Volfgit

Bragari Alexandru, Mennuti Luigina Annamaria, Violani Matteo

25 ottobre 2022

Indice

Capitolo 1

Analisi

Il software Volfgit si propone come rivisitazione di Volfied, gioco arcade, single player, anni '80. Commissionato da Unibo, in particolare per il corso di Programmazione ad Oggetti della facoltà di Ingegneria e Scienze Informatiche.

1.1 Requisiti

Lo scopo del Software è di produrre un videogioco all'interno del quale si deve cercare di superare i vari livelli, cercando di ottenere il più alto punteggio possibile. Ogni livello può essere superato:

- diminuendo progressivamente l'area in cui il *boss* può muoversi, costruendo dei muri, fino al raggiungimento o superamento dell'80% di superficie occupata (quindi il 20% disponibile al *boss*). All'aumentare della superficie occupata, corrisponderà un corrispettivo aumento del punteggio ottenuto.

Requisiti funzionali

Il gioco si compone di diversi livelli. La mappa di ogni livello, ovvero l'area complessiva effettivamente giocabile, è rettangolare e mantiene proporzioni fisse.

All'inizio della partita, la mappa di gioco è completamente libera (con libera si intende lo 0% dello spazio occupato) e, al suo interno, sono presenti:

- il boss del livello
- i nemici secondari

- gli item
- il player

La schermata complessiva comprende, esternamente alla mappa:

- il numero del round che si sta giocando
- il punteggio aggiornato in tempo reale
- il più alto punteggio effettuato da un giocatore
- la percentuale di superficie occupata in quello specifico livello
- il numero di vite rimanenti
- un conto alla rovescia che indica la durata rimanente dello scudo del giocatore

Ulteriori requesiti funzionali comprendono:

- Salvataggio dei punteggi e visualizzazione degli stessi
- Schermata per visualizzare e modificare le impostazioni

1.2 Analisi e modello del dominio

Il software dovrà essere in grado di generare dei *livelli* consecutivi e dovrà essere in grado di riconoscere quando il *livello* è concluso per passare a quello successivo. In ogni *livello* sono presenti delle *entità*. Tali *entità* hanno il movimento circoscritto dalla *superficie libera* che, all'inizio del *livello*, coinciderà con la totalità della *mappa* e, successivamente, dai nuovi *bordi* definiti dal *giocatore*. Le *entità* sono descritte in seguito.

Boss

Il *boss* del *livello* è il *nemico* principale. Lo scopo del *boss* è quello di ostacolare lo svolgimento della partita, facendo perdere vite al *giocatore*.

Mosquitoes

I mosquitoes sono i *nemici* secondarie sono di ulteriore ostacolo al *giocatore* si accinge nella costruzione di nuovi *bordi*. Anch'essi hanno la capacità di far perdere la vita al *giocatore*.

Mystery Box

Le *mystery box* sono raffigurate come dei quadrati, con posizione fissa e peculiare per ogni *livello* ed è previsto che i *box* possano essere intermittenti o meno. In caso di intermittenza, questi scompaiono a intervalli regolari, riapparendo poi nello stesso posto.

I bonus che possono essere ottenuti dalle *mystery box* sono:

- P: extra score
- T: freeze del tempo, per cinque secondi
- C: elimina tutti i mosquitoes istantaneamente
- S: aumento temporaneo della velocità

I *bonus* contenuti nelle *mystery box* sono casuali.

Player

All'inizio della partita il *giocatore* ha sei vite, può muoversi lungo la cornice della *mappa* ed è protetto da uno scudo. Quest'ultimo protegge il *giocatore* da contatti con i *nemici* o da attacchi da loro effettuati. Lo scudo ha un tempo limitato di utilizzo, allo scadere del quale, si disattiva. Se il *giocatore* perde l'ultima vita, la partita finisce e il punteggio ottenuto viene salvato nella leaderboard.

La capacità del *giocatore* è quella di modificare i *bordi* lungo cui muoversi, costruendo dei *muri* che andranno ad occupare la superficie della *mappa*, restringendo l'area disponibile al *boss* per muoversi.

Nel momento in cui si vuole creare un nuovo muro, si esce dal bordo e lo scudo viene temporaneamente disabilitato e riabilitato nel momento in cui viene creato un nuovo bordo. Se la *scia*, nel chiudersi, intrappola:

- una *mystery box*, ne acquisisce il bonus
- uno o più mosquitoes, li elimina

Quando il *giocatore* supera il *livello* acquisisce una vita bonus, sempre per un massimo di 6 vite. La scia, nel chiudersi, crea sempre un *muro* in direzione opposta alla posizione del *boss*. Questa è una delle sfide principale, insieme alla gestione di tutte le *entità* e delle loro collisioni.

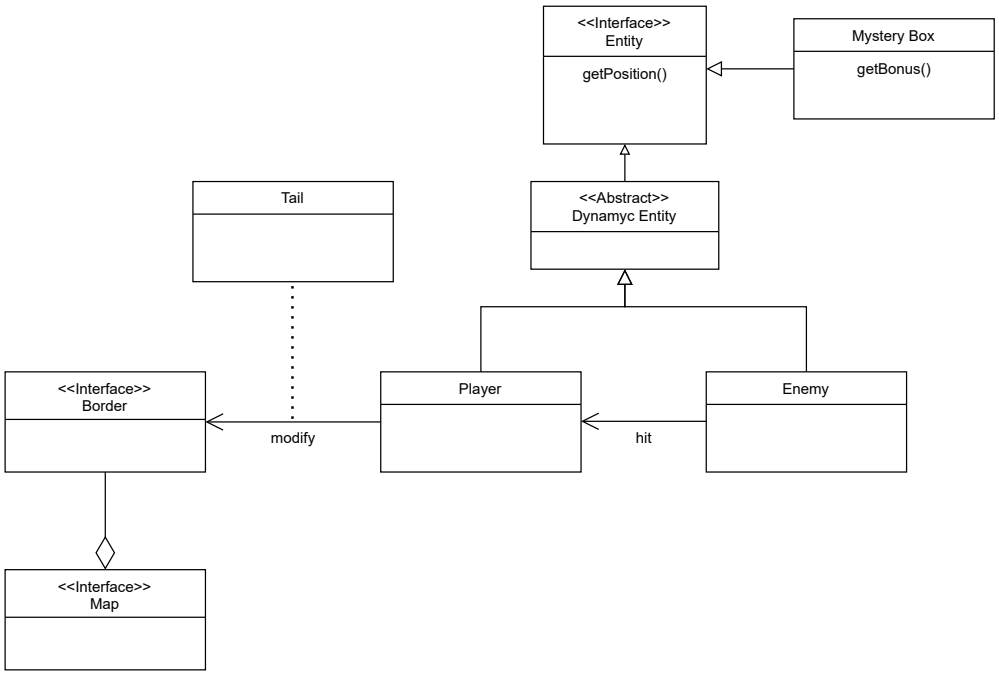


Figura 1.1: Schema UML dell'analisi del problema, con rappresentate le entità principali ed i rapporti fra loro

Capitolo 2

Design

2.1 Architettura

L'architettuta di Volfgit è stata impostata seguendo il pattern architetturale MVC, Model, View e Controller. L'intera gestione della sessione di gioco è stata gestita attraverso il pattern del game-loop, definendo inizialmente gli stati del gioco e le fasi da eseguire ad ogni ciclo. Ad ogni iterazione vengono processati gli input da tastiera, viene aggiornato il model quindi vengono verificate le condizioni di terminazione del livello e successivamente viene aggiornata la View.

2.1.1 Model

Il model è implementato in modo da fornire al controller l'accesso ai dati, gestendo i cambiamenti dei vari stati. Il model, inoltre, è l'entità sulla quale si basa la corretta comunicazione tra il controller e la view.

2.1.2 View

La view è composta dalla Gui che rimane indipendente dal model e dal controller di gioco. Il suo scopo è quello di facilitare l'interazione dell'utente con il model di gioco, fornendo una grafica semplice ef efficace.

2.1.3 Controller

Il controller è quel componente architetturale che ha il compito di controllare e gestire l'ordine e la consequenzialità degli eventi. Ha il compito di gestire i comandi con dell'utente ed aspettare nuove risposte indirizzandole, eventualmente, alla view.

2.2 Design dettagliato

2.2.1 Alexandru Bragari

Model: il software richiedeva che entità di vario tipo interagissero su una mappa comune, in cui il giocatore, operando sulla mappa, possa superare i livelli e proseguire nel gioco.

