BomberOne

Francesco Agostinelli
Tommaso Brini
Luigi Borriello
Gustavo Mazzanti

24 giugno 2021

Indice

Capitolo 1

Analisi

Il software ha lo scopo di replicare in una formula rivisitata lo storico videogame arcade NEO Bomberman (`https://it.wikipedia.org/wiki/Neo_Bomberman`).
Il player, nei panni del Bomberman avrà l'obbiettivo di distruggere tutte le casse presenti nella mappa entro lo scadere di un certo limite di tempo.
Ad ostacolare il Bomber saranno presenti dei nemici, anch'essi da eliminare.

1.1 Requisiti

Requisiti funzionali

- Creazione di una mappa 2D nel quale saranno presenti svariate entità, con le quali il protagonista potrà interagire.
- Implementazione di due diverse difficoltà, che andranno a modificare l'esperienza di gioco:
  - **Difficoltà Easy**: I nemici non respawnano e si muovono casualmente.
  - **Difficoltà Hard**: I nemici respawnano e se vedono il Bomberman, iniziano ad inseguirlo.
- Realizzazione del Bomberman in grado di spostarsi all'interno della mappa e di rilasciare le bombe.
- Realizzazione di nemici dotati di intelligenza, e in grado di colpire corpo a corpo il Player.
- Realizzazione di svariati PowerUp (potenziamenti per il Bomber).
- Creazione di elementi distruttibili, nonchè le casse di gioco.

- Realizzazione del respawn dei nemici.
- Fornire all'utente la possiblità di salvare il proprio score, in una classifica globale.
- Realizzazione di un menu nel quale l'utente potrà navigare e scegliere se lanciare una partita, oppure accedere alla classifica dei punteggi.

Requisiti non funzionali

- Prestazioni in grado di fornire una buona esperienza di gioco.
- Disposizione casuale delle casse all'interno della mappa.
- Possibilità da parte dell'utente di scegliere i comandi di gioco e l'aspetto del Bomber.

1.2 Analisi e modello del dominio

Una volta avviata la partita(**GameMatch**) il giocatore verrà collocato nei panni del **Bomber** all'interno della mappa(**World**) nella quale saranno presenti casse (**Box**), nemici(**Enemy**) e muri indistruttibili(**HardWall**).
Il tipo di partita è definito dalla **Difficulty**, la quale, per l'appunto, determinerà le sue specifiche caratteristiche relative al numero delle casse generate, alla velocità dei nemici e alla rigenerazione di quest'ultimi.
Il Bomberman per vincere la partita dovrà aver distrutto, tramite il rilascio di bombe, tutte le casse e tutti i nemici presenti, entro lo scadere del tempo limite, con la possibilità di avvalersi dei **PowerUp** rilasciati casualmente da alcune casse in grado di modificare le sue caratteristiche e delle sue bombe.
Ogni partita conterrà varie informazioni relative ad essa fra cui:

- Lo stato del Bomberman e le caratteristiche delle sue Bombe.
- Il Timer, utilizzato per determinare la fine della partita al suo scadere.
- Lo stato del **World** e degli oggetti presenti in esso.
- L'User associato, ovvero nome e punteggio relativo alla partita, che verrà poi registrato nella classifica una volta terminata.

