIRI REI 789973
CRISTUREAN DENIS 800957
VAIENTI ANDREA 789719
VINCENZI MATTIA 792970

SOMMARIO

L’idea di realizzare questo gioco è nata dall’incontro di opinioni di ciascun membro del gruppo. In comune accordo abbiamo deciso di ricreare una versione semplificata della modalità “**Greed**” del gioco “**The Binding of Isaac**”, un gioco rougelike 2D topdown, pubblicato da Nicalis.

INDICE

Capitolo 1

ANALISI

Il gioco mira a riprodurre la modalità "**Greed**" dell'originale **The Binding of Isaac**. Quest'ultima è organizzata a livelli, detti **round**, che diverranno via via sempre più difficili. Con round si intende un'insieme di nemici che compaiono contemporaneamente e che sono da sconfiggere nel minor tempo possibile. Naturalmente la nostra versione sarà molto semplificata rispetto all'originale e con personali adeguamenti in base alle necessità.

1.1 Analisi dei requisiti funzionali

Requisiti funzionali obbligatori

- Lo scopo del gioco, nella sua versione minimale, sarà quello di terminarlo nel minor tempo possibile, tentando di guadagnare il maggior numero di **punti** possibili. In questa modalità, il gioco potrà terminare solo dopo la sconfitta del **boss**.

- Il gioco presenterà un' organizzazione a **round** di difficoltà crescente. All'avvio di ogni round partirà anche il tempo e, contemporaneamente, compariranno i nemici. Questi, saranno da sconfiggere prima di giungere al round successivo. I round prefissati per la modalità base del gioco sono 3, in modo che, una volta terminati, sia possibile accedere alle altre **stanze** presenti nel gioco.

- Il **player** deve poter muoversi all'interno dell'ambiente di gioco e sparare nelle quattro direzioni, uccidendo i nemici, guadagnando punti e finendo il gioco nel minor tempo possibile.

- Elemento chiave è il **tempo** che viene sospeso nel momento esatto in cui si termina un round, per poi riprendere all'inizio del successivo. Il tempo è fondamentale, perchè solo per i giocatori più forti, cioè coloro che riusciranno a terminare il gioco uccidendo il boss, riceveranno un bonus in **punti** (calcolati in base al tempo impiegato).

- Dalla definizione del tempo si evince che un altro elemento fondamentale sono i **punti**. Questi permettono al giocatore di dimostrare le proprie abilità, in quanto, essendo un elemento chiave del videogioco rappresentano anche la "moneta" con cui sarà possibile acquistare un **power up** all'interno del negozio. I punti si ottengono uccidendo i nemici e terminando il gioco ma, naturalmente, possono anche essere perduti. Ciò accade quando si viene colpiti dai proiettili nemici, oppure nell'acquisto di **power up**.

- Il **power up** disponibile all'interno del negozio consente di acquistare una vita in più.

- Le **stanze** all'interno del gioco sono 3: Main Room, Boss Room e Shop Room. Queste stanze saranno esplorabili in sequenza, partendo dalla Main Room, dove si svolgono i round iniziali, passando per la Shop Room, dove a seconda della necessità dell'utente si

potranno acquistare dei power up, ed infine la Boss Room in cui si svolgerà il combattimento finale contro il boss.

- I nemici sono regolati dall'utilizzo di una **AI**, cioè un'intelligenza artificiale che, seppur minimale, permette ai nemici di avere una propria strategia di movimento e di sparo dei proiettili.
- All'interno del gioco deve essere possibile mettere in **pausa** in un qualsiasi momento.

Requisiti funzionali opzionali

- Realizzazione di una seconda modalità gioco sempre regolata a tempo ma a sopravvivenza (**survival mode**). Questa modalità consiste nella generazione di round dinamici ed infiniti.
- Realizzazione di una modalità in cui il player non può morire, in modo da poter testare i comportamenti dei nemici e del boss senza però subire danni.
- Fornire un' AI più avanzata al boss, così che questi possa distinguersi dai nemici comuni.
- Aggiunta di più item all'interno dello shop.

Requisiti non funzionali

- Il gioco, presentando al suo interno più thread in esecuzione simultaneamente, deve garantire un corretto funzionamento.
- Il gioco deve essere fluido e non impegnare eccessivamente le risorse.
- Il gioco deve essere **resizable**, ossia in grado di adattarsi alle dimensioni che l'utente desidera mantenere della finestra di gioco.

1.2 Analisi del modello di dominio

Kill To Survive (**KTS**) è un gioco composto da diverse entità che interagiscono tra di loro in maniera più o meno complessa.

All'interno del nostro mondo di gioco una stanza (**room**) è composta da **muri**, **porte**, **nemici** e/o oggetti (**power up**) a seconda del tipo di stanza. I muri delineano il confine delle stanze, che il nemico non potrà mai valicare, mentre le porte rappresentano il passaggio tra una stanza e la successiva.

Come accennato precedentemente la modalità normale (**normal mode**) del videogioco (requisito funzionale obbligatorio) si compone di tre stanze:

- **Main Room**. Stanza principale in cui si effettueranno i **rounds**. Al centro di questa è possibile trovare un bottone (**button**) che permette al giocatore, mediante la sua pressione, di avviare i round.
- **Shop Room**. Una volta completati i round nella stanza iniziale, si sbloccherà la porta per accedere alla Shop Room. All'interno di questa stanza ci saranno uno (requisito funzionale obbligatorio) o più (requisito funzionale opzionale) oggetti (detti **power up** nel nostro dominio di gioco).
- **Boss Room**. Una volta acquistati gli oggetti desiderati si potrà accedere a questa ultima stanza. Non appena entrati il tempo partirà e si dovrà affrontare un nemico con caratteristiche e comportamenti diversi dei nemici incontrati finora (**boss**).

All'interno della main room, dove si svolgeranno i rounds, si incontreranno vari nemici ognuno dei quali mosso da una AI (**intelligenza artificiale**). Questa, seppur minimale, permette ad ognuno di essi di muoversi, rimanere fermo, sparare, ecc.

I round vengono avviati dalla pressione di un **bottone** all'interno della stanza principale. Alla fine di ogni round il tempo verrà fermato e il bottone tornerà nello stato iniziale, in modo che possa essere premuto nuovamente per poi fare partire il round successivo.

Tramite l'uccisione di questi nemici si guadagneranno **punti** che serviranno poi per acquistare gli oggetti presenti nel negozio. I punti tuttavia possono anche essere perduti. Infatti, ogni qualvolta che si viene colpiti dal proiettile nemico, se ne perderanno un numero prefissato.

Durate lo svolgimento dei rounds l'avanzare del **tempo** calcolerà quanto l'utente impiega per portare a termine ogni round. Questo, rappresenta un fattore fondamentale in quanto, solo i giocatori più abili che riusciranno ad uccidere il boss, guadagneranno un bonus incrementale. I punti guadagnati, se tanto elevati da superare quelli di altri giocatori, saranno poi utili per ottenere una posizione nella **classifica** dei giocatori.

Come richiesto dai requisiti non funzionali dovrebbero essere sviluppate altre due modalità:

- **God Mode**. Modalità in cui il player non potrà perdere vite, ma potrà giocare senza la preoccupazione di poter essere ucciso.

- **Survival Mode**. In questa modalità molto difficile il gioco sarà limitato alla sola Main Room, senza poter mai accedere alle altre stanze di gioco. Di conseguenza non sarà mai possibile accedere alla Shop Room per acquistare oggetti, oppure terminare il gioco uccidendo il boss. In questa modalità l'obbiettivo è quello di sopravvivere affrontando di volta in volta round dinamici via via sempre più complessi.

In entrambe le modalità il player avrà a disposizione un numero prefissato di vite, al termine delle quali morirà e sarà possibile iniziare una nuova partita da capo.

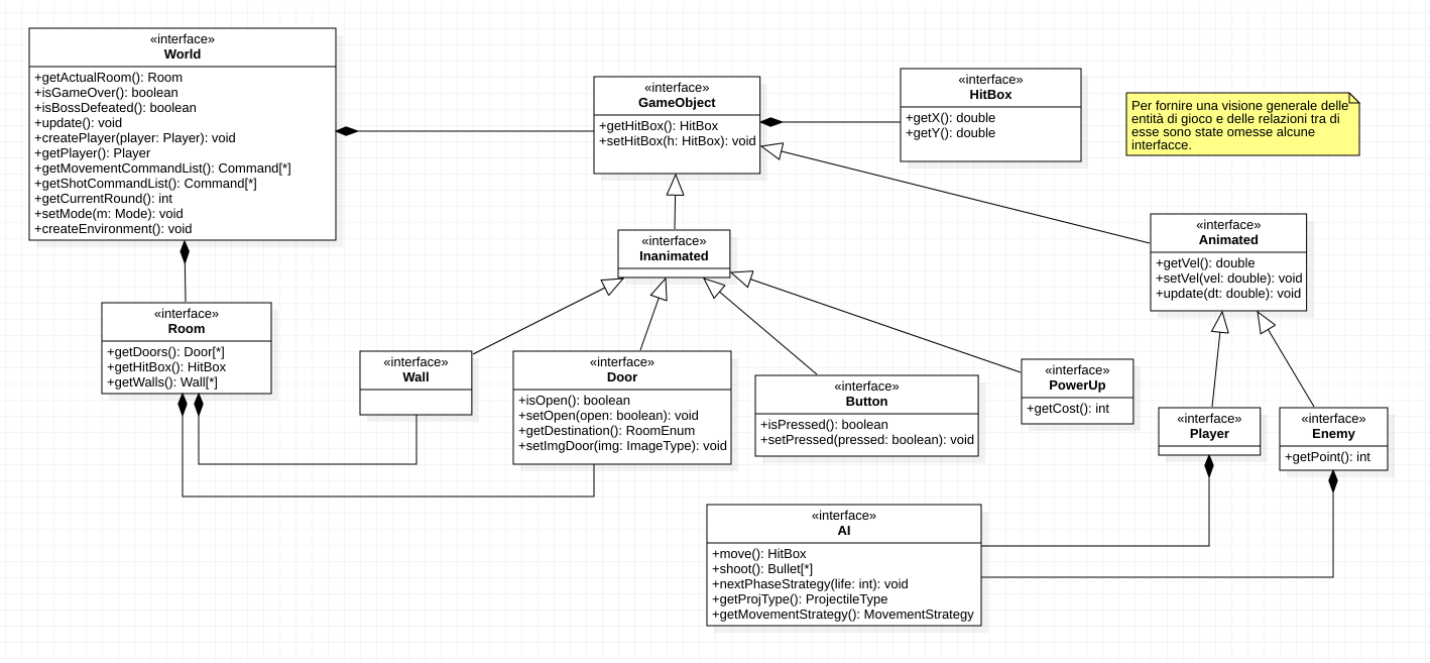

Figura 1.1: Schema UML esemplificativo che riporta le entità descritte in linguaggio naturale.

Capitolo 2

DESIGN

Il software adotta uno specifico pattern architetturale di programmazione, il **model-view-controller** (**MVC**). Grazie alla scelta di questo abbiamo un'architettura multi-tier ovvero dove le varie funzionalità del software sono logicamente separate.
Nello sviluppo del progetto abbiamo cercato di mantenere un MVC il più "puro" possibile, in cui la componente del model riceve i dati ma non comunica con nessun'altra componente, mentre il controller e la view possono scambiarsi i dati.

2.1 Architettura

Il pattern è basato sulla separazione dei compiti fra i vari componenti.

- Il **model** si occupa dell'intera logica di manipolazione e di aggiornamento dei dati. Esso si occupa dunque del mantenimento dei dati relativi al mondo di gioco, necessari per la corretta rappresentazione grafica delle varie entità effettuata nella view.

- Il **controller** detta il tempo di gioco, i tempi di aggiornamento del model e della view. Questa componente decide la frequenza di aggiornamento delle entità nel mondo di gioco e di rappresentazione grafica. Rappresenta un "traghettatore" tra le due componenti, come se tra loro fosse posto un filtro.
  Il controller ad esempio riceve dalla view la notifica dei comandi premuti dall'utente e, comunicandoli al model, verranno attuate le modifiche necessarie alle componenti di gioco. Ad ogni ciclo del controller esso richiamerà il model per aggiornare i dati del mondo (movimenti, collisioni, spari ecc...), per poi notificare alla view di disegnare sullo schermo un'immagine che rappresenti le componenti sempre aggiornate.

- La **view** si occupa infine della rappresentazione grafica delle componenti di gioco memorizzate nel model. Un suo compito fondamentale è quello di rimanere in attesa del verificarsi di determinati eventi quali la pressione di un tasto o di un bottone sullo schermo da parte dell'utente, per poi comunicarlo al controller.

L'architettura si basa su di una "semplice" macchina a stati finiti dove il controller richiama il model, che manipola i dati secondo le leggi del mondo di gioco. La view rappresenta questi dati e, in caso di ricezione di un comando da parte dell'utente, lo notifica al controller.
Ognuna delle tre componenti principali (**model**, **view** e **controller**) comunica con le altre tramite un'interfaccia, che espone le funzionalità pubbliche accessibili dall'esterno.