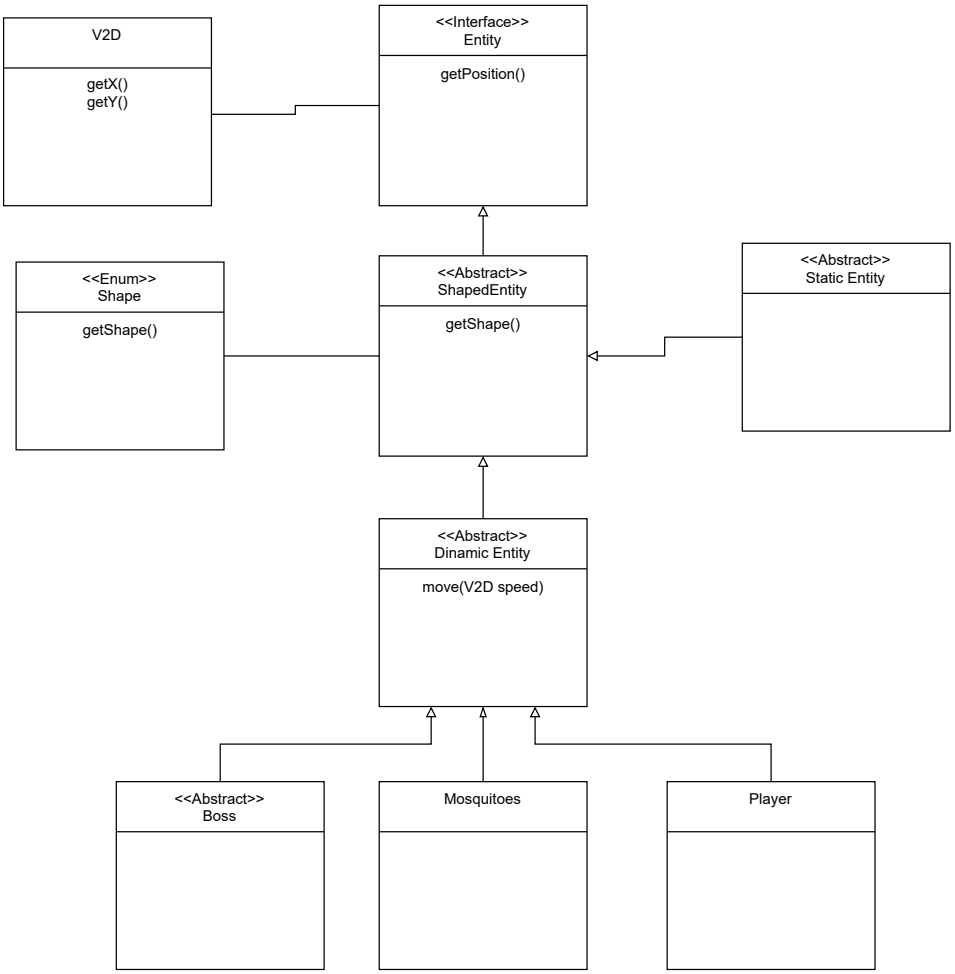


Figura 2.1: Schema UML del design dettagliato delle entità

La soluzione ai requisiti è stata quella di creare una gerarchia di entità, partendo da quelle più generale **Entity** e mano a mano specializzando le classi che la implementano seguendo il pattern **Strategy**. La divisione principale è quella tra **DynamicEntity** e **StaticEntity**, grazie all'utilizzo del pattern **Template** è stata creata la classe **Player** che estende **DynamicEntity** e in particolare il metodo **move()** che ogni entità che si muove ha, ma in maniera più articolata seguendo delle logiche interne che differiscono dalla maggior parte di classi che hanno esteso da **DynamicEntity** quali **Boss** e **Mosquitoes**. Tutte le entità hanno in comune l'avere una propria posizione sulla mappa di gioco, in coordinate cartesiane, incapsulate nella classe **V2D** all'interno della quale sono anche presenti i metodi matematici più utilizzati sui vettori. Oltre alla posizione, la caratteristica comune a tutte le entità utilizzate per costruire il software è il possesso di una forma, definita in un'enum, che può essere quadrata oppure circolare.

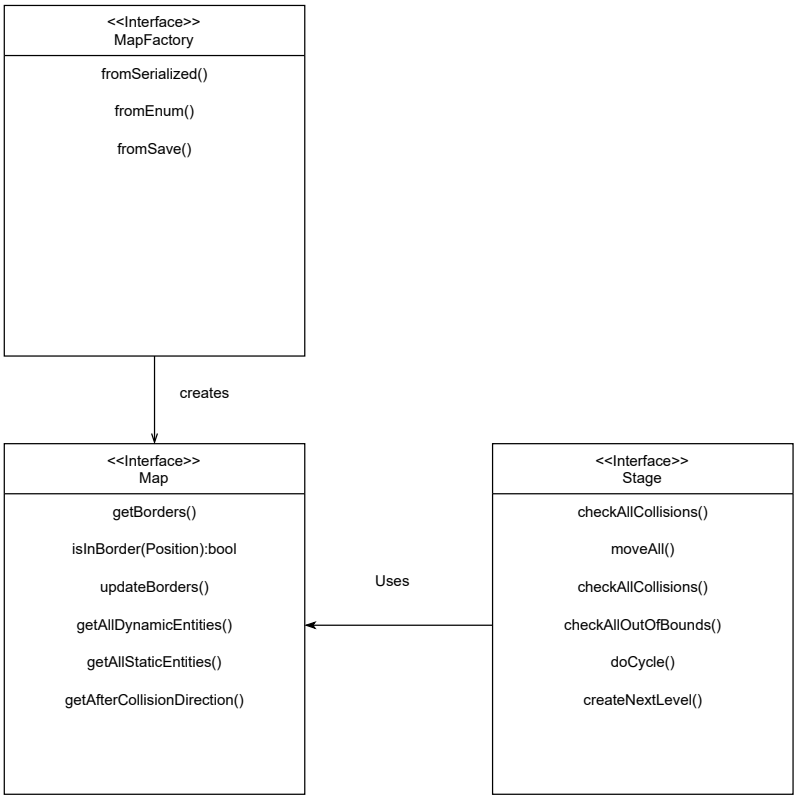


Figura 2.2: Schema UML del design dettagliato di Stage e Map del Model

Considerando tutte queste caratteristiche l'interfaccia **Map** mette a disposizione tutti i metodi che servono per far interagire le entità tra di loro, ad esempio se hanno una posizione che collide, il metodo **getAfterCollisionDirection()** calcola il nuovo vettore velocità che l'entità dovrà acquisire. L'interfaccia Map è stata costruito con un tipo generico, quello che verrà utilizzato per le posizioni e le velocità e nella realizzazione della classe che implementa **Map**, in particolare **MapImpl** è stato utilizzata la classe **V2D** per rappresentare i vettori.
La classe **Map** contiene anche i **bordi** della mappa di gioco, rappresentati come una sequenza di punti adiacenti, un **Boss**, fondamentale per l'applicazione di algoritmi per la modifica dei bordi ed la classe **Player**. I metodi e le logiche messi a disposizione da **Map** vengono utilizzate dalla classe **StageImpl** che implementando **Stage** con l'utilizzo di **V2D** (generico in **Stage**), rappresenta la classe che forma il gioco nella sua controparte model del pattern MVC. All'interno salvati il livello che si sta giocando e la percentuale di mappa catturata dal giocatore, che viene utilizzata come condizione di vittoria per passare da un livello all'altro. Inoltre contiene i metodi che fanno interagire tutte le entità presenti sulla mappa quali **moveAll()**, **checkCollsions()**, **checkAllOutOfBounds()** e **doCycle()**.
Quest'ultimo è fondamentale perché chiama in una sequenza logica e ordinata tutti i metodi necessari affinché le entità svolgano un ciclo di gioco. Molto comodo per avere un "tempo" discretizzato interno che è facilmente manipolabile dal controller di gioco nel caso si volesse ridurre o velocizzare il gioco stesso.
Utilizzando questo tipo di design è garantita una buona estendibilità, poiché se si volessero aggiungere altri tipologie di entità sarà facile comprendere quali interfacce e classi devono implementare/estendere per essere facilmente integrate con le altre. Oppure è possibile cambiare il tipo di dato utilizzato per i vettori e basterà reimplementare i metodi dell'interfaccia **Map**.
Per quest'ultima è stata proposta una **factory** che crea una mappa a partire dal livello inserito, sono messe a disposizione varie tipologie per produrre le mappe, quella maggiormente sviluppata utilizza la serializzazione della classe **MapImpl** che poi verrà letta e data in gestione a **Stage**. Questo sistema è molto comodo in un futuro in cui si vorranno implementare dei salvataggi a metà partita, uno degli obbiettivi facoltativi non raggiunti.

A livello di **view**, visto che è stata utilizzata la libreria JavaFX che facilità l'utilizzo del pattern MVC, vi era la necessità di gestire l'input da parte dell'utente che si concretizzi, e la soluzione è stata l'implementazione di una classe che implementi EventHandler di JavaFX.
Quest'ultima chiamata **KeyEventHandler** seguendo la falsariga del pattern template, fa l'ovverride del metodo **handle()** e per ogni tasto digitato, trasforma l'evento in un KeyAction, un tipo di dato che ogni controller che lo riceve avrà la responsabilità di trasformare in una chiamata interna. Internamente, ogni KeyEvent (ovvero l'evento di input di un tasto catturato da JavaFX) è mappato ad uno specifico KeyAction, presente in una classe KeySettings. Questo è utile nel caso si volessero fare delle modifiche alle impostazioni dei tasti. Il passaggio tra view, controller e model è stato piuttosto di successo.