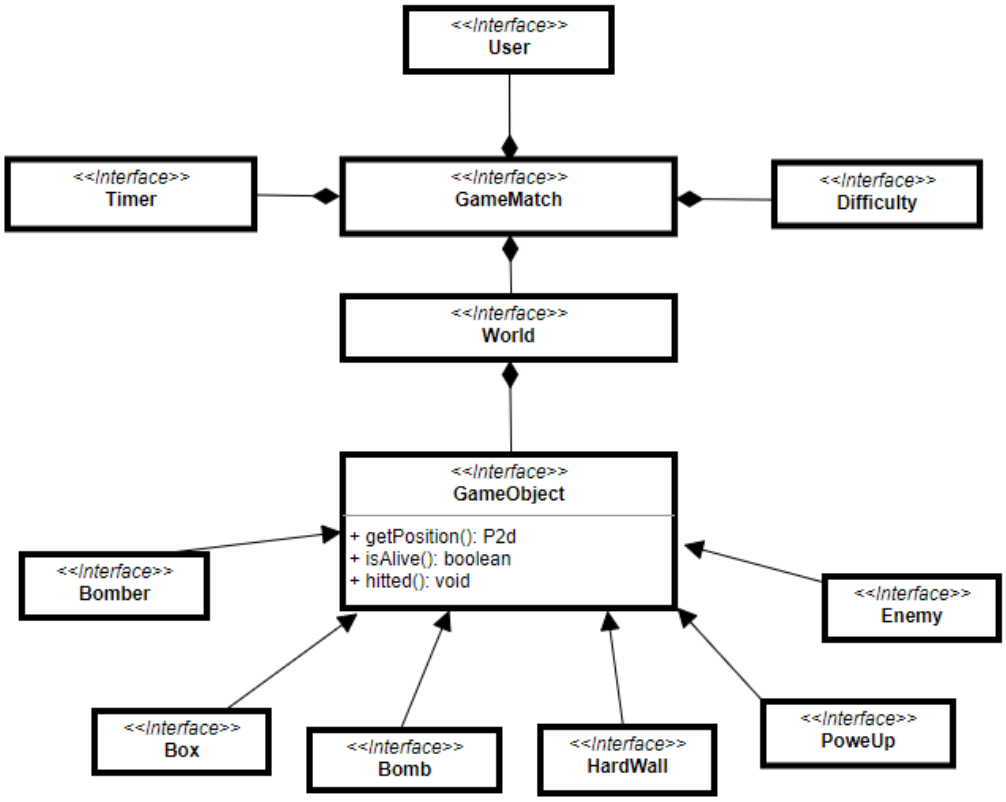

Figura 1.1: Schema UML dell'analisi dei requisiti indicante le principali entità e relazioni che comporranno l'applicazione.

Da sottolineare come tutte le varie entità coinvolte nel *World*, siano considerate come oggetti di gioco(*GameObject*), aventi delle proprietà in comune fra cui:

- La posizione: coordinate all'interno del World.
- Il loro stato (vivo o morto).
- La possibilità di essere colpite.

Le difficoltà primarie saranno:

- Far interagire il protagonista con i nemici e l'ambiente di gioco gestendo al meglio le varie collisioni e i vari eventi.
- Realizzare due difficoltà del gameplay caratterizzate da dinamiche differenti.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione del nostro Software abbiamo optato per l'uso di una architettura *MVC(Model-View-Controller)*.

Il **Model** corrisponde allo stato attuale dell'applicazione e si occupa di fornenirne metodologie di accesso ai vari Controller. Il Model è rappresentato dall'entità GameModel e, GameModelImpl corrisponde alla relativa implementazione.

Le **View** rappresentano le varie schermate dell'applicazione e gestiscono sia la logica di quest'ultime, ma anche l interazioni con l'utente, che vengono segnalate poi ai relativi Controller.
Sono state realizzate con *JavaFX* e rappresentano nell'architettura FXML, gli *"fx:controller"*. Infatti tutta la parte relativa allo stile è stata realizzata con il linguaggio FXML in file separati.

I **Controller** si occupano principalmente di effetturare modifiche al Model e fornire alle View eventuali informazioni riguardanti l'applicazione, fungono dunque da "tramite" tra *M & V*.

Ogni View possiede un proprio controller, eccetto *Home, Credits e Rules*, le quali, non dovendo apportare modifiche al Model, condividono tutte lo stesso Controller(*HomeController*), che possiamo definire "innocuo".