Obbiettivo che abbiamo cercato di raggiungere durante il lavoro è stato quello di mantenere una separazione funzionale delle tre componenti del pattern, in modo che ognuna potesse lavorare indipendentemente dalle altre e fornisse verso l'esterno, tramite l'utilizzo di interfacce, solo i mezzi di comunicazione tramite cui inviare/ricevere dati. Tutto questo per far si che al cambiamento radicale della view le altre componenti potessero continuare a funzionare senza problemi, oppure permettendo al mondo di gioco di essere riutilizzabile in altre rappresentazioni.

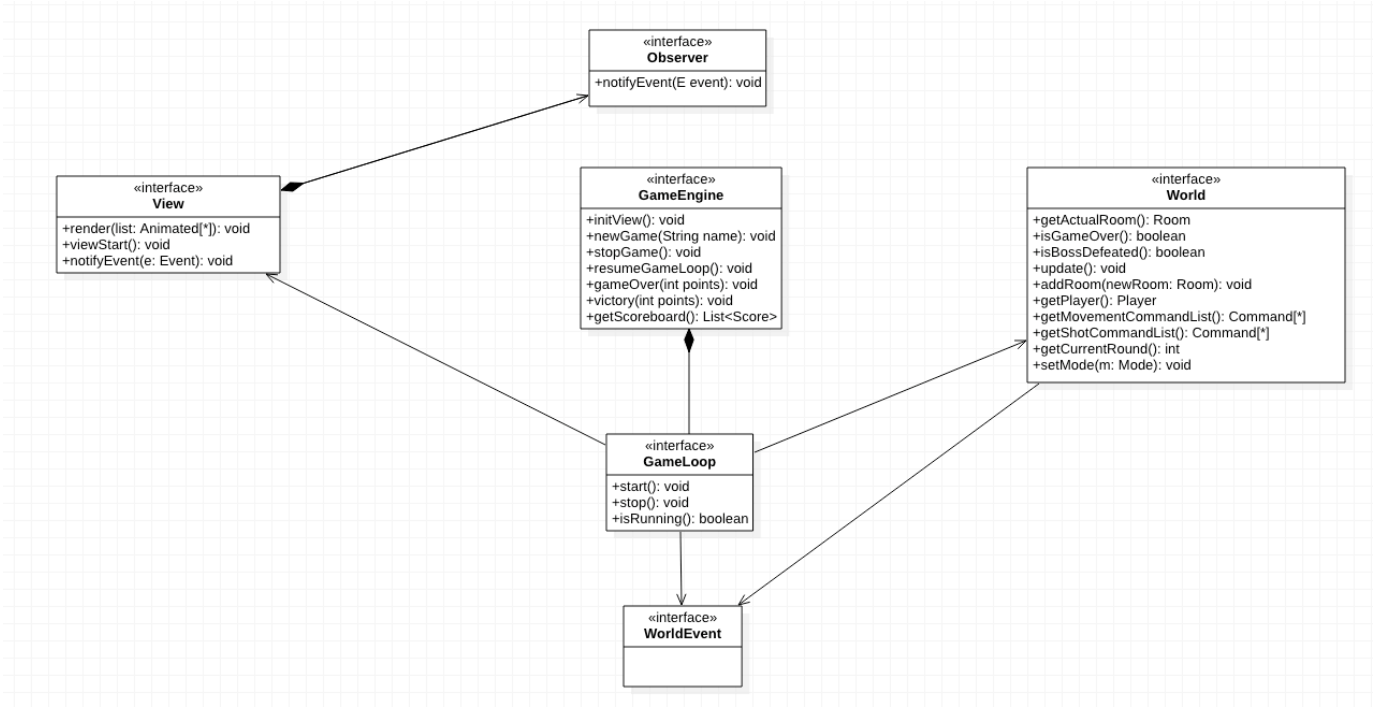

Figura 2.1: Schema UML esemplificativo che fornisce una visione generale delle principali componenti del pattern MVC e di come esse comunicano tra loro.

2.2 Design Dettagliato

- *Vincenzi Mattia*

Il primo aspetto su cui mi sono concentrato durante la progettazione e realizzazione è stata quella del world di gioco. In particolare modo ho prestato attenzione all'implementazione degli oggetti animati quali il *player* e i *nemici*.

Animated Objects

Come mostrato dalla ***figura 2.2.1*** ho deciso di implementare una gerarchia di interfacce, in modo che ognuna potesse aggiungere qualcosa a quella precedente e che il suo "compito" potesse essere esplicato solo dalla lettura del nome.
Ad esempio, partendo dalla radice, l'interfaccia **GameObject** aggiunge solo le proprietà fondamentali per ogni oggetto del gioco come l'HitBox. Successivamente c'è l'interfaccia **Animated**, che rappresenta gli oggetti animati, cioè dotati di velocità e sono proprio queste le proprietà che aggiunge. In seguito, con l'interfaccia **LivingObject** vengono aggiunti i metodi necessari per lavorare con la vita di un' entità. L'interfaccia **Shooter** aggiunge le funzionalità per sparare, con tutti i relativi metodi. Infine, l'interfaccia **Character** (Personaggio) fornisce un solo un metodo molto importante, cioè quello dell'AI.

In questa parte ho deciso di utilizzare un **template method** all'interno dell'interfaccia **AbstractAnimated** per definire la strategia di movimento di un' entità. Il template method è il metodo **update** che al suo interno utilizza il metodo astratto **move**, questo verrà poi implementato nelle classi che estenderanno la classe astratta e la sua implementazione dipenderà dalle specifiche necessità di tale classe.

Questa decisione è nata dal fatto che sia i **Bullet** che le entità, quali player e nemici, avevano la necessità di muoversi, ma in modi diversi (infatti i proiettili necessitano anche del decremento del range ad ogni movimento).

In un futuro, avendo utilizzato questa strategia di risoluzione, se nel gioco vi fosse la necessità di aggiungere nuove entità in movimento, queste potrebbero essere inserite nella gerarchia senza dover apportare modifiche. Inoltre, grazie all'utilizzo del pattern **template method** tale entità aggiunta potrà implementare la propria strategia di movimento.

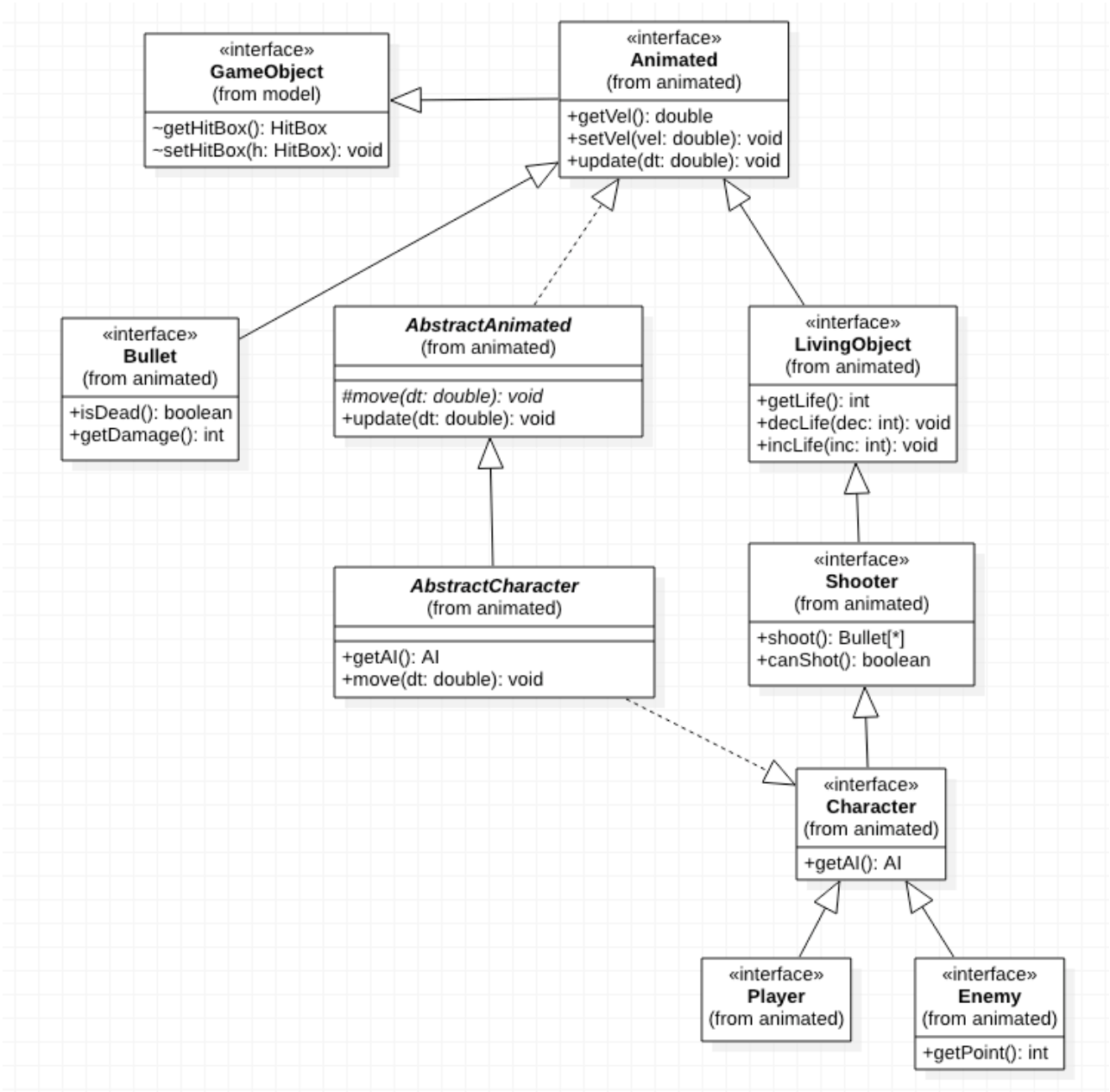

Figura 2.2.1: UML della gerarchia degli oggetti/entità di gioco.

Artificial Inteligence

Forse l'aspetto che mi ha stimolato maggiormente a ragionare e di conseguenza a valutare più situazioni possibili è quello dell'AI, ossia l'intelligenza artificiale.

Per arrivare alla soluzione finale, presentata nella ***figura 2.2.2***, sono passato attraverso varie fasi di ragionamento tramite un'approccio incrementale e valutando pro e contro di ogni implementazione ideata.

In primis, nel modo più semplice ed immediato possibile, avevo pensato di creare una classe concreta per ogni tipo di nemico, quindi per ogni combinazione possibile di uno sparo e di un movimento dovevo creare una classe. Questo approccio è parso subito errato, per via della proliferazione delle classi e della ripetizione di codice.

Da qui ho notato l'esigenza di estrapolare la logica di movimento per dare una maggior chiarezza al codice e utilizzare il pattern **strategy** per nascondere dietro una generica interfaccia tutti i tipi di movimento. Ho subito notato che lo stesso ragionamento poteva essere effettuato anche per la logica di sparo di diversi tipi di proiettili, e per questo motivo anche in questa situazione ho utilizzato il pattern **strategy**.

Arrivato al punto in cui le mie entità incapsulavano al loro interno un oggetto dedicato al movimento ed uno alla strategia di sparo ho deciso di fare un ulteriore passo aventi, dando una maggior manutenibilità al codice e favorire l'indipendenza funzionale dei concetti. Per fare ciò ho creato il concetto di intelligenza artificiale (**AI**), che al suo interno incapsula la strategia di movimento e di sparo.

I miei personaggi di gioco (**Character**) incapsuleranno quindi un solo oggetto AI che verrà costruito alla creazione del medesimo personaggio e potrà accoppiare dinamicamente una qualsiasi strategia di movimento e di sparo di proiettili.

Come annotato a fianco della figura sono state riportate solo alcune delle specializzazioni realmente implementate nel codice, ma non penso sia importante andare ad addentrarmi nella spiegazione di queste.

Naturalmente per favorire la riutilizzabilità del codice le strategie di movimento sono state utilizzate sia per le entità che per i proiettili.

Questo tipo di soluzione adottata si predispone ai cambiamenti e all'aggiunta di ulteriori tipi di movimento o di sparo. Una volta pensato questo design durante lo sviluppo è stato semplice ed immediato poter aggiungere nuove classi come specializzazioni delle interfacce *MovementStrategy* e *ProjectileType* così da creare nuovi nemici sempre diversi.
Di conseguenza, anche un futuro ampliamento del gioco, ad esempio costituito da più livelli, da più di un boss o comunque da un numero più ampio di nemici, sarà facilmente attuabile senza stravolgere l'architettura.