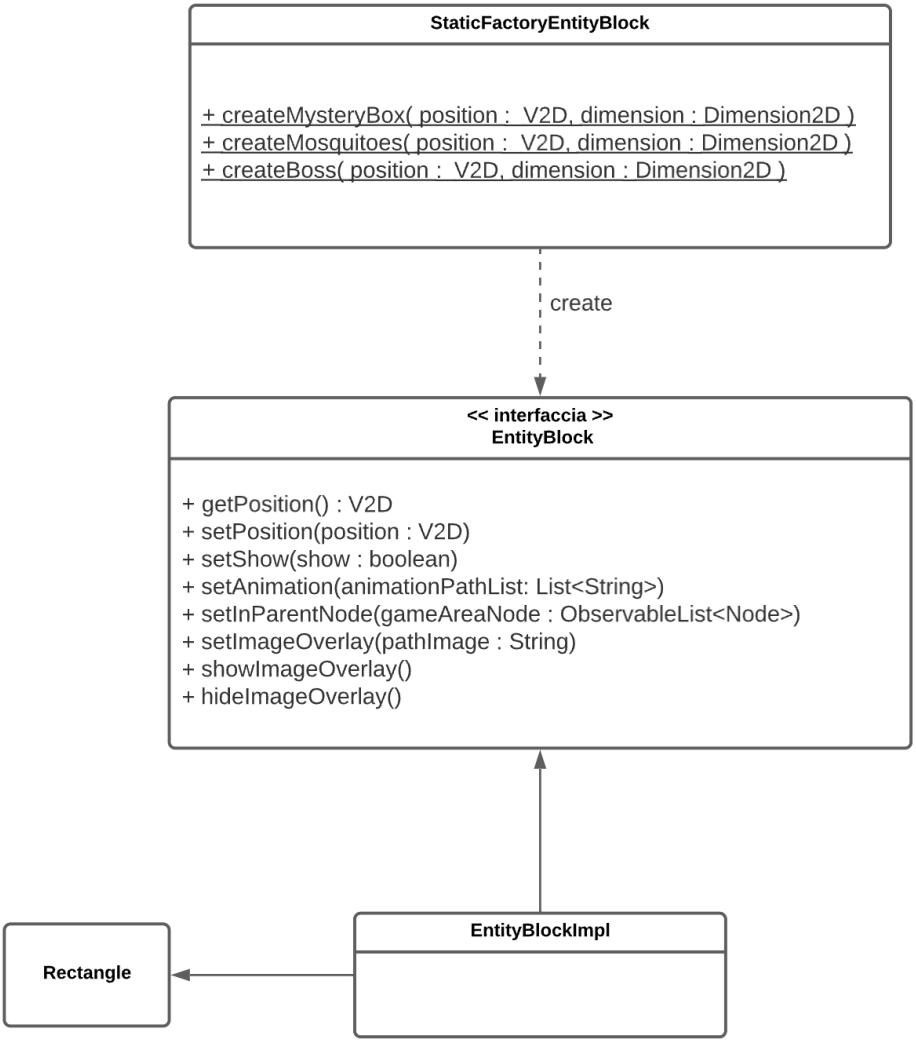
2.2.2 Mennuti Luigina Annamaria

Implementazione del componente grafico delle entità di gioco

La classe `EntityBlock` permette di creare un componente grafico nell'area di gioco, tramite una posizione e una dimensione e fornisce, inoltre, la possibilità di avere un'animazione grafica. Lavorando con la libreria JavaFx, estende la classe `Rectangle` in modo tale da permettere l'effettiva visualizzazione.
La creazione di un'unica classe permette l'aggiunta di nuove funzionalità grafiche a tutte quelle entità che la implementano, come per esempio la possibilità di avere un'animazione.
Ho adottato il pattern Static Factory per la specializzazione della creazione di:

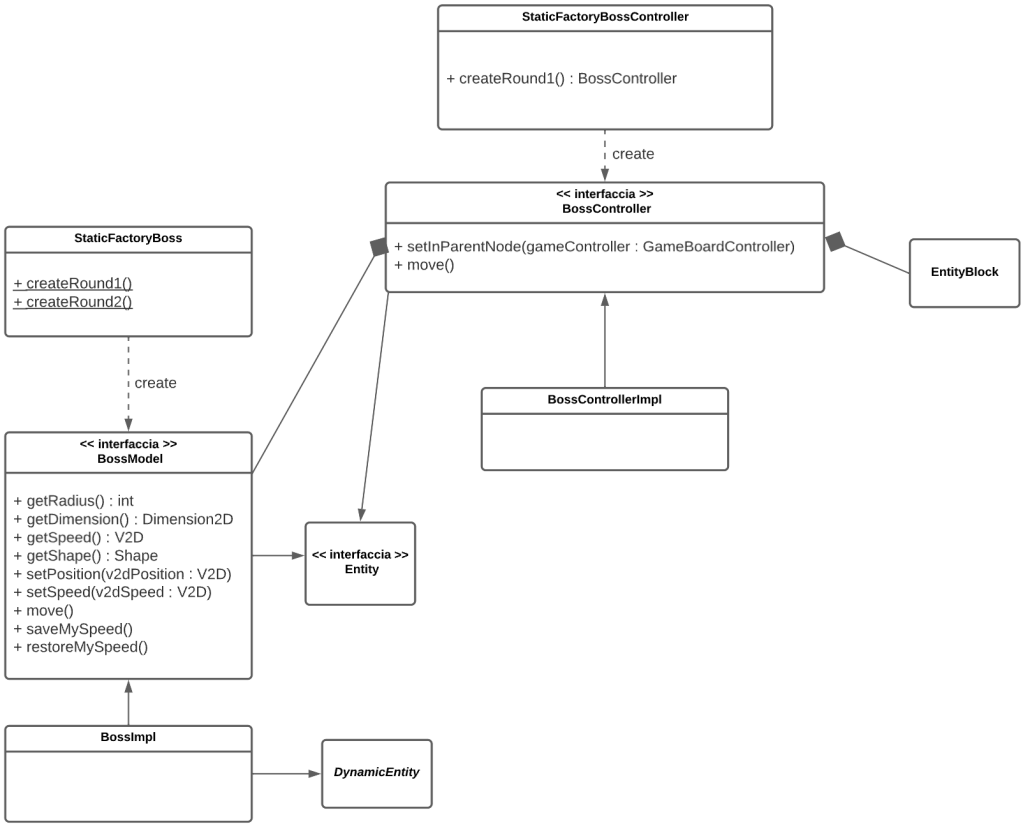
- mystery box
- boss
- mosquitoes

in modo tale da permettere la creazione futura di eventuali nuove entità grafiche.
Essendo stata utilizzata questa classe per la realizzazione delle view di tutti gli oggetti da me creati non ripeterò in maniera dettagliata la creazione delle view nei successivi paragrafi.


StaticFactoryEntityBlock
+ createMysteryBox(position : V2D, dimension : Dimension2D)
+ createMosquitoes(position : V2D, dimension : Dimension2D)
+ createBoss(position : V2D, dimension : Dimension2D)
create
<< interfaccia >>
EntityBlock
+ getPosition() : V2D
+ setPosition(position : V2D)
+ setShow(show : boolean)
+ setAnimation(animationPathList: List<String>)
+ setInParentNode(gameAreaNode : ObservableList<Node>)
+ setImageOverlay(pathImage : String)
+ showImageOverlay()
+ hideImageOverlay()
EntityBlockImpl
Rectangle