Nello specifico, il *Gameloop* è contenuto nel Controller relativo alla partita(**MatchController**), il quale per l'appunto ogni *tot* millisecondi, andrà

ad aggiornare lo stato del Model e della View associata(**MatchView**).

La scelta di isolare i Controller di MVC dalla libreria grafica utilizzata, permette quindi, di poter cambiare quest'ultima, senza dover poi ritoccare le parti relative al Model e ai Controller, bensì solo la parte riguardante le View.

Per una scelta progettuale sono state realizzate delle implementazioni "basilari" sia del Controller che della View, le quali vengono poi estese ogni volta con l'aggiunta di funzionalità, a seconda dello scopo specifico.
Abbiamo optato per questa soluzione per cercare di migliorare l'estendilità del codice, nel caso in cui, in futuro ci fosse la necessità di aggiungere eventuali schermate/funzionalità all'applicazione.

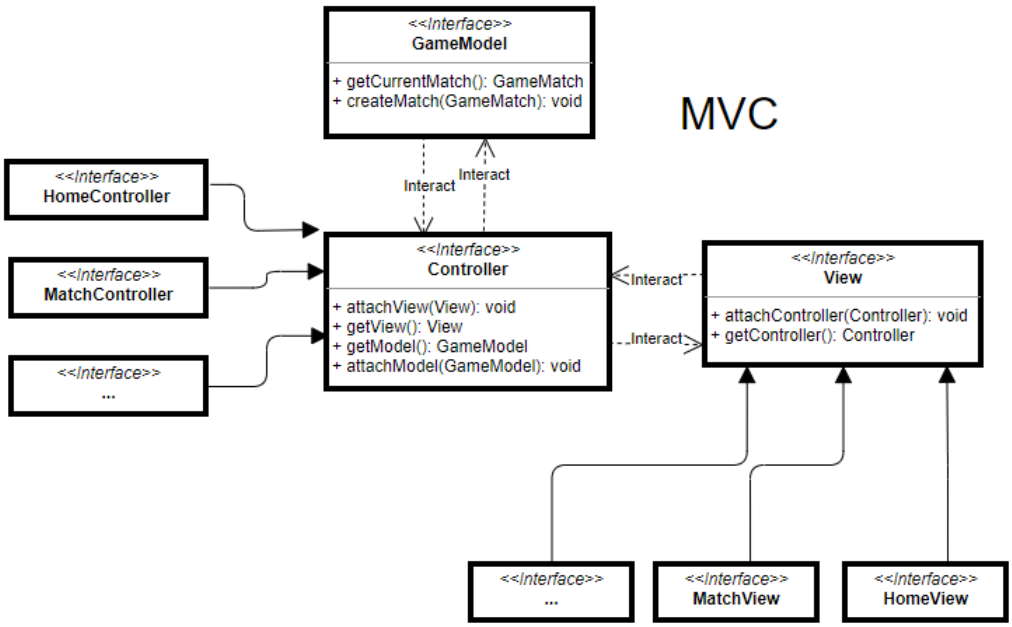

Figura 2.1: Schema UML dell'architettura M-V-C

2.2 Design dettagliato

2.2.1 Luigi Borriello

Oggetti di gioco

Come prima cosa, mi sono occupato della creazione degli oggetti di gioco, realizzati con l'aiuto del mio collega Gustavo Mazzanti.
Ho creato dunque un entità **GameObject** che rappresenta un oggetto "base" contenente tutte le caratteristiche che effettivamente tutti gli oggetti condividono.

Ogni tipo di oggetto di gioco realizzato in seguito, è dunque una specializzazione dell'oggetto base `GameObject`, e, avendo realizzato quest'ultimo utilizzando il pattern **Template Method**, il suo metodo astratto `update()` verrà implemetato in ogni oggetto a seconda del comportamento che deve assumere ad ogni aggiornamento segnalato dal `GameLoop`.
Facendo così sarà inoltre possibile implementare nuovi oggetti anche in futuro, semplicemente estendendo `GameObject`.