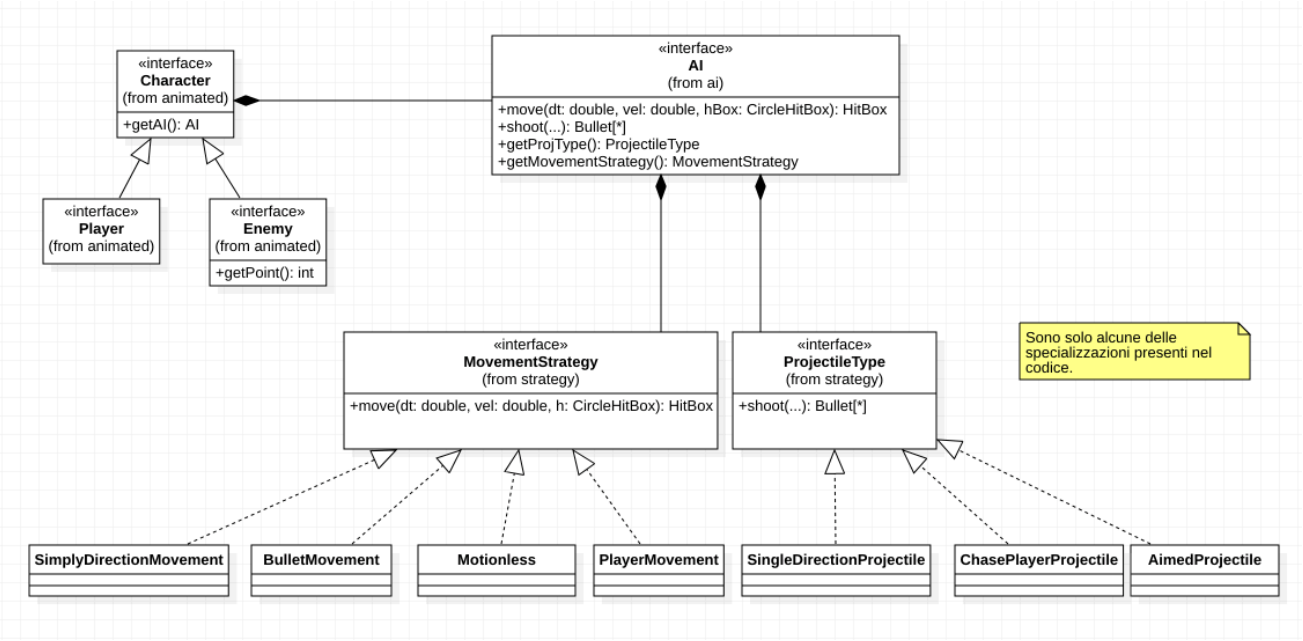

Figura 2.2.2: UML che rappresenta il pattern strategy utilizzato per l'AI.

Creazione personaggi

Per quanto riguarda la creazione dei personaggi di gioco, ossia il personaggio, i vari tipi di nemici ed infine il boss ho deciso di utilizzare il **pattern factory** in modo da concentrare dentro una classe la creazione dei personaggi di gioco. Tale classe. che permette di creare gli oggetti, è nascosta dietro ad un' interfaccia in modo che esponga i metodi utilizzati per la creazione.

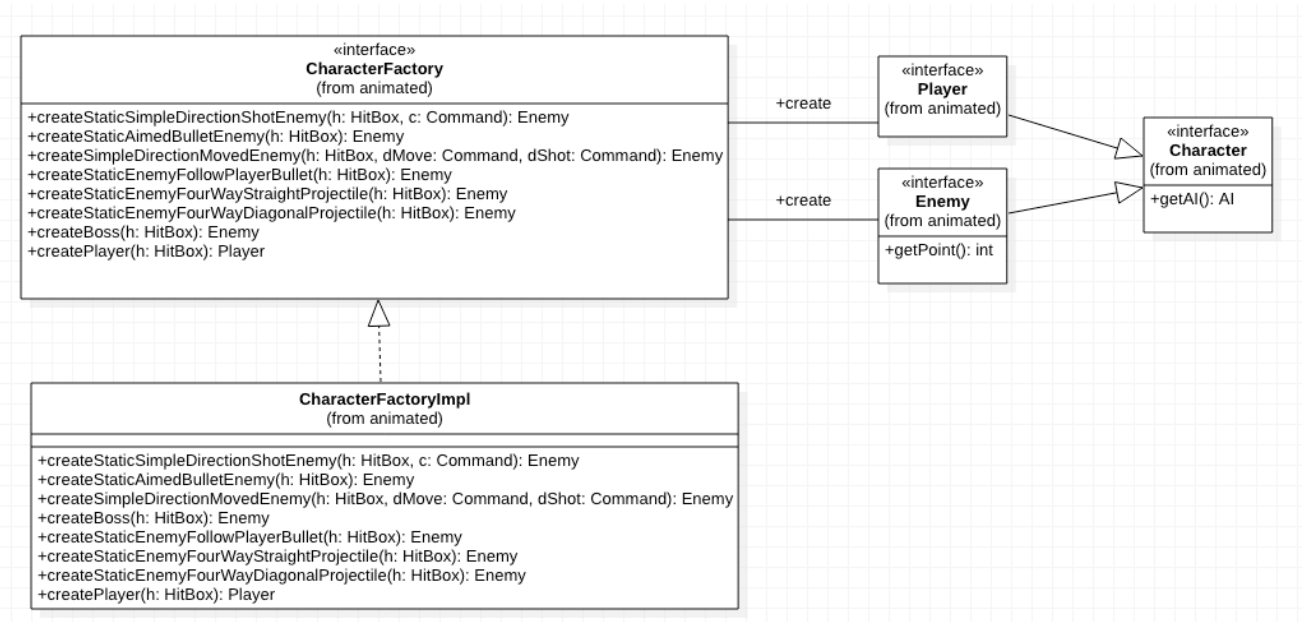

Figura 2.2.3: Schema UML rappresentante la factory di character.

View Design

Nella ***figura 2.2.4*** viene mostrata una panoramica principale dei componenti della view. **View** rappresenta l'interfaccia di comunicazione dal controller alla view e viceversa, permette di notificare gli eventi provenienti dalle scene di gioco agli observer e quelli derivanti dal controller alla view.

Il **DrawerManager** si occupa di disegnare sui canvas di gioco l'intera scena e di attuare cambiamenti alle dimensioni dei canvas al verificarsi dell'evento di resize.
Infine, il **ViewManager** al suo interno utilizza uno stack per mantenere lo stato delle scene attive, mantenendo in cima quella che deve essere mostrata all'utente e al di sotto quelle attive precedentemente (e che possono tornare attive). Per gestire questo stack si utilizzano i classici metodi di push e di pop.

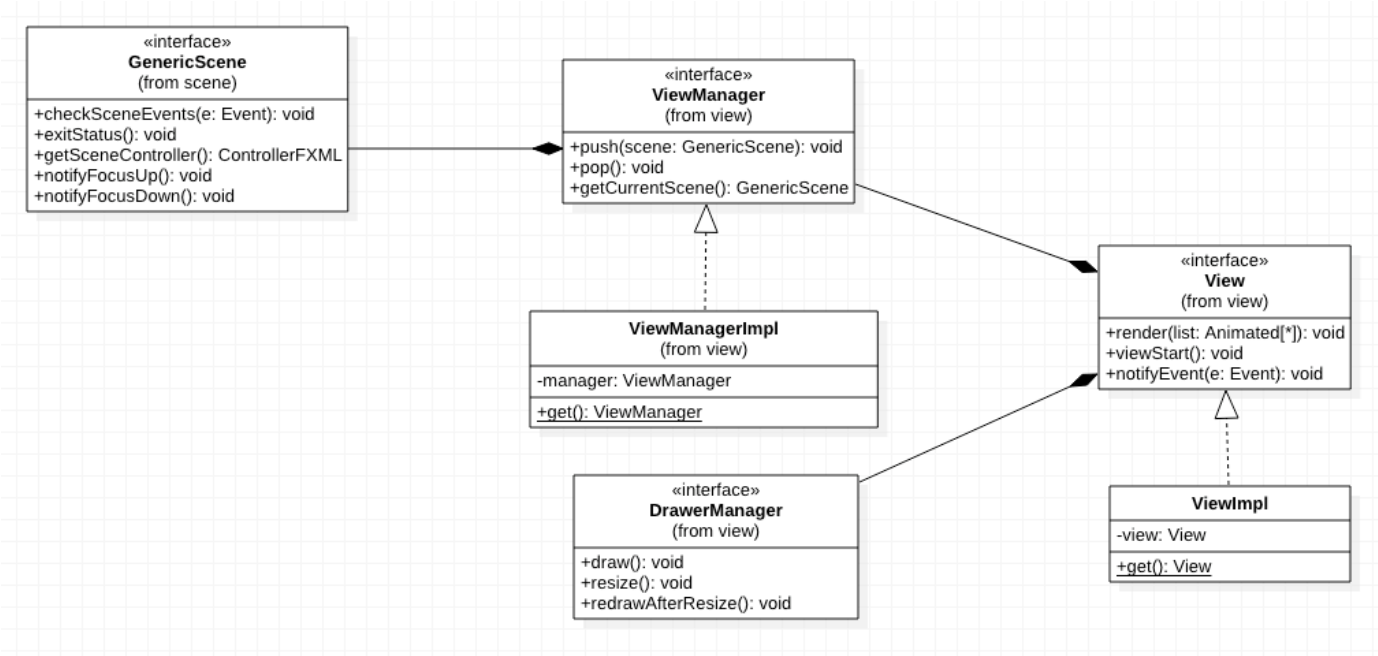

Figura 2.2.4: UML che rappresenta l'organizzazione delle principali componenti della view w il SINGLETON pattern.

Ho deciso di utilizzare sia per la **View** che per il **ViewManager** il **pattern creazionale singleton**, perchè di essi sarà presente sempre e solo un'unica istanza. Questo permette sia di utilizzare la classe da un qualunque punto, sia di non doverla necessariamente passare con metodi setter oppure come parametro nei costruttori a qualunque classe ne necessiti il riferimento. Questo pattern si può vedere sempre nella ***figura 2.2.4***, perchè in entrambe le classi è presente un campo privato del tipo della classe e un metodo *get()* che si occuperà di restituirne l'unica istanza.

Scene di gioco

Come mostrato nella ***figura 2.2.5***, per quanto riguarda la rappresentazione delle varie scene di gioco ho deciso di costruire una gerarchia.

Per controllare i diversi eventi che possono accadere nelle varie scene e, di conseguenza, in alcune di esse fare accadere qualcosa alla pressione del tasto 'esc' ho implementato nuovamente il pattern **template method**. Questo è uno solo e corrisponde al metodo attaccato mediante l'utilizzo di una **lambda expression** all'*handler* di eventi di tipo *KeyEvent*.

All'interno del template method vengono richiamati nel caso della pressione di 'esc' il metodo astratto *exitStatus()*, e poi l'altro metodo astratto *checkSceneEvents()*. Il metodo *exitStatus()* consente di attuare azioni dipendenti dalla scena attuale, ad esempio nella scena di gioco viene fatta caricare la scena di pausa.
Il metodo *checkSceneEvents()* permette di controllare eventi specifici nelle diverse scene, e di creare e notificare eventi agli observer.

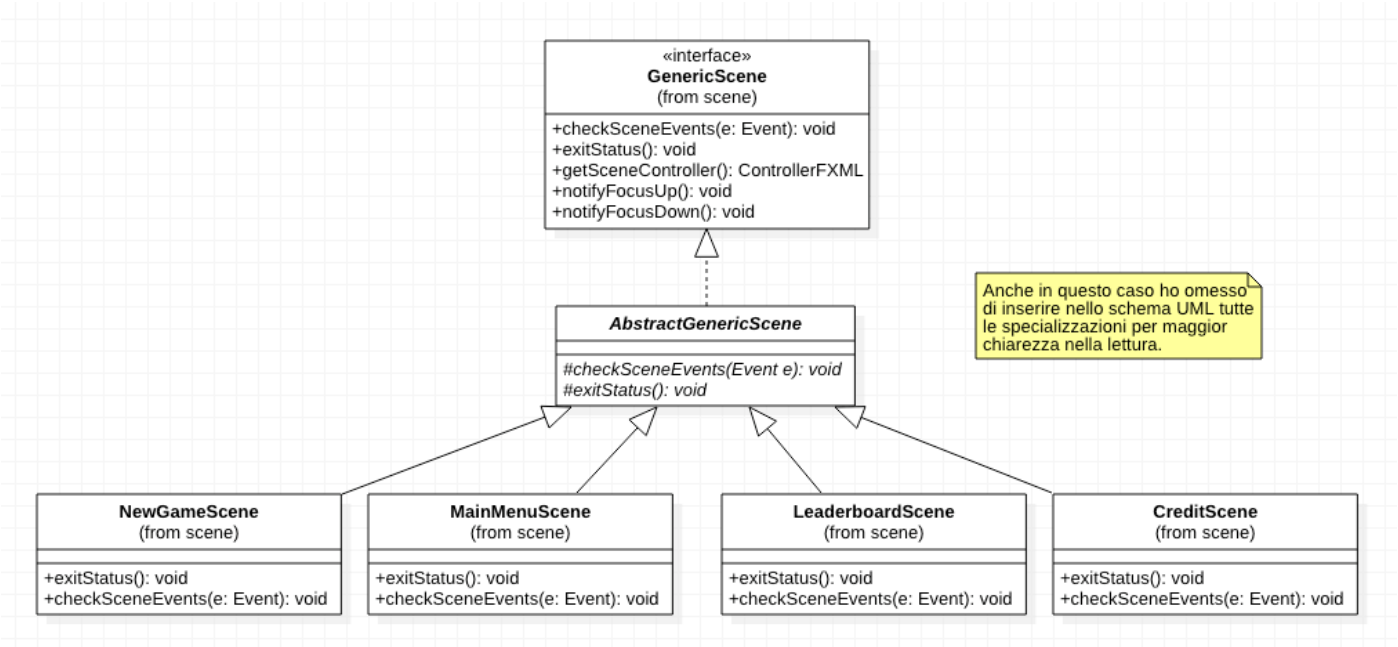

Figura 2.2.5: Schema UML rappresentante la gerarchia delle scene del gioco.