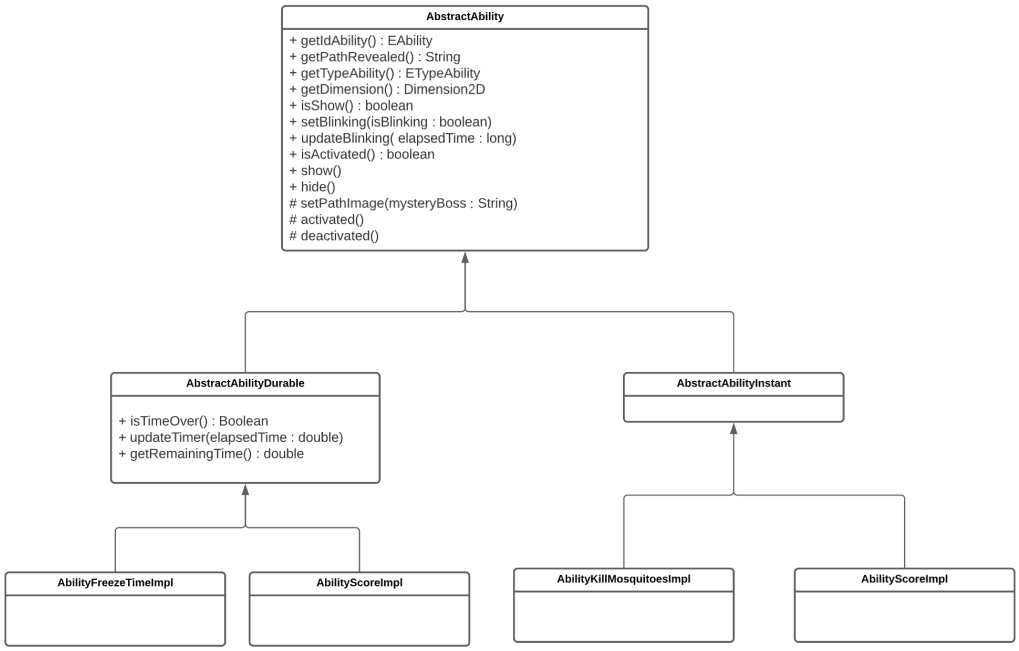
Boss

Il boss è stato creato secondo pattern MVC.
Il model contiene tutti i dati che servono per manipolare il boss. La classe `BossImpl` estende *DynamicEntity* per riuscire a gestire le collisioni.
Per la creazione del modello è stato adottato il pattern Static Factory, specializzando la creazione in base al livello.
L'interfaccia grafica è resa possibile grazie ad `EntityBlock`.
Il controller incapsula i dati del modello e view, permettendo di avere una corrispondenza biunivoca tra model e la grafica. Il controller è stato creato, a sua volta, tramite Static Factory per rispecchiare la creazione secondo i livelli.

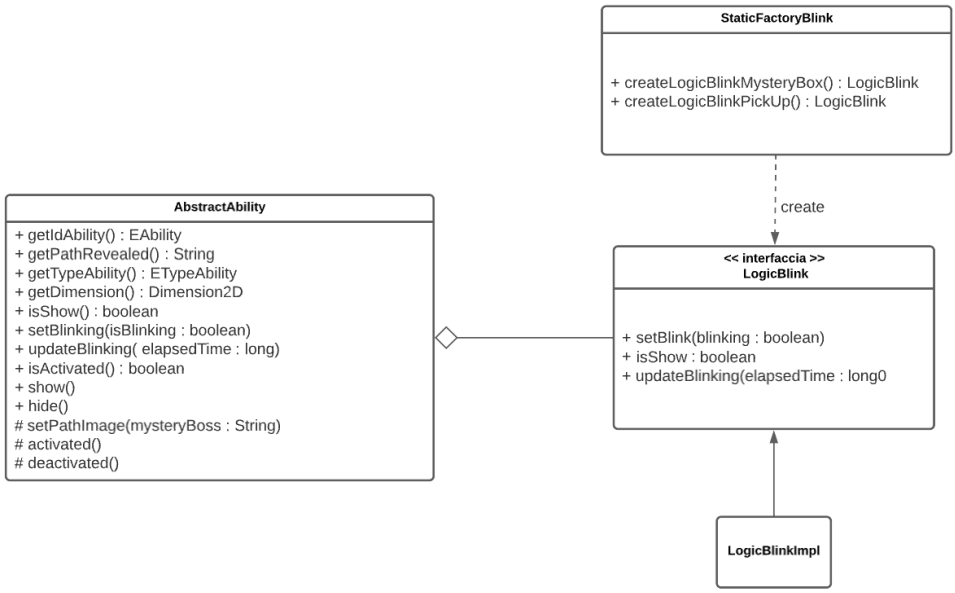


Mystery Box Model

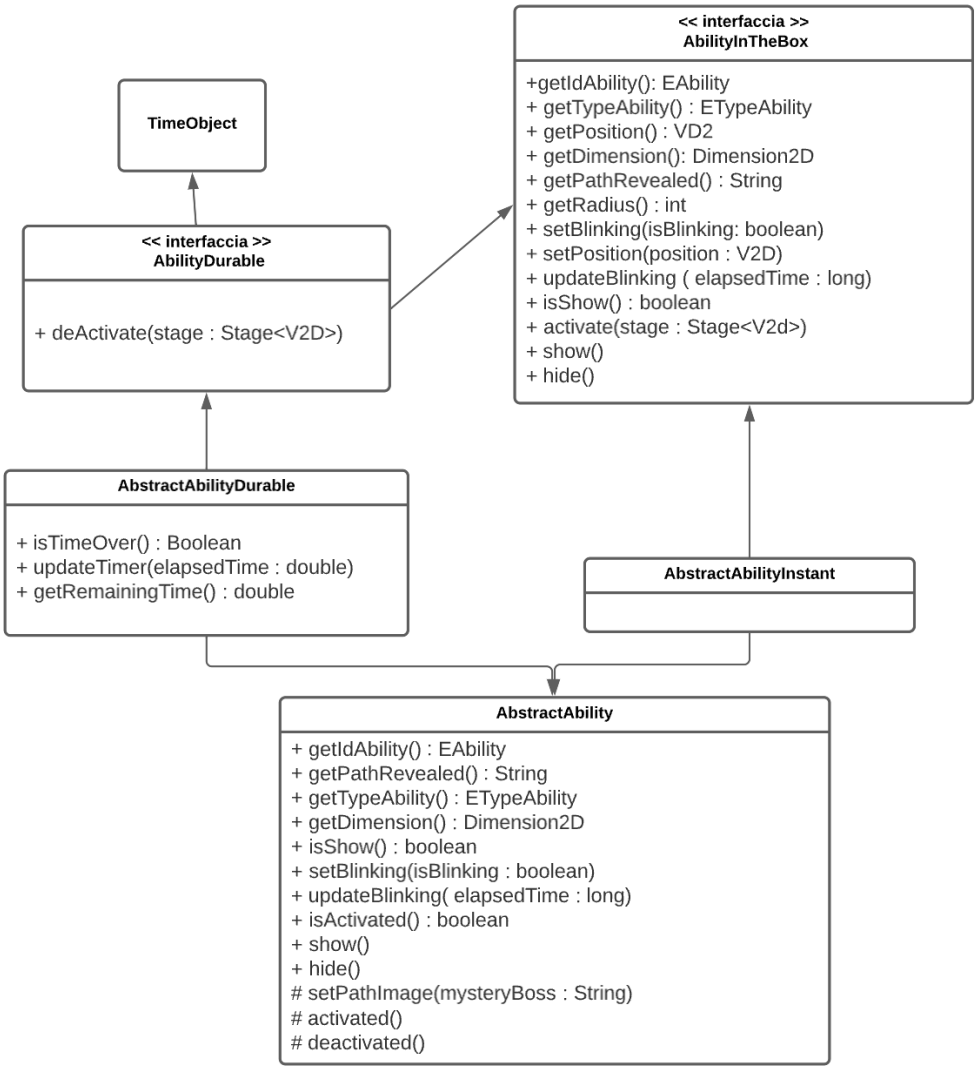
Il model delle mystery box è gestito tramite une gerarchia di classi, dove alla base vi è l'`AbstractAbility`, classe che contiene tutti i campi e le funzionalità base di ogni mystery box. Essa estende *StaticEntity* che riprende il funzionamento delle collisioni. Essendoci due tipologie di mystery box, temporizzate o meno, la specializzazione è stata fatta attraverso l'uso delle classi `AbstractAbilityDurable` e `AbstractAbilityInstant`.



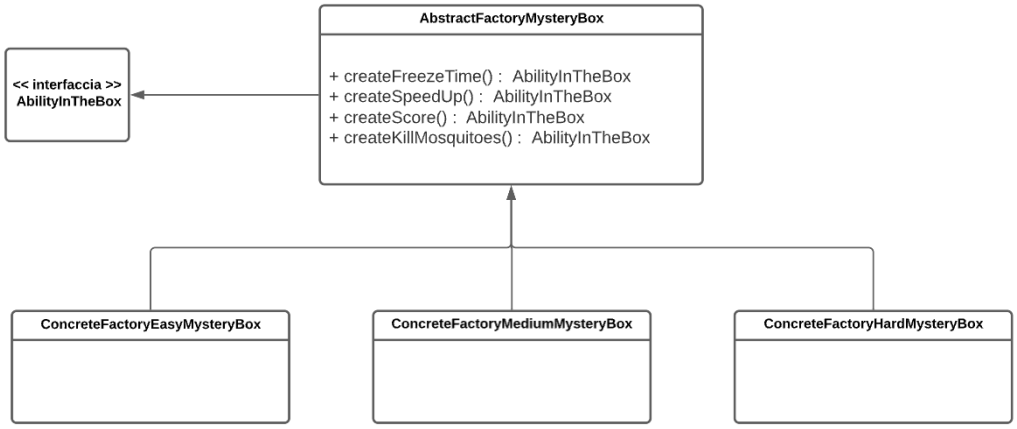
Vi è anche la gestione del blink temporizzato sempre tramite l'uso di TimeObject. Questo è visibile nell'interfaccia AbstractAbility. In aggiunta, vi è la gestione del blink anche quando la mysterybox è stata catturata, cambiando immagine.