Tutti gli oggetti vengono collezionati nel `World` sotto forma di `GameObject`, nei quali poi andrà semplicemente a richiamare su tutti il metodo `update()` (approfondito successivamente), senza preoccuparsi di come effettivamente questi lo abbiano implementato.
Per ogni oggetto è stata realizzata un'apposita interfaccia che ne definisce il comportamento.

Ogni `GameObject` possiede un *collider*, nonchè un quadrato-2D(realizzato con *Rectangle2D*, importato dalla libreria *JavaFX.Geometry*) che ne definisce le dimensioni all'interno della mappa, mentre la posizione associata ad ogni oggetto corrisponde al vertice in alto a sinistra del collider. Quest'ultimo, viene generato automaticamente partire dalla posizione dell'oggetto e viene utilizzato per controllare e gestire le varie collisioni fra i GameObject.

MoveableObject rappresenta più nello specifico tutti quegli oggetti di gioco che possono muoversi e spostarsi quindi all'interno della mappa(`Bomber` e `Nemici`). E' caratterizzata infatti da metodi volti a modificarne la posizione, di fatto tutte le entità in grado di muoversi nella mappa estenderanno `MoveableObject`.

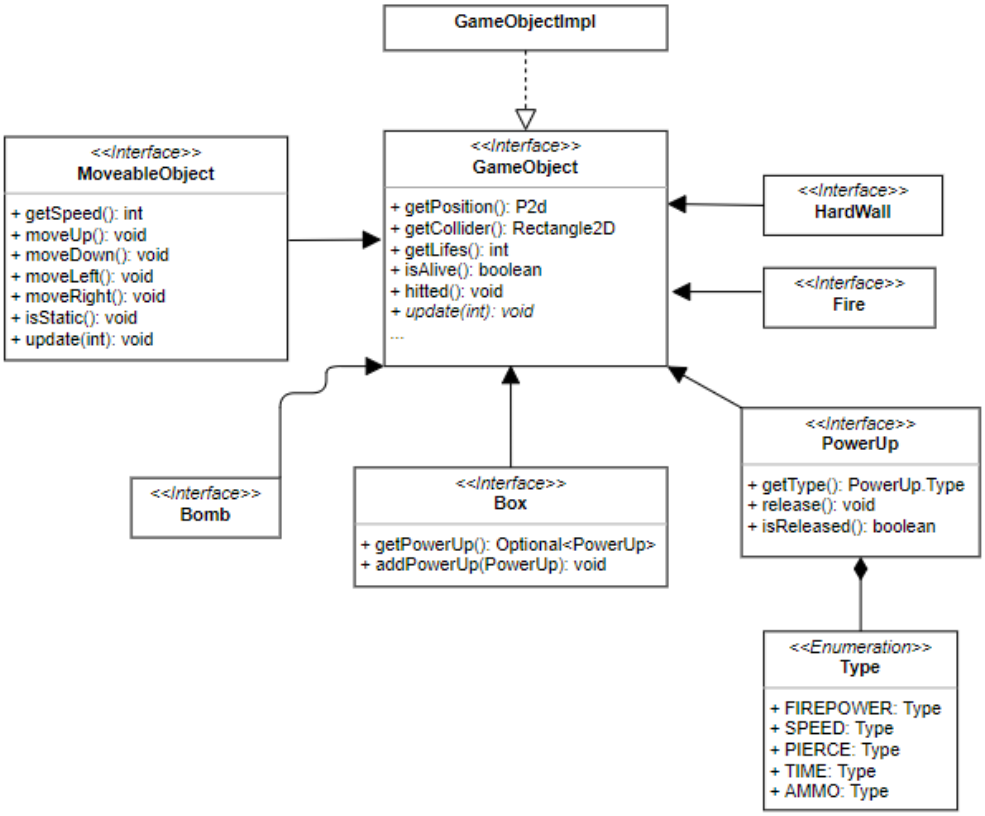

Figura 2.2: Schema UML della `gerarchia` degli oggetti di gioco.