Per quanto riguarda la creazione delle scene di gioco mostrate nella ***figura 2.2.5*** ho utilizzato sempre il **pattern factory** ma, in tal caso, ne ho utilizzato un altra variante, ossia lo *static factory* (vedi ***figura 2.2.6***).
A differenza della versione precedente del pattern factory che avevo utilizzato nel model per la creazione dei personaggi di gioco (dove ogni qual volta voglio utilizzare la factory la devo istanziare), in questo modo posso richiamare il metodo di creazione direttamente dal nome della classe, perchè statico.

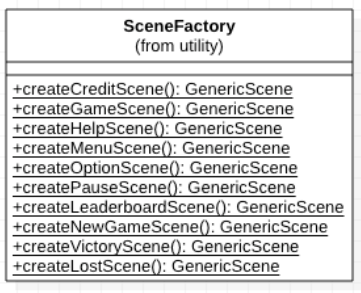

Figura 2.2.6: Schema UML rappresentante la factory delle scene di gioco.

Proxy Pattern

Per accedere al file system e caricare sia le immagini che i file fxml ho deciso di utilizzare un pattern non visto a lezione, il **pattern proxy**.
È un pattern strutturale che permette di nascondere dietro a una interfaccia l'accesso ad una risorsa, in questo caso ai file fxml o alle immagini.

Nella ***figura 2.2.7*** viene rappresentato lo schema UML relativo al caricamento dei file fxml, sarebbe inutile rappresentare anche quello per le immagini perchè identico.

La classe *ProxyFXMLLoader* contiene al suo interno una classe innestata (*RealFXMLLoader*), alla quale chiede il "reale" caricamento dei file se ancora non sono stati caricati.
Le immagini o i file fxml, vengono caricati in modo "*lazy*" (ritardato), cioè solo su richiesta, oltretutto, una volta caricati la prima volta vengono tenuti in memoria (tramite l'utilizzo di una mappa), in modo da non effettuare inutilmente dei dispendiosi accessi al file system.

Per specificare quali file fxml richiedere all'interfaccia del proxy e, di conseguenza anche come chiave della mappa per ricordare quali sono stati caricati e quali no, viene utilizzata una enumerazione (*SceneType*). L'utilizzo di una enumerazione è preferibile rispetto a quello di semplici stringhe o costanti, rende il codice più duttile a modifiche e più semplice la realizzazione confronti.

L'utilizzo di questo pattern e di una enumerazione ha facilitato l'aggiunta di schermate o di immagini durante lo sviluppo del gioco, infatti una volta scritto il codice per aggiungere/modificare un'immagine o un file è stato necessario solo modificare la rispettiva voce nell'enumerazione.

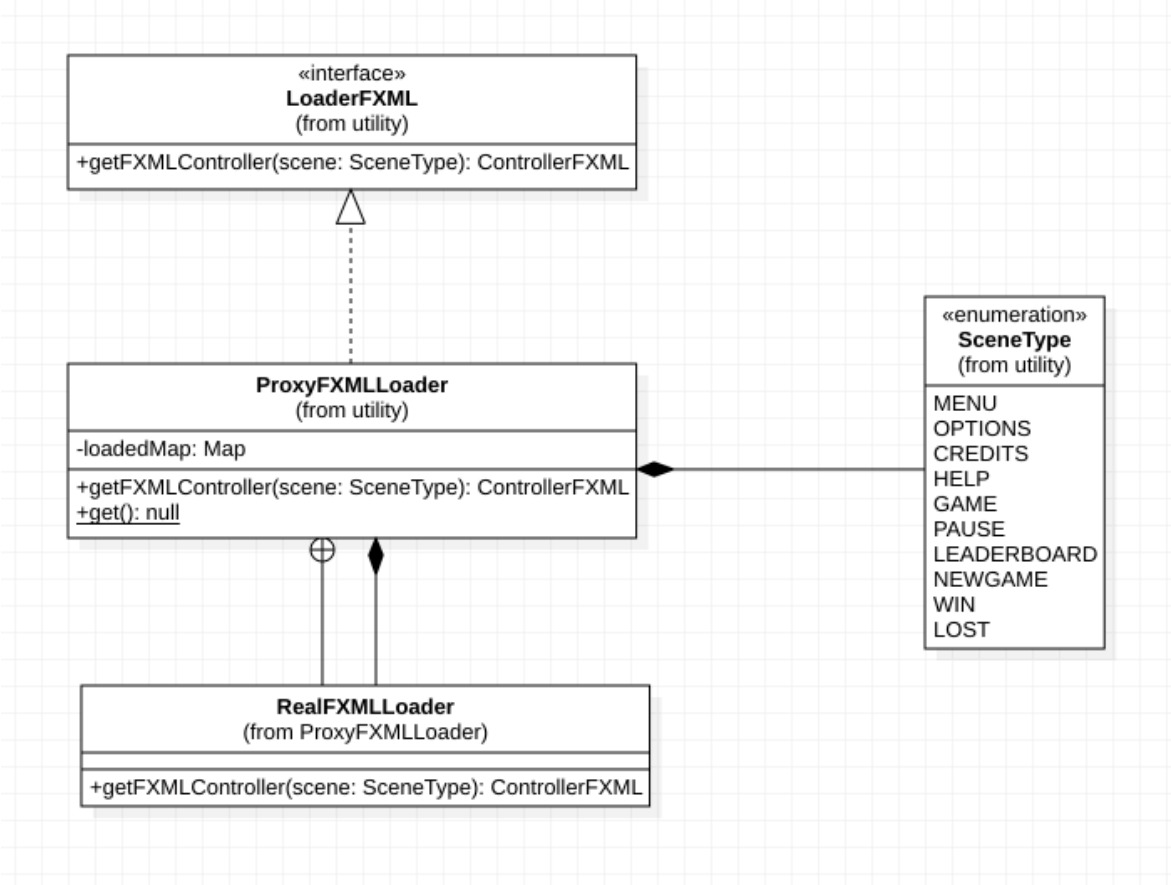

Figura 2.2.7: UML che rappresenta la realizzazione del pattern proxy per il caricamento dei file FXML.

- *Vaienti Andrea*

Mi sono occupato dell'implementazione del controller, componente del MVC che si occupa della gestione dell'intero gioco; l'essenza di esso è racchiuso nelle classi: *GameEngine*, *GameLoop*, *Observer*. (***figura 2.1***).

GameEngine

E' il fulcro del controller e gestisce l'applicazione dalla creazione dell'aspetto grafico, appena *Kill to Survive* viene eseguito, fino alla chiusura di tutti i thread quando l'utente esce dal programma. *GameEngine* coordina non tanto le fasi di gioco, che sono completamente affidate al model, bensì le fasi dell'applicazione, come il menù iniziale, l'avvio di una nuova partita di gioco, il menù di pausa e la vittoria o sconfitta del giocatore.
Inoltre avendo una leaderboard all'interno del gioco, si occupa di tutte le fasi di lettura e scrittura su file, implementando così la funzione di salvataggio.
Non avendo l'esigenza di avere più controller, in quanto GameEngine è sempre attivo durante tutto il periodo di esecuzione dell'applicazione, ho deciso di adottare il **pattern Singleton**: in questo modo è impossibile crearlo più volte e inoltre risulta più semplice, per le classi che lo utilizzano, richiamare i metodi.

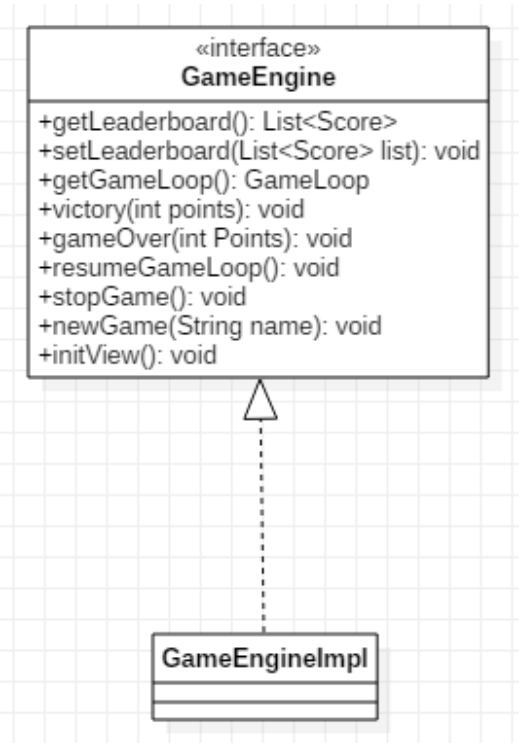

Figura 2.2.8: UML raffigurante il GameEngine.

GameLoop

E' una classe di fondamentale importanza non solo per il controller ma anche per l'intera applicazione. Si occupa dell'aggiornamento del mondo di gioco comunicando al model quali sono stati gli spostamenti o gli spari del player tra il frame precedente e quello successivo; di conseguenza richiama anche i metodi della view per aggiornare l'aspetto grafico.
Ciò avviene invocando all'interno del metodo *run()*, che implementa l'interfaccia *Runnable*, i metodi *update()* e *render()*. Inoltre, sempre all'interno del metodo *run()* viene invocato *checkEvent()*, attraverso il quale il model comunica al controller gli eventi accaduti all'interno del mondo di gioco, in maniera tale da poter aggiornare le statistiche del giocatore, come vita e punteggio. Di conseguenza l'oggetto di tipo *Gameloop* viene istanziato all'esecuzione di ogni nuova partita, che può essere avviata più volte nella stessa sessione di gioco. Per questo motivo, a differenza di GameEngine, si è deciso di <u>non</u> utilizzare il **pattern Singleton**.

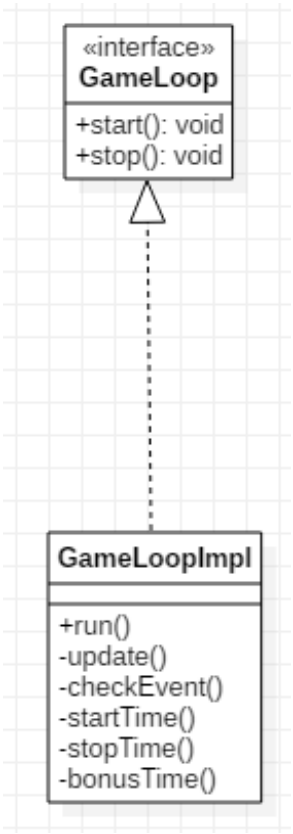

Figura 2.2.9: UML raffigurante il GameLoop.

Observer

Le comunicazioni dalla view al controller sono implementate attraverso il **pattern Observer**, che controlla gli eventi, definiti dall'interfaccia *Event*, che gli vengono notificati dalla view. Scendendo più nel dettaglio abbiamo dovuto implementare *ButtonEvent*, relativo alla pressione di un bottone all'interno della schermata di gioco, e *KeyEvent*, relativo alla pressione di un tasto.
La view appena percepisce una di queste azioni genera il corrispettivo evento, che poi notificherà al relativo *Observer* del controller, poiché anch'esso si suddivide in *ButtonObserver* e *KeyObserver*, in base all'evento che devono gestire. Sarà poi compito dell'Observer, dopo aver esaminato l'evento, attuare i dovuti cambiamenti.

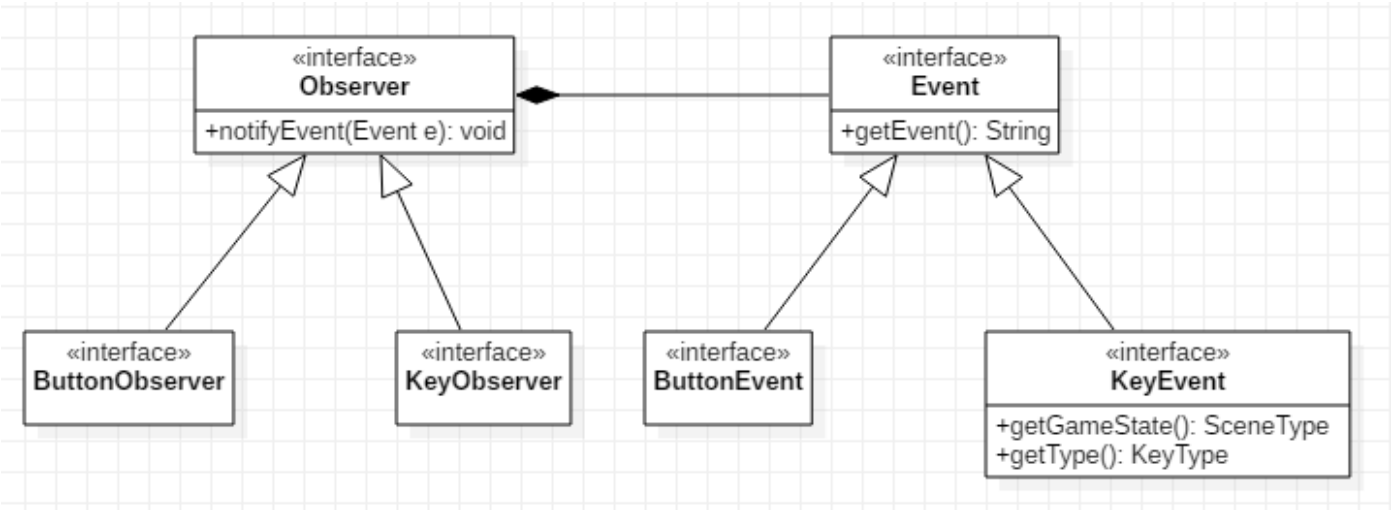

Figura 2.2.10: UML raffigurante il pattern Observer e le componenti coinvolte.