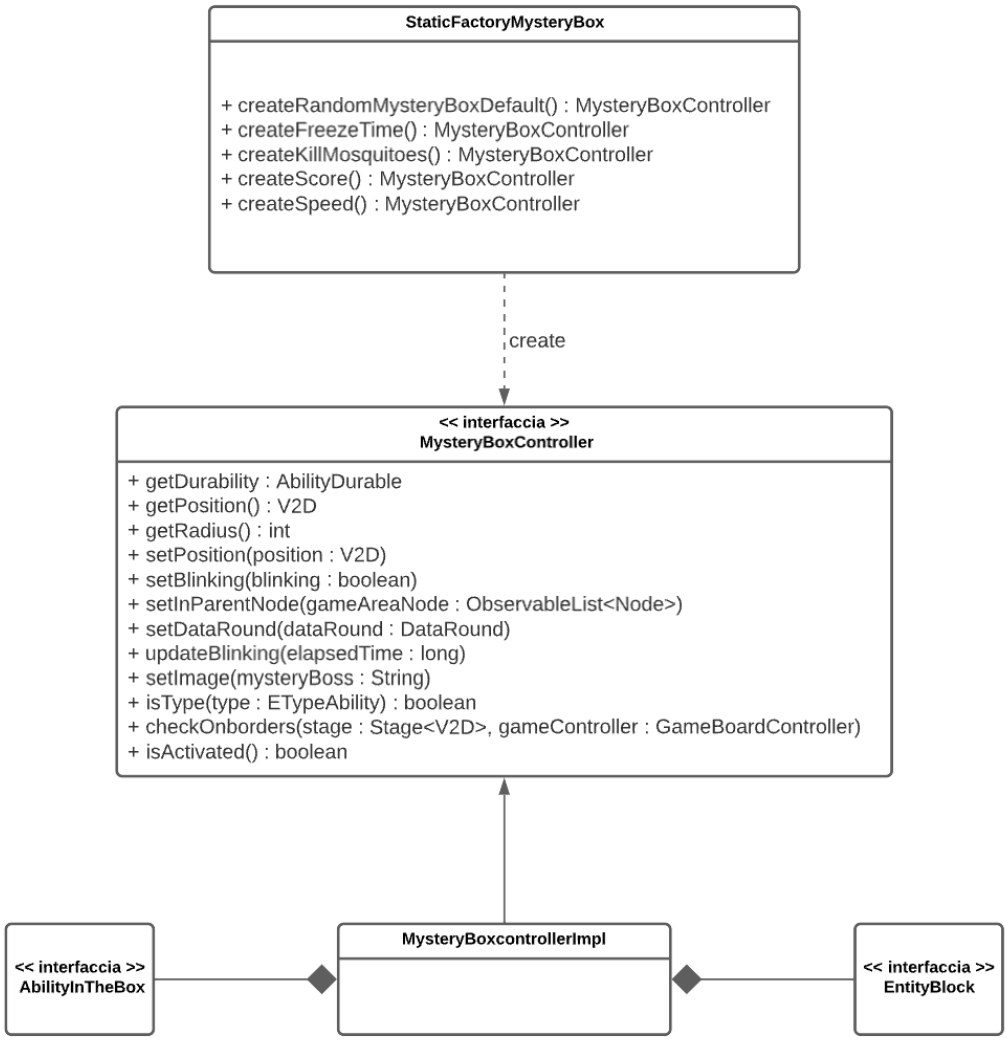
L'interfaccia padre di ogni mystery box è `AbilityInTheBox`. Vi è una seconda interfaccia che la eredita, `AbilityDurable`, che è stata creata per quelle funzioni temporizzate non necessarie alle mystery box con effetto istantaneo e, perciò, implementata solo dalla classe `AbstractAbilityDurable` e dalle relative implementazioni. Inoltre, l'interfaccia `AbilityDurable` estende TimeObject permettendo il riuso della gestione del tempo degli oggetti.



Per la creazione delle mysterybox è stato adottato il pattern Abstract Factory che permette di inserire oggetti fra loro correlati, in questo caso la correlazione è fatta basandosi sulla difficoltà. Incapsulando la responsabilità e il processo di creazione di oggetti prodotto, si rende il client indipendente dalle classi effettivamente utilizzate per le implementazioni dell'oggetto. Questo permette una futura implementazione di nuove mystery box.

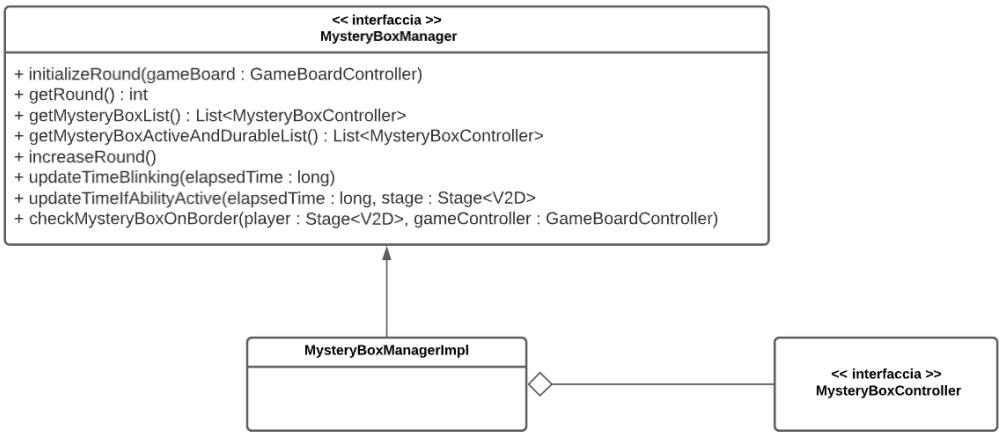


Il controller della mystery box è composto dal model e dalla view, riprendendo i componenti suddetti. Tale controller riesce a manipolare i dati e attivare le abilità lato model, rispecchiando la grafica relativa. La creazione è stata effettuata tramite lo Static Factory.

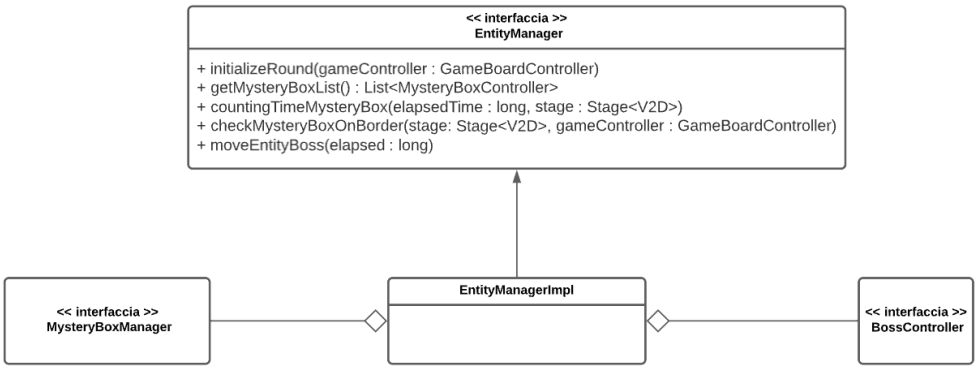


Manager

Per la gestione delle mystery box in gioco ho creato la classe `MysteryBoxManager` che incapsula tutte le mystery box, gestendo il loro funzionamento per tutta la durata dei gioco e nei vari livelli, segregando, così, il loro comportamento.



Per tutte le mie entità di gioco ho creato un `EntityManager` che incapsula la gestione delle mystery box tramite l'interfaccia MysteryboxManager e il comportamento del boss, tramite il suo controller. In questo modo abbiamo un incapsulamento e una segregazione delle loro funzionalità, avvalendosi delle loro interfacce e dando la possibilità di una scalabilità per la creazione di nuove entità (ad esempio, per la creazione di nuovi nemici con il proprio manager).



2.2.3 Matteo Violani

In questa sezione viene trattato il design del Player e di come viene aggiornato il movimento e tutto ciò che riguarda il design del ViewManager e delle View (in particolare quelle di transizione e dei punteggi di mia competenza).

Player e Shield

Per quanto riguarda la modellazione del player dall'analisi è richiesto possegga uno Shield a tempo attivabile in determinate condizioni e una Tail con la quale vengono tracciati i nuovi bordi.

In una ottica di future estesioni si è pensato di modellare lo Shield come una estensione di un `TimedObject`, un oggetto che mantiene al suo interno un timer. In questo modo è possibile creare altri `Shield` o oggetti di altro tipo che possiedono un timer interno e inoltre si è separata la logica specifica dello Shield da quella del timer.