Punteggio

Inoltre, essendo il raggiungimento del punteggio più alto all'interno della leaderboard lo scopo del gioco , abbiamo dovuto inventare una funzione matematica che potesse calcolare il punteggio finale sulla base del tempo impiegato per sconfiggere il boss e vincere la partita. Confrontando varie proposte abbiamo ritenuto opportuno utilizzare una **funzione gaussiana** che abbiamo poi adattato al nostro specifico caso con appropriati cambiamenti.

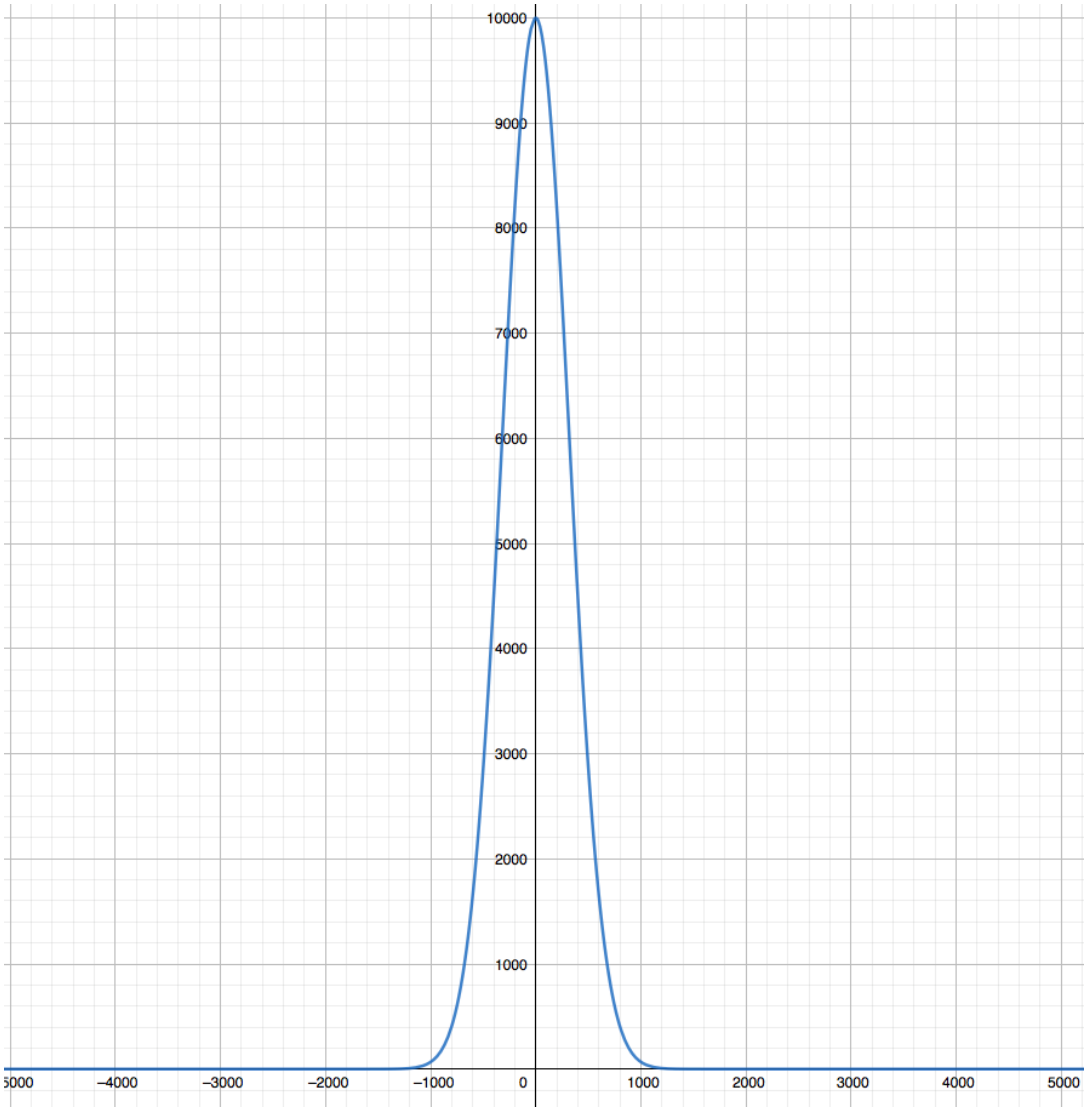

Figura 2.2.11: Schema rappresentante la funzione di calcolo del punteggio in base al tempo (x).

- *Cristurean Denis*

Bullet

Per modellare l'azione di sparo ho deciso di affidare la creazione del proiettile all'interfaccia *Bullet*, che oltre a tenere traccia del danno che questo può infliggere e del range da esso percorso, estendendo l'interfaccia *Animated* si porta dietro funzioni utili come update, getter e setter per la velocità.

Per definire il movimento dei *Bullet* (*move()*) si è deciso di utilizzare un *MovementStrategy*, e siccome *move()* deve tenere conto del range che i proiettili hanno, ho deciso di definirlo come una funzione che calcola la distanza percorsa dal proiettile dall'ultimo update.

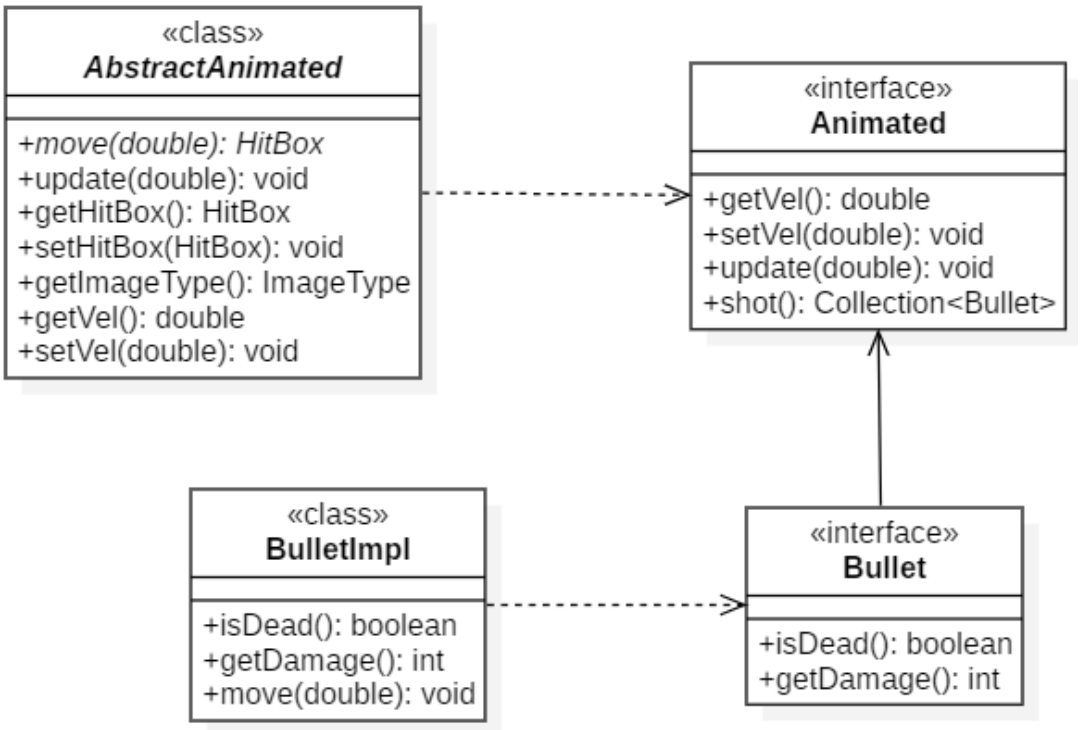

Figura 2.2.12: Schema UML rappresentante i proiettili.

Un'altra parte dello sviluppo dei proiettili è stata l'implementazione dei pattern di attacco vari nemici e del giocatore. Applicando il **pattern strategy** attraverso l'interfaccia *ProjectileType* (*shoot()*) è stato possibile diversificare il comportamento dei proiettili per nemici, player e boss in modo da avere proiettili che rincorrono, mirati, a croce, a X, multipli paralleli e multipli inseguitori.

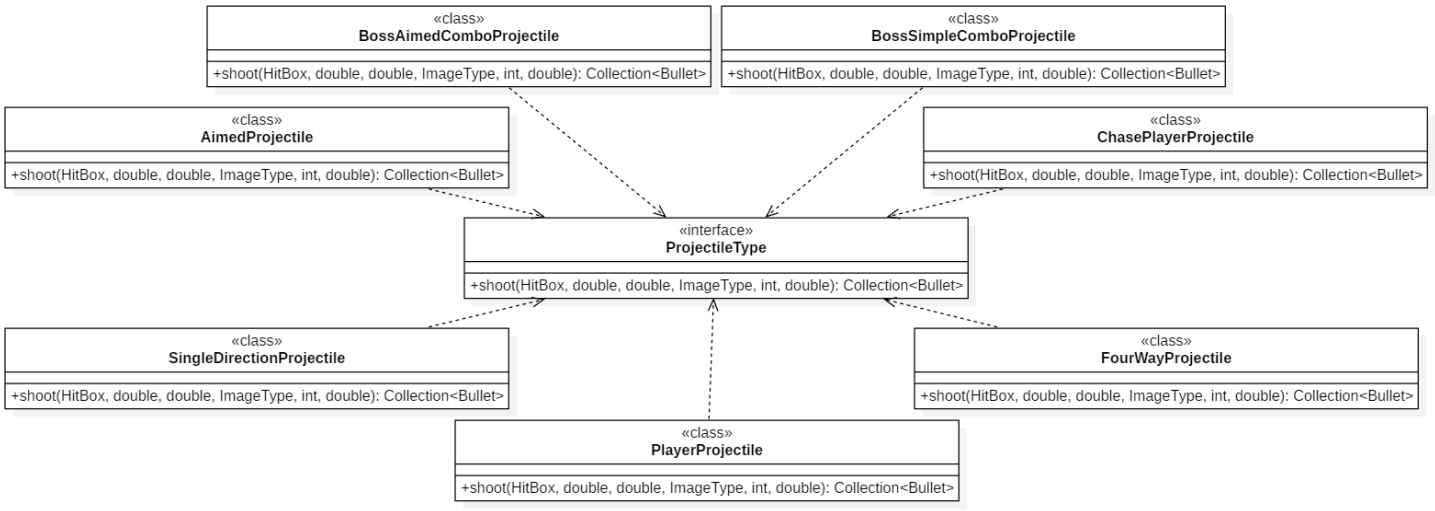

Figura 2.2.13: Schema UML rappresentante l'interfaccia projectile.

Collision

Una parte di cui mi sono occupato è stata la rilevazione delle collisioni. Nel gioco sono infatti presenti diverse entità, items, muri e porte che ad urto rilevato devono poter reagire, ognuna secondo regole diverse. In particolare ho identificato:

- Collisione tra entità e muro. [9]
- Collisione tra due entità [10]
- Collisione tra player e porta.

Tutte queste regole restituiscono un tipo Bool, eccetto la rilevazione di collisione tra entità che restituisce una *Collection<Command>* e rappresenta le direzioni vietate. Ho deciso di svilupparla come classe utility perchè in fase di sviluppo (carta e penna) ho notato la mancanza di campi e quindi nessun bisogno di creare un'istanza.

Views

Per la realizzazione delle scene ho utilizzato il tool **SceneBuilder** producendo cosi diversi file FXML, per ciascuno di questi ho poi scritto il relativo Controller. Per gestire l'intercambio delle view è stata utilizzata una struttura dati di tipo *Stack* realizzata dal mio collega Mattia Vincenzi.