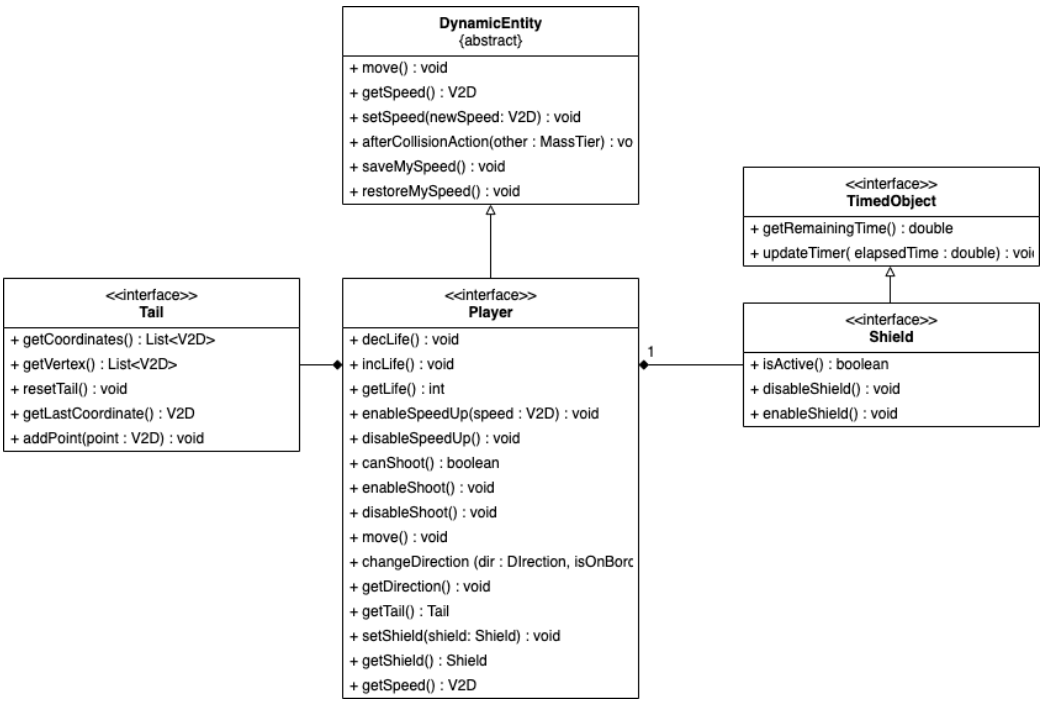


Figura 2.3: Schema UML del Player e dei suoi componenti che lo caratterizzano: Shield e Tail

Per quanto riguarda il movimento del player è stato utilizzato il pattern **Command** in modo da poter muovere il player rispettando i tempi di aggiornamento del dominio dettati dal refresh rate del gioco nel game loop.

Andando nello specifico il `GameController`, che implementa l'interfaccia `SceneController`, riceve le azioni associate ai tasti premuti (`KeyAction`) e aggiunge nella coda dei movimenti il comando di modificare la direzione del player. Nella fase di elaborazione dell'input quindi viene eseguito il comando e rimosso dalla coda.

Come si può vedere dallo schema in figura 2.4 si ha che il `GameController` funge da CommandInvoker mentre il Player è il Receiver che riceve ed esegue il ConcreteCommand definito a runtime in base all'input ricevuto dal `GameController` in quanto osservatore dei `KeyEvent` per mezzo dell'interfaccia `SceneController` (quest'ultimo è stato omesso dallo schema). Il comando che viene eseguito cambia solo nei parametri che si passano al metodo che si chiama sul Player, quindi il GameController può essere considerato come Client in quanto si occupa della creazione del ConcreteCommand e di impostare il ricevitore che in questo caso è sempre il Player.

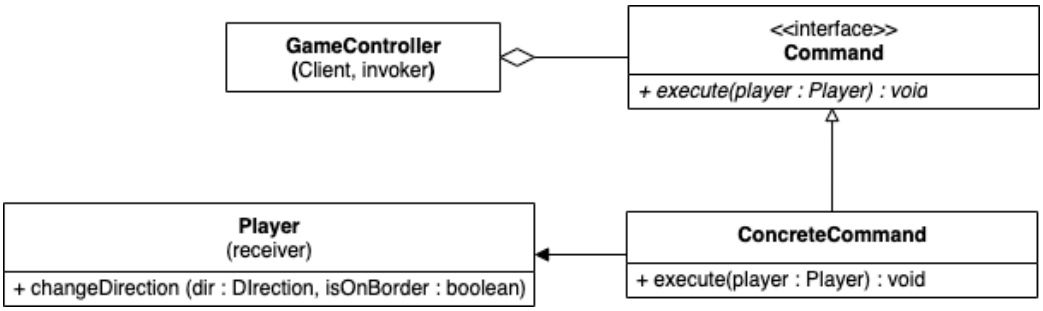


Figura 2.4: Schema UML del pattern Command applicato al Player e GameController

View e ViewManager

View Lo schema mostrato in figura 2.5 mostra il design di una qualsiasi View che può essere gestita correttamente dal ViewManager e quindi essere visualizzata e reattiva agli input. Ogni ViewImpl possiede un riferimento al suo `ViewController` e allo `SceneController` specifico per il comportamento della view.

Schermata dei punteggi Per la realizzazione della schermata dei punteggi, come per tutte le altre schermate, si è cercato di applicare il pattern

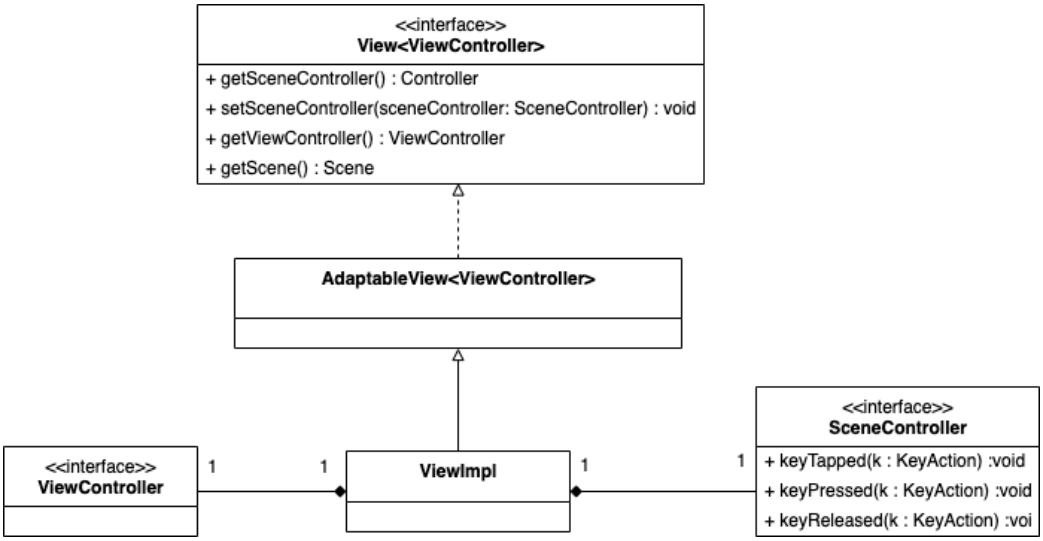


Figura 2.5: Schema UML del design della View

MVC. Il design assunto è come quello descritto in precedenza, con la presenza del controller specifico `LeaderBoardController` che implementa l'interfaccia `SceneController` e si occupa di recuperare i dati dei punteggi memorizzati (model) e di passare alla View in modo da poterli renderizzare.

ViewManager Data la quantità di diverse `View` è stato necessario creare una classe che gestisse la navigazione fra queste. La classe `ViewManager` si occupa appunto di mostrare l'ultima View aggiunta e di delegare il comportamento allo SceneController associato in seguito ad un KeyEvent, . Ogni volta che si vuole mostrare una View il `ViewManager` imposta al `KeyEventHandler` il nuovo SceneController la quale riceve le `KeyAction` associate al `KeyEvent` passate dal `KeyEventHandler`.

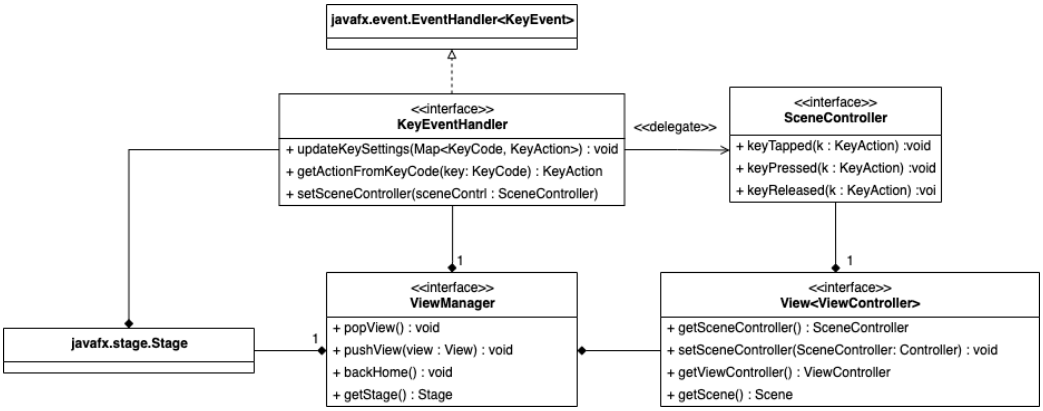


Figura 2.6: Schema UML del ViewManager

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per effettuare i test JUnit è stato configurato ed utilizzato Gradle e sono stati eseguiti i seguenti test:

- `levelGeneratorTest` Viene verificato se il salvataggio e lettura dei dati dei livelli generati avviene correttamente;
- `DynamicEntityTest` Venogno testati i movimenti e la correttezza della velocità;
- `MapImplTest` Test per la verifica del calcolo della percentuale di occupazione della mappa, del movimento delle entità dopo una collisione, di aggiornamento dei bordi, e di modifica della coda;
- `PlayerTest` Testa il corretto movimento, modifica delle vite e attivazione dello speedup del player
- `TestEntityCollisions` Test delle collisioni fra entità di Shape differenti;
- `PathImageMysteryBoxTest` Test di verifica delle path delle immagini dei bonus.
- `PathSoundTest` Test di verifica delle path dei suoni;

3.2 Metodologia di lavoro

La parte di analisi e modellazione del dominio è stata svolta insieme e nel corso dello sviluppo ha subito rari modifiche

3.2.1 Alexandru Bragari

Sicuramente mi sono concentrato in maniera prioritaria al parte model di MVC ma non completamente.
Ho dato il via anche ad alcuni cose di view e controller anche se poi non sono state portate avanti da me, ad esempio AdoptableView che poi è stata integrata nel ViewManager.
Anche l'idea di gestire il sistema di controller come una macchina a stati; ho partecipato più che altro agli albori e poi quando il processo era ben avviato ho lasciato quella parte di software per concentrarmi all'implementazioni di tutte le dinamiche nel model, creazione di nuovi bordi della mappa, calcolo della percentuale occupata, collisioni e spostamenti.
Ho cercato di impostare il model in modo che sia facile interfacciarsi dallo Stage del model (che effettua i cicli interni in maniera automatica e separata dai controller) al Controller di stato e di avere una progressione temporale ben definita e consistente. Inoltre ho implementato l'EventHandler di JavaFX che riceva gli input da tastiera.

Riguardo al DVCS è stato utilizzato git in maniera non molto avanzata ma solida. Essendo in 3 abbiamo lavorato praticamente sempre su un unico branch con relativamente pochi merge confilct e mai gravi.
Questo dal mio punto di vista è un buon indice di un lavoro distribuito bene. So che ognuno di noi in qualche occasione ha creato dei branch a parte per il testing di alcune implementazioni un po' più laboriose.

3.2.2 Mennuti Luigina Annamaria

Ho iniziato il lavoro creando piccolo schema che rappresentasse in maniera sommaria il lavoro che sarei, poi andata a svolgere. Partendo dalla creazione grafica del layout di gioco ho potuto capire come sviluppare integrando la libreria JavaFx, dando una base per poter sviluppare le entità di gioco ed i vari layout. I primi approcci con le interazioni tra java e javafx, specialmente con i file fxml, è stata inizialmente dispersiva e confusionaria. Successivamente, avendone pressochè capito la dinamica sono passata alla creazione delle entità grafiche e ho implmentato tutte le sue funzionalità. Per la creazione delle mystery box ho inizialmente creato una classe base, rendendomi poi conto delle varie differenze che andavano a far divergere aspetti distanti, ma contenuti nelle stessa classe. Così facendo sono riuscita a creare una struttura gerarchizzata che organizzasse la creazione in modo flessibile ed estensibile. Questo comportamento è stato poi riutilizzato per la creazione dei suoni. Una volta che l'architettura MVC era costruita ho cercato di renderla il più

possibile scalabile, segregando ogni componente dentro un'interfaccia e isolando la creazione di tali componenti nelle factory (static e abstract).
Per il lavoro in team ho fornito elementi finiti, come EntityBlock per tutti i componenti grafici del gioco, SoundManager per la gestione degli effetti sonori e EntityManager per la manipolazione di tutti gli oggetti da me creati.

3.2.3 Matteo Violani

Durante lo sviluppo è stato utilizzato il DVCS git cercando il più possbile di lavorare su un branch distinto per la creazione di nuove feature o di quelle più complesse.

- Inizialmente ho sviluppato le interfacce e le classi relative al Player, allo Shield e alla Tail (vg.model.entity.dynamicEntity.player) e alla loro gestione lato controller e model.

- Successivamente mi sono concentrato sulla creazione del game loop all'interno della classe GameController quindi della gestione dei diversi stati di gioco del game-loop con le relative schermate e alla gestione dei movimenti del Player secondo gli input ricevuti da tastiera.

- Infine ho sviluppato il ViewManager per la gestione delle schermate e lo sviluppo del codice lato View per mostrare correttamente i dati (nel caso della schermata dei punteggi, Leaderboard), il player nella mappa e l'aggiornamento dei contatori di gioco (score, livello, vite, timer Shield, percentuale mappa).

3.3 Note di sviluppo

3.3.1 Alexandru Bragari

- Progettazione con generici
- Uso di lambda expressions
- Uso di `Stream`
- Uso di `Optional`
- Uso di gradle come build system
- Libreria di terze parti: JavaFX, Google Guava (per gli optional serializzabili)

Ho implementato una specie di algoritmo di Ray Casting per la creazione dei bordi, nulla di particolarmente complesso ma neanche così banale sul momento, visto che ho scoperto dopo che esisteva un algoritmo con anche non poca teoria attorno. Anche se è stato stimolante è un buon esempio su come è meglio documentarsi anziché inventare di nuovo la ruota perché avrebbe semplificato di molto il lavoro.

3.3.2 Mennuti Luigina Annamaria

Feature avanzate

- Uso di lambda expressions
- Uso di `Stream`
- Uso di JavaFx

3.3.3 Matteo Violani

- Generici e generici bounded nelle classi `Controller` `AdaptableView` e nelle interfacce `Command` e `View`
- `Lambda` nel definire i Command in GameController
- `Stream`
- `Optinal` invece dei null.
- JavaFX

Capitolo 4

Commenti finali

In quest'ultimo capitolo si tirano le somme del lavoro svolto e si delineano eventuali sviluppi futuri.

4.1 Autovalutazione e lavori futuri

4.1.1 Alexandru Bragari

Come autovalutazione personalmente, mi ritengo soddisfatto. Se mi dovessi dare un voto dire tra il 24 e il 26. La progettazione è stata piuttosto solida e il lavoro totale è andato piuttosto liscio.
mi dispiace che la qualità, almeno nel mio lavoro sia diminuita progressivamente sopratutto negli ultimi giorni a causa della pressione della scadenza, alcune cose le avrei potuto fare meglio.
Il progetto dubito verrà portato avanti ma oggettivamente parlando sarebbe facile riprenderlo ed estenderlo creando un gioco effettivamente fruibile.
Inoltre è pur sempre la prima volta che faccio parte dello sviluppo di un vero e proprio software e devo dire che avendo già seguito Ingegneria del Software del terzo anno, è stato molto più facile se dovessi immaginare di svilupparlo senza avere quel tipo di concetti (abbiamo effettivamente fatto degli UML prima di iniziare a scrivere codice!). Uno particolare spazio lo voglio dare ai test di JUnit che ho trovato estremamente comodi, ora capisco molto meglio le potenzialità del test-driven development, di cui ho solo sentito parlare da colleghi della magistrale e citato in laboratorio.

4.1.2 Mennuti Luigina Annamaria

Ritengo di aver fatto complessivamente un buon lavoro, ma anche che ci sia un buon margine di miglioramento dell'applicativo. Durante la stesura del

codice ho sempre cercato di appoggiarmi ad un'interfaccia, piuttosto che a testing. Probabilmente, avendo organizzato meglio il tempo ci sarebbe stata la possibilità di aggiungere ulteriori feature alle mie entità, ma come versione base di un primo progetto arcade mi ritengo soddisfatta.
Ritengo importante una migliore capacità di comunicazione che ha reso per me e per gli altri componenti del gruppo, a volte, difficile continuare a sviluppare. Credo che questo rientri negli scopi del progetto, permettendo di rendermi conto che una definizione prioritaria, in maniera dettagliata, del lavoro da svolgere sia necessaria per evitare successive ed ulteriori perdite generali di tempo in cui, poi, per non perdere del lavoro fatto, sono state adottate delle soluzioni che non mi hanno pienamente soddisfatto. D'altro canto la capacità di superare questa difficoltà in ogni caso, permettendo di portare a termine un lavoro finito ed in tempo, mi ha comunque gratificata.