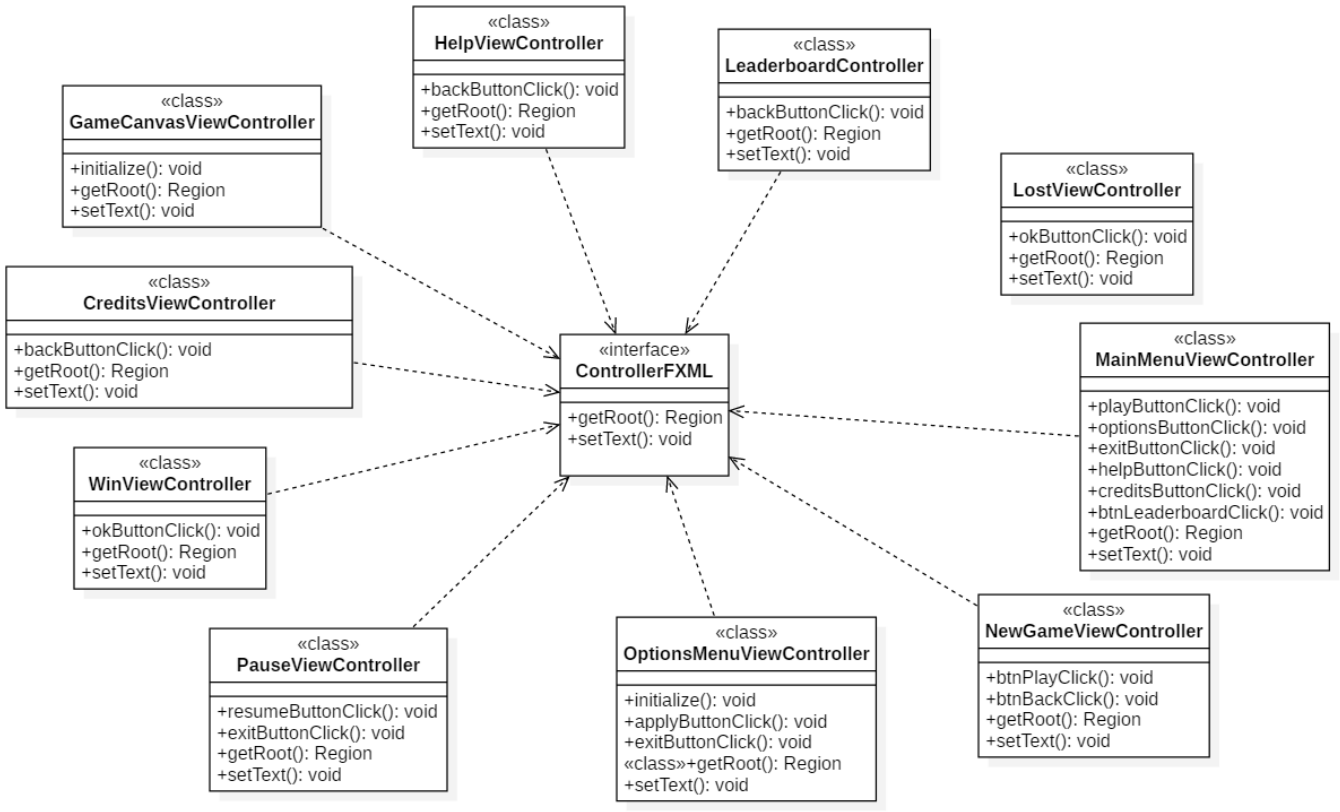

Figura 2.2.14: Schema UML rappresentante i controller della view.

- *Beshiri Rei*

World di gioco

Lo sviluppo del model e in particolare del mondo di gioco è un aspetto fondamentale della struttura del software in quanto si concentra nel trattare le logiche di gioco in modo che rispettino il più possibile il concetto del mondo che si è ipotizzato in fase di progettazione.
Per la descrizione del model si fa riferimento generale alla ***figura 1.1***.

Le entità che devono essere gestite sono state divise in due macro-entità : gli oggetti "*inanimati*" nel mondo di gioco quali muri, porte, bottoni e gli oggetti "*animati*" che sono tutte quelle entità di gioco che hanno un ruolo attivo all'interno del mondo quindi il player principale, i vari nemici comprendendo anche il Boss e i proiettili sparati.

Inoltre un'altra differenza tra le entità di gioco è la loro forma, infatti alcune di esse hanno forma rettangolare come ad esempio i muri e la stanza di gioco stessa mentre altri di forma circolare come i proiettili il player e i vari nemici. Per questo si è deciso di creare due tipi di *hitbox*: uno circolare e uno rettangolare.

Con l'aiuto dell'hitbox inoltre si sono potute gestire le *collisioni*, principalmente tra oggetti circolari o tra un oggetto circolare e un oggetto rettangolare. Non è stata gestita la collisione tra due oggetti rettangolari in quanto nel dominio della nostra applicazione questo sarebbe stato impossibile ed inutile.
Sia le entità animate che inanimate ereditano da *GameObject* che è composto quindi da hitbox.
Le entità che estendono da *Inanimated* sono : *Wall*, *Door*, *Button* e *PowerUp*.
I *Wall* sono oggetti di gioco inanimati che insieme alle *Door* formano la struttura base di una *Room*.

La costruzione del mondo di gioco è stata delegata alla classe ***WolrdEnvironment***, utile per l'init del gioco in quanto vengono strutturate le varie stanze di gioco a partire dalla *MainRoom* fino allo *Shop* e alla *BossRoom*.

Per la creazione delle stanze in WolrdEnvironment è stata utilizzata una **factory** come **pattern creazionale**, più precisamente la *RoomFactory*. (***figura 2.2.15***)

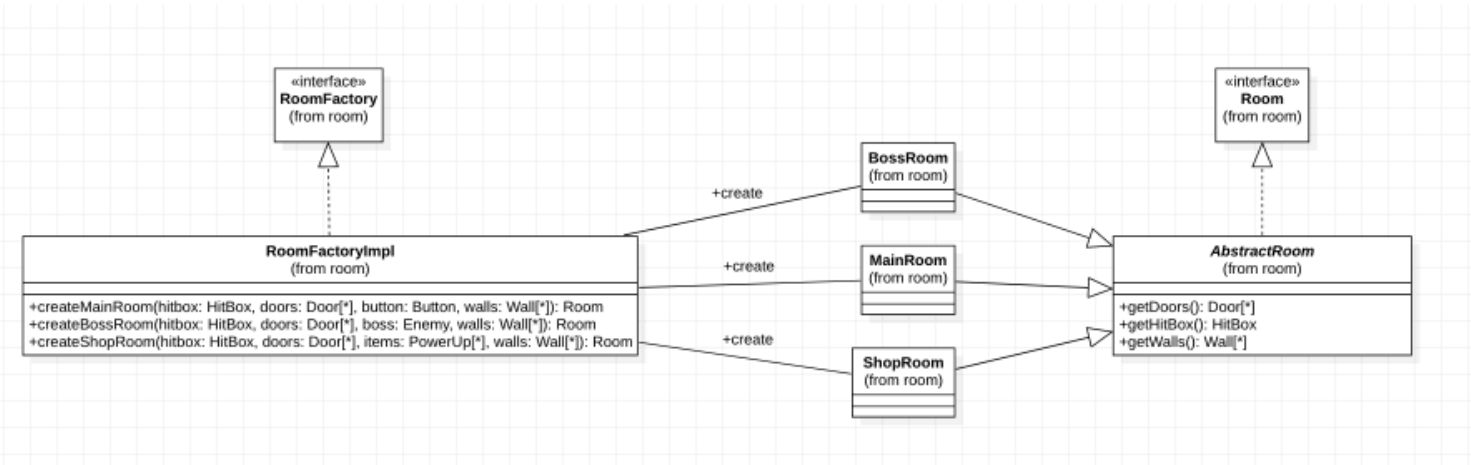

Figura 2.2.15: UML rappresentante la room factory.

Power Ups

Tra le entità inanimate, in particolare per i *PowerUp*, quindi *Heart*, *VelocityUp*, *DamageUp*, *RangeUp*, si è scelto l'utilizzo del **pattern Template Method** per ridurre ripetizioni di codice raggruppando alcuni comportamenti comuni nella classe *AbstractPowerUp*, con il metodo *getSpecificEffect()* che veniva definito solo dalle classi che estendevano quest'ultima. (***figura 2.2.16***).

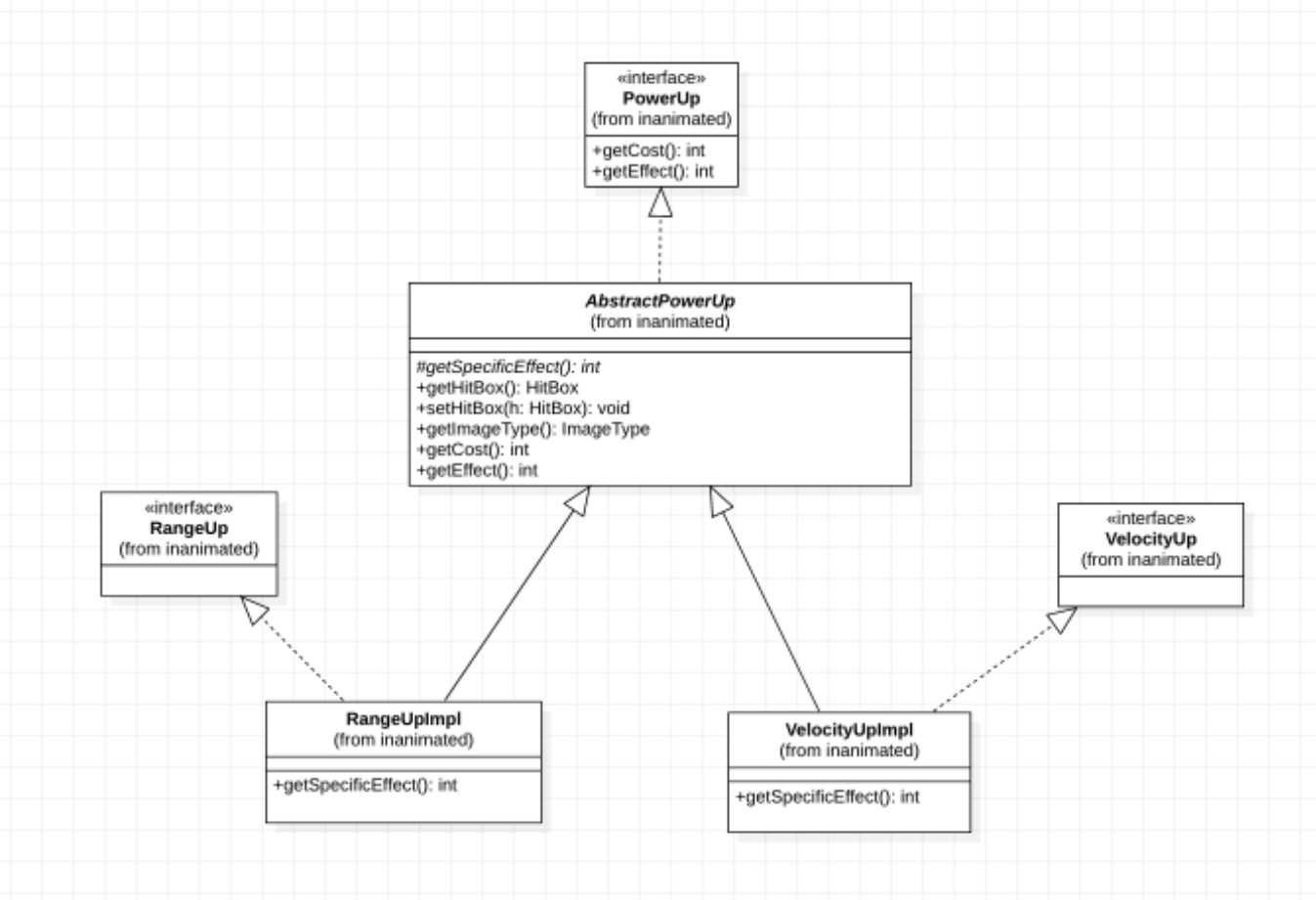

Figura 2.2.16: UML rappresentante il template method realizzato nei power up.