4.1.3 Matteo Violani

Sono abbastanza soddisfatto del risultato complessivo ottenuto. Sono consapevole che il codice che scritto sia migliorabile sia in termini di organizzazione, design e efficienza. Nello specifico il `GameController` ritengo sia ancora troppo vincolato alla implementazione della view e sarebbe stato più opportuno separare ulteriormente la gestione delle schermate senza il bisogno della creazione delle schermate e settaggio del viewManager da parte del GameController. Penso di aver utilizzato correttamente il pattern utilizzati senza forzare il design. Non penso porterò avanti il progetto ma molto probabilmente lo migliorerò con tecniche migliori e perfezionamenti per soddisfazione personale.

Appendice A

Guida Utente

All'avvio del gioco si presenta il Menù Principale (Figura A.1). Occorre muoversi utilizzando le frecce e per confermare premere invio (enter).

Si può avviare una partita, vedere la classifica dei giocatori, modificare le impostazioni oppure chiudere il gioco.

Figura A.1: Menù Principale

Una volta avviata una partita (A.2) per muovere il personaggio occorre utilizzare i tasti frecce della tastiera. Se si desidera mettere in pausa occorre premere P, se ripremuto si riprende la partita. Per terminare anticipatamente

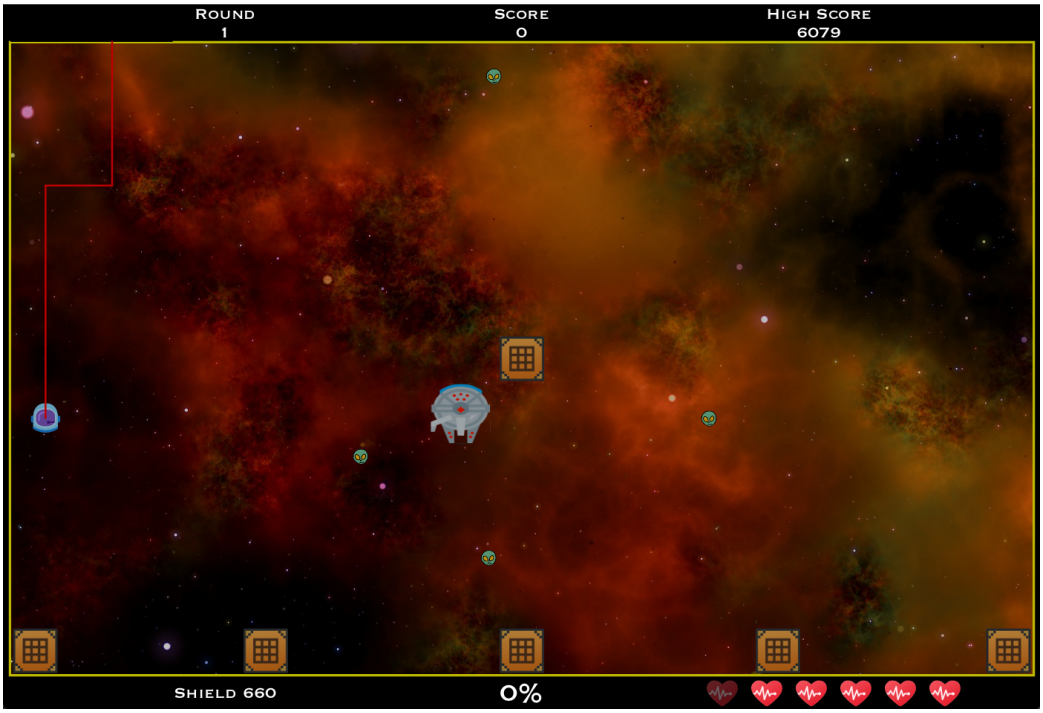


Figura A.2: Schermata di gioco

la partita bisogna premere il tasto ESC e a quel punto muoversi con le frecce destra e sinistra quindi premere invio (ENTER) (A.3). Al gameover (A.4) viene richiesto l'inserimento del nome del giocatore quindi per confermare occorre premere invio (ENTER) e si ritorna al menù principale.

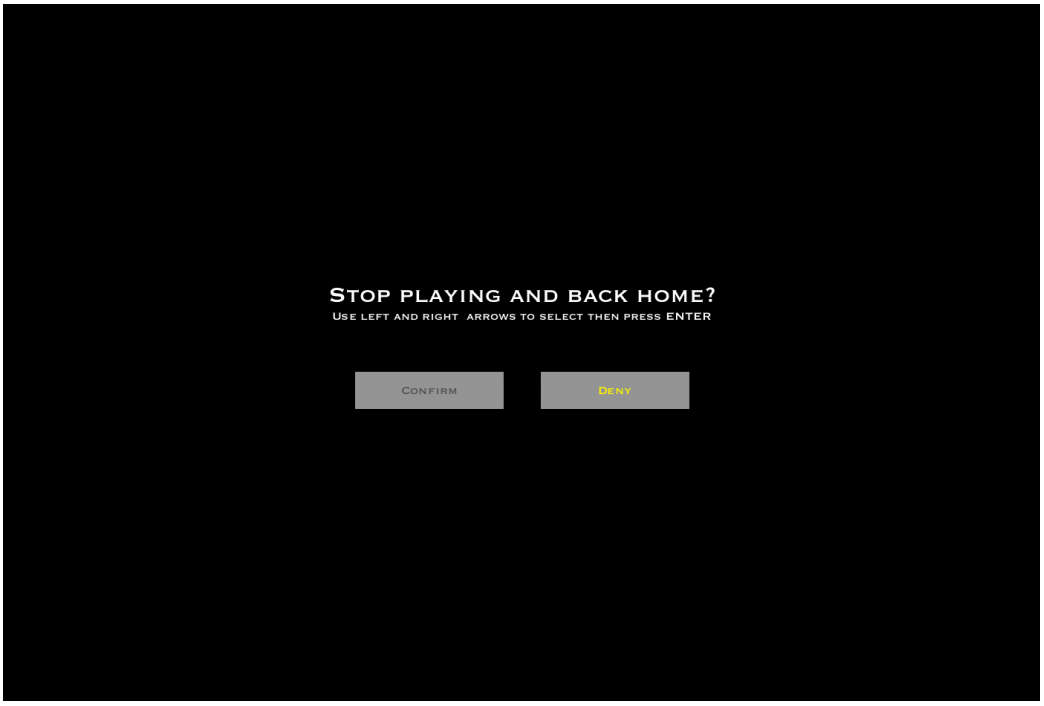


Figura A.3: Schermata di conferma uscita

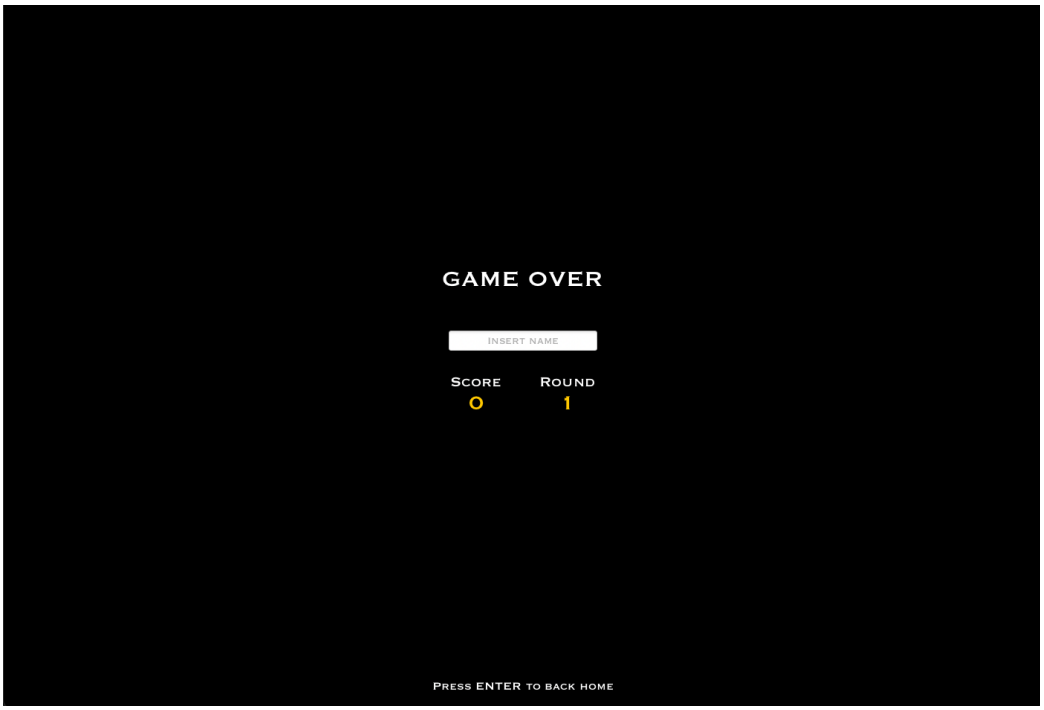


Figura A.4: Schermata di gameover

Appendice B

Esercitazioni di laboratorio

B.0.1 Alexandru Bragari

B.0.2 Luigina Annamaria Mennuti

B.0.3 Matteo Violani

- Lab 04: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101137`
- Lab 05 : `https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101137`
- Lab 07 : `https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p101137`