Round Generator

Per la creazione dei round si ha una sola interfaccia comune di *RoundsGenerator*, con una successiva implementazione delle classi *StaticRounds* e *DynamicRounds*. Le classi hanno solo un metodo publico, la *generateMonster()*. Nella *StaticRounds*, nel caso in cui la modalità sia la Normal o la GodMode lo Spawn dei nemici è prefissato mentre nella *DynamicRounds* qualora la mode sia la Survival si avvale di un semplice algoritmo per la creazione di rounds il più casuali possibile.
Per l'implementazione di queste classi si è scelto come **pattern creazionale** il **singleton**, in quanto serve che venga creata una e una sola istanza di queste determinate classi. (***figura 2.2.17***)

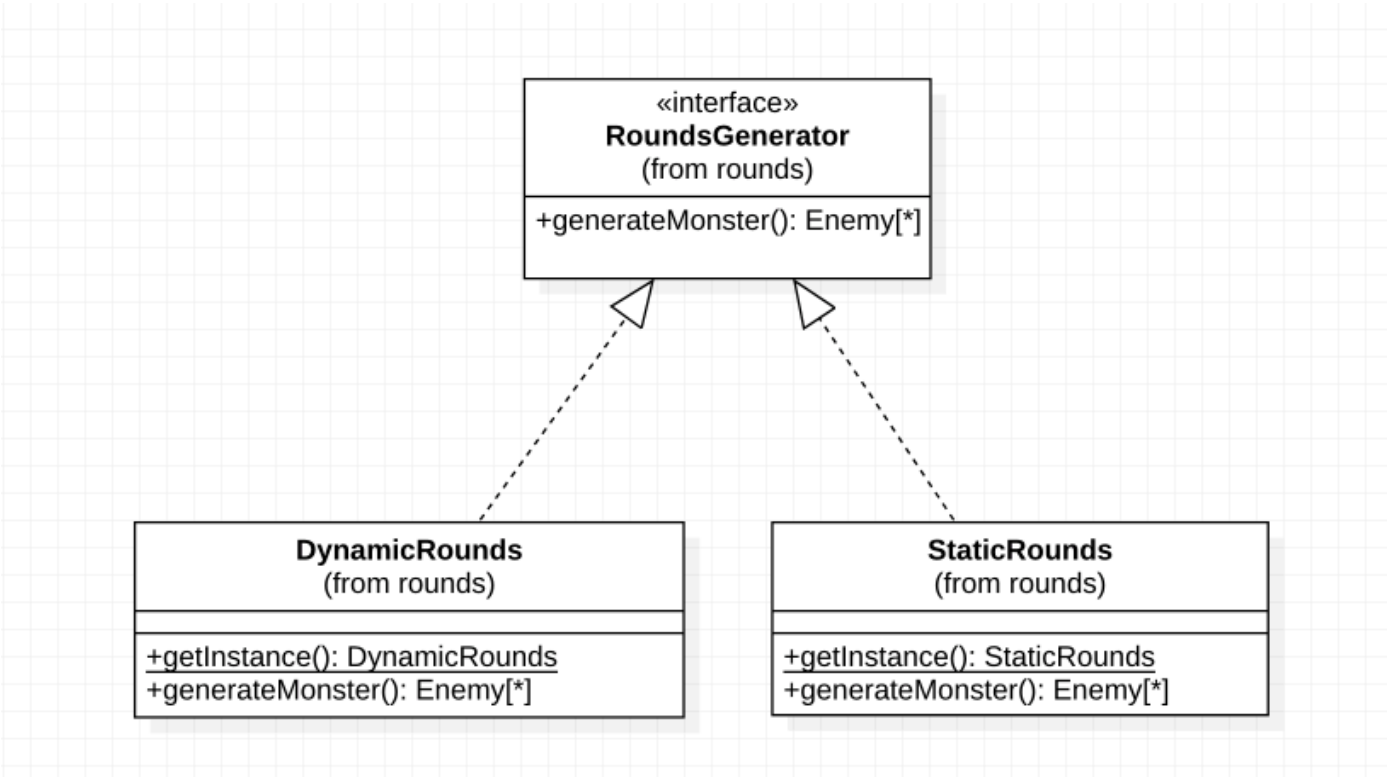

Figura 2.2.17: UML rappresentante la generazione dei round. (**pattern singleton**).

Logica di gioco

Gli Spawn sono gestiti nella *model.utility* e servono principalmente per il posizionamento dei *nemici* e del *player*.
La logica di gioco sviluppata nel *World* è contenuta per la maggior parte nell'update, che serve ad aggiornare le variabili di gioco così come la posizione delle entità quali *player*, *enemy* e *bullet*.
Per il player vengono fornite al model dal controller due liste di comandi che rappresentano i comandi premuti dall'utente.
Una serve per il movimento del player mentre l'altra per la creazione di proiettili.
La creazione dei proiettili, che vengono anche creati periodicamente dai vari enemy quali nemici generali e Boss diventano quindi entità di gioco.
Per l'aggiornamento di tutte le entità animate di gioco è fondamentale il parametro del *deltaTime* che rappresenta il tempo trascorso dall'ultimo update. Mediante questo valore si riesce a rappresentare il movimento delle entità.
Cercando di mantenere un mvc "*puro*" quindi dove il Model non comunica direttamente con la view è risultato fondamentale l'utilizzo della classe *ModelUtility*.
La classe *ModelUtility* che contiene solo variabili e metodi statici e viene aggiornata periodicamente ad ogni update riguardo le principali entità di gioco quali gli oggetti animati, la stanza corrente, il round corrente e gli eventi che vengono generati all'avvenimento di specifiche situazioni di gioco.
La *ModelUtility* viene quindi utilizzata dal controller per ottenere le informazioni necessarie da passare alla view per disegnare sui canvas.

Capitolo 3

SVILUPPO

3.1 Testing Automatizzato

All'interno del progetto sono state predisposte classi di test che, in particolare modo, mirano a testare il corretto funzionamento del mondo di gioco (*model*), del suo aggiornamento, degli spostamenti delle varie entità ed anche della corretta creazione degli oggetti.

Per i test è stato predisposto un package di nome '*test*'.
Al suo interno si possono trovare le seguenti classi:

- *BulletTypeTest*. Questa classe mira a testare la corretta creazione dei proiettili, il loro aggiornamento e il corretto movimento. Essa sfrutta dunque le classi che implementano lo **strategy** per il movimento e la strategia di sparo.
  Questa classe è risultata molto importante per la correzione di errori all'interno del codice, ci ha permesso di ragionare e di apportare modifiche su aspetti che all'inizio erano stati messi in secondo piano o sembravano più semplici del previsto, come ad esempio il sistema di riferimento e il conseguente movimento delle entità.

- *CharacterMovementTest*. Con questa classe abbiamo cercato di testare al meglio possibile il movimento dei personaggi del gioco (es. player, enemies).

- *CollisionTest*. Questa classe di test è stata predisposta per testare le collisioni tra tutti i tipi di HitBox presenti nel gioco. Sono state testate normali collisioni tra due cerchi (rappresentanti player e nemici), tra cerchi e rettangoli (ad esempio tra player e bordi della stanza, tra player e porte, ecc.) ed inoltre i comandi bloccati in cui l'oggetto che collide non può muoversi nel caso di collisioni particolari.

- *TimeTest*. Questa classe è stata predisposta per il testing del thread dedicato al tempo (Time). Durante tale test sono stati predisposti solo semplici controlli riguardo il corretto override di toString(), del corretto aggiornamento del tempo e dello start/stop del thread.

- *ImageLoaderTest*. Questa semplicissima classe dei test permette di controllare il corretto caricamento di alcune immagini.

Per quanto riguarda la view non sono state testate automaticamente le sue funzionalità.
Sarebbero state testate solo poche funzioni, come ad esempio il corretto funzionamento delle operazioni di *push/pop* dello stack utilizzato nel *ViewManager*.

In aggiunta ai test automatici il programma è stato testato anche manualmente al fine di verificare sia determinate situazioni di gioco che per ottenere un feedback visivo su ciò che si stava testando.

3.2 Metodologia di lavoro

Durate la fase iniziale di progettazione, avvenuta durante il periodo delle lezioni, non vi sono stati particolari problemi negli incontri svolti; non ci siamo avvalsi di alcuno strumento di progettazione bensì semplicemente di carta e penna.

Nonostante alcune difficoltà dovute all'inesperienza dei vari membri e alle diverse visioni di ognuno, siamo giunti alla produzione di un primo livello di progettazione (naturalmente solo limitato alle interfacce) piuttosto soddisfacente.

Essendo stato il primo progetto, come facile intuire, col procedere del tempo sono stati necessari cambiamenti piuttosto radicali alla struttura inizialmente pensata ma i concetti base dietro alle interfacce iniziali sono rimasti più o meno gli stessi.

Dopo questa prima fase ognuno ha iniziato a lavorare sulla propria parte, rimanendo tuttavia costantemente in contatto con gli altri membri per chiedere consigli o informarli di eventuali modifiche alla propria parte. Da questo momento in poi si è cercato di utilizzare il DVCS il più frequentemente possibile, anche per piccoli cambiamenti, in modo che fossero tutti documentati.

Sin da subito, si è cercato di arricchire il codice nel modo in cui era stato prefissato, senza la preoccupazione di dover produrre immediatamente una demo. Dal momento che, ciascun componente doveva lavorare su più di una parte di MVC siamo partiti sviluppando il model e il controller lasciando così indietro, almeno inizialmente, la view.

La divisione iniziale era piuttosto equilibrata e non presentava differenze eccessive nella suddivisione del carico di lavoro ma, durante la vera e propria fase di sviluppo, quando si è verificata l'esigenza di consistenti modifiche al codice, alcuni membri hanno dovuto lavorare su parti che inizialmente non gli erano state assegnate. Dunque, seppur tale divisione non è stata pienamente rispettata, le parti assegnateci erano piuttosto indipendenti e, una volta definito ciò che si doveva esporre verso l'esterno tramite l'interfaccia ognuno poteva agire liberamente.

Se durante la fase centrale si verificavano ambiguità o problemi vari, il membro di riferimento per eventuali delucidazioni è stato Vincenzi Mattia.

Il lavoro è stato portato avanti utilizzando costantemente il DVCS. Una volta giunti al punto in cui ogni membro aveva terminato la propria parte si è passati alla risoluzione degli errori di comunicazione tra le componenti che non permettevano il corretto funzionamento del progetto. In tale fase sono stati molto utili anche i test realizzati, che ci hanno permesso di individuare eventuali errori logici.

Per risolvere tali errori, principalmente per via di impegni lavorativi di alcuni membri, hanno svolto il lavoro principalmente Beshiri e Vincenzi, al fine di integrare le varie parti ed attuare le modifiche necessarie. Il lavoro di integrazione e di testing manuale da loro svolto per l'unificazione delle varie parti così da garantire una corretta interazione delle componenti, ha tuttavia messo in evidenza ulteriori problematiche nell'avvio della demo (con conseguente risoluzioni dei problemi da parte degli stessi).

3.3 Note di sviluppo

- *Vincenzi Mattia*

  - **javaFX**. Libreria grafica di java utilizzata per realizzare le varie scene di gioco ed il loro susseguirsi. Per imparare ad utilizzare tale libreria ho utilizzato prevalentemente Internet, ricercando informazioni relative ad aspetti basilari quali una semplice introduzione [1], il funzionamento del sistema sistema di riferimento dei canvas e le funzionalità di translate [2], fino ad argomenti molto più complessi come ad esempio l'utilizzo dei metodi *save()* e *restore()* [3] dei *GraphicsContext* o come scrivere un key listener [4].

  - **Utilizzo dello stack per gestire le scene attive**. Svolgendo ricerche su internet relative al metodo con cui gestire il susseguirsi di più scene, ho trovato un'interessante articolo [5] che descriveva l'utilizzo di uno stack per la gestione di più finestre (*stage*). Il codice e la descrizione, opportunamente riadattati, si sono dunque rivelati molto utili.
    L'idea è quella di mostrare all'interno dello stage la scena che è contenta in cima allo stack e di fare operazioni di *pop e push* a seconda che si debbano susseguire in un breve lasso di tempo.

  - **Stream**. Ho cercato di utilizzarli ovunque fosse possibile, tentando di eliminare costrutti leggermente deprecati come i for/while ecc. Ove possibile, ho cercato di spronare gli altri membri del gruppo al loro utilizzo.

  - **Lambda expression**. Ho utilizzato le lambda ovunque fosse presente un interfaccia funzionale, per rendere il codice più pulito e chiaro.

  - **Proxy pattern**. Ho utilizzato questo pattern strutturale per accedere alle immagini o ai file fxml e nascondere il loro caricamento tramite accesso al file system. Anche in questo caso ho svolto varie ricerche su internet così da informarmi e poter comprenderne l'implementazione. [6] [7]

Oltre agli argomenti e link sopra elencati per capire al meglio il funzionamento dei vari costrutti utilizzati, ho fatto riferimento alla miriade di javadoc presenti su internet.

Non avendo mai sviluppato progetti di questo tipo, con una fase di progettazione alle spalle ed un codice ben organizzato, inizialmente ho seguito le lezioni di "*Game as a Lab*" e studiato il codice fornitoci dal prof. Ricci [8].

Oltretutto sono stati visitati vecchi progetti di vario genere per cercare di capire particolari tipi di pattern e costrutti utilizzati.

Per quanto mi riguarda il codice da me scritto non è stato mai copiato riga per riga. Al più, sono stati ripresi concetti che, adeguatamente rielaborati e personalizzati, sono poi stati inseriti autonomamente.

Per quanto riguarda tutte le competenze trasversali (extra-informatiche) quali trigonometria, movimento delle entità, ecc. ([11] [12] [13]) ho avuto modo di studiare i vari argomenti facendo ampio uso di tutto il materiale presente online. Per capire al meglio i pattern di programmazione ed avere un esempio di mera e semplice implementazione ho invece consultato [14].

Vaienti Andrea

- **Generici**: Ho usato dei generici per impedire che ad un Observer venissero passati oggetti che non fossero un estensione dell'interfaccia Event.
- **Lambda Expression**: Dovunque fosse possibile ho utilizzato le lambda per diminuire le dimensioni del codice. Sono utili e velocizzano molto la programmazione.

Cristurean Denis

Per quanto riguarda lo sviluppo dello strategy dei proiettili devo precisare che ho realizzato soltanto l'interfaccia, i proiettili mirati e quelli a croce e X, i restanti sono stati sviluppati in un secondo momento da Mattia Vincenzi.

Una parte che ha richiesto particolari attenzioni è stata la gestione delle collisioni, infatti ho abbandonato per un momento internet e sono ricorso a carta e penna per capire meglio il problema.

Ho inoltre realizzato il file CSS prendendo come esempio ***moderna.css*** di javaFX per dare uno stile un pò meno monotono alle varie viste.

Beshiri Rei

- **Lambda expression**: utilizzate dovunque fosse stato possibile utilizzare le lambda per diminuire le dimensioni del codice.
- **Abstract**: l'utilizzo delle classi astratte ovunque fosse stato possibile e avrebbe fornito vantaggi.
- **Algoritmo DynamicRounds**: un semplice algoritmo per la generazione dinamica di nemici.

Capitolo 4

COMMENTI FINALI

4.1 Autovalutazione e lavori futuri

- *Vincenzi Mattia*

Mi ritengo soddisfatto del progetto finale ed anche del compito da me svolto. Penso che questo progetto sia stato molto stimolante, portandomi ad imparare molte nuove cose, sia per necessità in fase realizzativa che per interesse personale; questo tipo di prove, seppur molto dispendiose, portano ad un buon livello di consolidamento dei concetti appresi durante il corso. Nel nostro progetto, in particolare nella mia parte occupandomi del movimento delle entità, ho avuto modo di imbattermi anche in argomenti esterni al campo informatico, quali la trigonometria, angoli, sistemi di riferimento ecc. Seppur con iniziale difficoltà tramite la consultazione di materiale esterno, anche questo ha contribuito al consolidamento di concetti totalmente nuovi o al massimo affrontati in altri corsi.

Penso di aver svolto un ruolo importante all'interno del gruppo in quanto, per consigli, problemi o per eventuali dubbi, gli altri componenti si riferivano a me. Per questo e per il fatto di aver messo mano su tutte e tre le parti principali di MVC mi sono ritrovato, seppur involontariamente, molto coinvolto nel progetto. Questo, se da un lato ha i suoi risvolti positivi, come ad esempio avere una visione più chiara ed ampia su tutte le parti del progetto, permettendomi dunque di poter attuare modifiche con maggior semplicità, d'altro canto mi ha caricato di un lavoro piuttosto oneroso, non inizialmente previsto. Mi ritengo tuttavia soddisfatto di quello che sono riuscito a raggiungere e delle competenze apprese.

Questo progetto è stato realizzato puramente a scopo didattico e penso che sarà anche il suo fine ultimo. Dunque, non credo che ci rimetterò mano per ulteriori espansioni.

Inizialmente non avrei mai pensato di essermi addentrato in un progetto così ampio, in quanto pensavo fosse un lavoro più limitato, quindi sono sicuro di avere ampiamente superato il monte-ore minimo di 80-100 prefissatoci dalla consegna.

- *Vaienti Andrea*

Sono complessivamente soddisfatto del risultato finale, in quanto è la prima volta che porto a termine un progetto di queste dimensioni.
Ritengo che il progetto sia cominciato bene sin dalle prime fasi, poiché per più di un mese abbiamo cercato di creare uno schema logico che comprendesse tutti gli scenari possibili. Cosi facendo non ci siamo buttati a capofitto a scrivere codice, che molto probabilmente avremmo dovuto scrivere più e più volte, bensì abbiamo provato a costruire una base solida su cui partire. Ovviamente, essendo per tutti la prima volta, abbiamo dovuto modificare qualche gerarchia in corso d'opera, poiché a progetto iniziato abbiamo riscontrato problemi a cui non avevamo pensato inizialmente.

Nonostante ciò lo schema finale non si discosta più di tanto da quello originario. Il gioco si presterebbe bene all'apporto di nuove funzionalità o modalità, rendendolo sempre più completo. Tuttavia non credo che al momento il software verrà implementato ulteriormente, poichè ritengo necessario l'utilizzo di tools migliori per avere un prodotto finale di qualità maggiore.

Solamente a fini ludici o d'istruzione ci potrebbero essere aggiornamenti.

- *Cristurean Denis*

Sono complessivamente soddisfatto del prodotto finito anche se avrei voluto essere un pò più presente nelle sue fasi finali. Inoltre mi rendo conto che di MVC ho fatto molto V poco M e praticamente niente C, purtroppo in fase di proposta di progetto ho scelto cosa mi sarebbe piaciuto fare e non cosa avrei dovuto fare per rispettare la consegna.

Da circa 2 anni continuo a ripetermi: ”scarica un game engine e prova”, grazie a questo progetto, che per me non rappresenta soltanto una primissima esperienza di lavoro in team ma anche un primissimo tentativo di realizzare un videogame, la voglia d’imparare e fare è più grande che mai.

- *Beshiri Rei*

Mi ritengo abbastanza soddisfatto del risultato finale e di come l'applicazione sia risultata molto simile a quella pensata in fase di sviluppo. Mi ritengo anche soddisfatto anche della parte da me programmata anche se alcune parti si sarebbe potuta svolgere in maniera migliore, o migliorabile, e che a causa di mancanza di tempo non più ritoccata.
Ci sarebbero state molte idee per nuove aggiunte al gioco che però sono state accantonate per dare spazio a miglioramenti tali da rendere il software solido e stabile.

Il progetto attuale credo sia terminato e non penso di rimettere mano sull'applicazione in futuro in quanto pur avendo molte idee, non sarebbero ben sfruttate per colpa delle limitazione dei tool a nostra disposizione in primis il motore grafico.

Tuttavia ritengo comunque utile e stimolante lo sviluppo di progetti così strutturati.

4.2 Difficoltà incontrate e commenti per i docenti

Vincenzi Mattia

Sfortunatamente, essendo stato per tutti i componenti del gruppo il primo progetto da realizzare avendo “carta bianca” e dovendo affrontare tutte le fasi di sviluppo, abbiamo avuto qualche difficoltà nella progettazione iniziale e, di conseguenza, è stato necessario modificare più volte parti di codice. Alcune di questo, oltre all’organizzazione interna del programma, sono dunque state riscritte più volte prima di raggiungere lo stato finale.

Sempre per via della mancata esperienza anche la suddivisione del lavori non è stata realizzata a regola d’arte, infatti, essendo alle prime armi abbiamo inizialmente attuato una divisione inconsci del vero carico di lavoro dietro ogni parte. A tal proposito, sono dispiaciuto del fatto che solo alcuni membri abbiano avuto la possibilità di applicare più pattern e imparare più cose, mentre altri no.

Ultima pecca è stato il fatto che nella fase finale, per via di problemi personali di alcuni membri, ci siamo ritrovati principalmente io e Beshiri a dover risolvere i problemi presenti nel progetto e, di conseguenza, a lavorare anche su parti non effettivamente implementate da noi.

Per quanto riguarda i tool utilizzati non ho avuto particolari problemi. Con l’ambiente di sviluppo (eclipse) mi sono trovato bene, mentre con l’utilizzo del DVCS, una volta superate le difficoltà iniziali, non si sono più verificati grossi problemi.

Vaienti Andrea

Ritengo questo elaborato molto utile, poichè anche se in piccolo, simula una vera esperienza lavorativa, con la collaborazione di diversi programmatori.
Ugualmente al progetto, anche l'utilizzo di Git lo reputo di fondamentale importanza per un futuro professionale.
Il corso è stato molto esaustivo, ricoprendo chiaramente tutti gli argomenti utili alla creazione di questo elaborato. Personalmente avrei preferito avere maggiore delucidazioni sui pattern, concentrandosi più sull'obiettivo che sull'implementazione.
Per il codice riguardante il gameLoop ho fatto riferimento a corsi esterni tenuti dal prof. Ricci riguardanti lo sviluppo di un video-game, poichè non possedevo conoscenze di base sufficienti per la realizzazione di un buon lavoro.
L'ultima difficoltà incontrata durante lo sviluppo è stata una cattiva organizzazione del lavoro, perchè è vero che alcuni membri del gruppo hanno dovuto implementare parti extra al loro compito, tuttavia ciò non è dovuto a menefreghismo, ma alla mancanza di tempo. Per quanto concerne la mia parte, il codice era pronto ad Aprile, mentre allo stesso tempo il codice dei miei compagni era ancora incompleto. Solamente all'esecuzione dell'applicazione, alcuni aspetti del mio codice potevano essere controllati e modificati. Ciò è avvenuto intorno alla fine del mese di Luglio, poichè per motivi di studio, altri elaborati e esami il progetto era stato messo in stand-by. A quel punto per me non è stato più possibile aiutare i miei colleghi come prima, per motivi lavorativi. Nel complesso ritengo questo elaborato come il più istruttivo tra quelli svolti, poichè attraverso le modalità dal prof definite, ho acquisito esperienza e dimestichezza in aspetti che, se non nel mondo lavorativo, non avrei potuto cogliere.

- *Cristurean Denis*

Per quanto mi riguarda, non avendo utilizzato molto in passato i DVCS è stato facile creare piccoli disastri inizialmente.

Successivamente un altro problema si è presentato utilizzando il tool *SeneBuilder*, in particolare nell'utilizzo di risorse esterne come immagini o, nel nostro caso, file *CSS*.

All'interno di SceneBuilder sembrava funzionare tutto alla perfezione, ho poi scoperto che tutti i path contenevano dei caratteri extra che a Java non piacevano.

- *Beshiri Rei*

Sicuramente la difficoltà iniziale sta nell'approccio di un progetto di tali dimensione, essendo il primo progetto da dover strutturare dalla fase di programmazione fino alla fase di debugging e testing da soli.

In fase di sviluppo alcune parti del codice sono state riscritte e ricorrette anche dovuto all'inesperienza nello sviluppo di videogiochi.

Inoltre per vari problemi personali la fase finale e quella di debugging è stata svolta principalmente da me e da Mattia Vincenzi.

Appendice A

GUIDA UTENTE

- **Menù**

All'avvio del gioco è possibile navigare tra la *leaderboard*, i *credits*, la finestra di *help* (che fornisce una spiegazione veloce del gioco) e le *opzioni*.
In quest'ultima schermata è possibile scegliere tra tre modalità di gioco:

1) **Normal**: è la modalità di default in cui parte il gioco, viene selezionata quando nessuna checkbox è spuntata e si tratta della modalità normale. In tale modalità dopo aver terminato tre *rounds* nella stanza principale è possibile accedere allo shop per il possibile acquisto di *power up*, infine si potrà affrontare il *boss* in una stanza dedicata per vincere ed ottenere un numero maggiore di punti.

2) **GodMode**: questa modalità è equivalente alla *Normal* con la differenza che il player non perderà vita nel caso in cui venga colpito da un proiettile nemico. Molto utile in fase di test per verificare determinate situazioni.

3) **Survival**: in questa modalità i nemici vengono posizionati randomicamente. Le ondate si susseguono ad oltranza dopo la pressione del bottone centrale fino a quando il player non perde tutte le vite.

Una volta avviato il gioco è necessario inserire il proprio nickname, che verrà utilizzato come nome nella leaderboard qualora si riesca a fare un punteggio maggiore di quelli presenti.

Per uscire dal gioco è sufficiente premere sulla casella EXIT o in alternativa cliccando sulla croce in alto a destra.

- **In Game**

<u>Comandi:</u>

- **W**: movimento verso l'altro del player.
- **A**: movimento verso sinistra del player.
- **S**: movimento verso il basso del player.
- **D**: movimento verso destra del player.

- **freccia sù**: sparo del proiettile da parte del player verso l'alto.
- **freccia sinistra**: sparo del proiettile da parte del player verso sinistra.
- **freccia destra**: sparo del proiettile da parte del player verso destra.
- **freccia giù**: sparo del proiettile da parte del player verso il basso.

- **Esc**: permette di mettere il gioco in pausa, ripremendo Esc o RESUME si riprende a giocare mentre premendo EXIT si ritorna al menù principale.

- **Fine Gioco**

Nella modalità *normal*, il gioco termina se si perdono tutte le vite con conseguente sconfitta o se si riesce a sconfiggere il Boss, con conseguente vittoria (e bonus aggiuntivo di punti).
Nella modalità *survival* invece non si vince mai, l'unico obiettivo è quello di sopravvivere il più possibile.

Bibliografia

- [1] https://gamedevelopment.tutsplus.com/tutorials/introduction-to-javafx-for-game-development--cms-23835
- [2] https://stackoverflow.com/questions/5789813/what-does-canvas-translate-do
- [3] https://html5.litten.com/understanding-save-and-restore-for-the-canvas-context/
- [4] https://stackoverflow.com/questions/29962395/how-to-write-a-keylistener-for-javafx
- [5] https://www.c-sharpcorner.com/code/2654/javafx-managing-multiple-stages.aspx
- [6] https://www.tutorialspoint.com/design_pattern/proxy_pattern.htm
- [7] https://it.wikipedia.org/wiki/Proxy_pattern
- [8] https://github.com/aricci303/game-as-a-lab.git
- [9] Cerchio-Rettangolo: https://stackoverflow.com/questions/401847/circle-rectangle-collision-detection- intersection
- [10] Cerchio-Cerchio: https://stackoverflow.com/questions/1736734/circle-circle-collision
- [11] https://gamedev.stackexchange.com/questions/36046/how-do-i-make-an-entity-move-in-a-direction
- [12] https://math.stackexchange.com/questions/67026/how-to-use-atan2
- [13] https://stackoverflow.com/questions/9970281/java-calculating-the-angle-between-two-points-in-degrees
- [14] https://www.tutorialspoint.com/design_pattern/index.htm