GameObjectFactory

Per la creazione degli oggetti, è stato scelto il pattern **Factory**, infatti ogni istanza del `World` sarà dotata di una **GameObjectsFactory** la quale verrà usata per generare i vari oggetti di gioco.

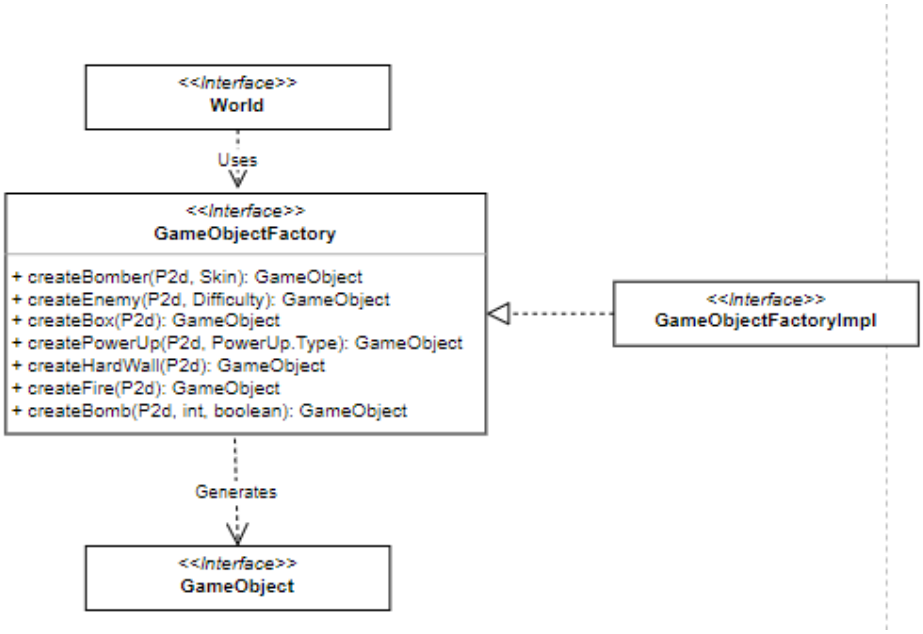

Figura 2.3: Schema UML della GameObjectFactory.

WorldEvent

Gli eventi di gioco, ovvero le varie collisioni fra oggetti di gioco e l'esplosioni delle bombe, sono tutti un'implementazione specifica dell'interfaccia **WorldEvent**, si dividono in:

- **ExplosionEvent**: viene modellata l'esplosione e generato il fuoco nella mappa a seconda delle caratteristiche del bomber(potenza di fuoco, ecc...) e dalla presenza o meno di muri.
- **PickPowerUpEvent**: viene applicato sul `Bomber` il `PowerUp` raccolto.
- **HitBorderEvent**: vengono gestite le collisioni di un `MoveableObject` con un muro/cassa.
- **HitEntityEvent**: vengono gestite le collisioni di un oggetto con il `Fire` delle esplosioni e l'eventuali collisioni fra gli `Enemy` e il `Bomber`.

Gli eventi vengono notificati con l'utilizzo del pattern **Observer**, infatti ad ogni frame, a seguto di controlli specifici (approfonditi successivamente), il `World`(*Subject*) segnala eventuali `WorldEvents` al `WorldEventListener`(*Observer*, componente del `MatchController`) il quale li accoda nella propria lista, e, di volta in voltà andrà a chiamare su tutti il metodo *process()*.

Ogni `WorldEvent` ha accesso al `GameMatch`, ovvero all'istanza della partita, passatagli dal Controller in modo tale da poter modificare lo stato di quest'ultima.

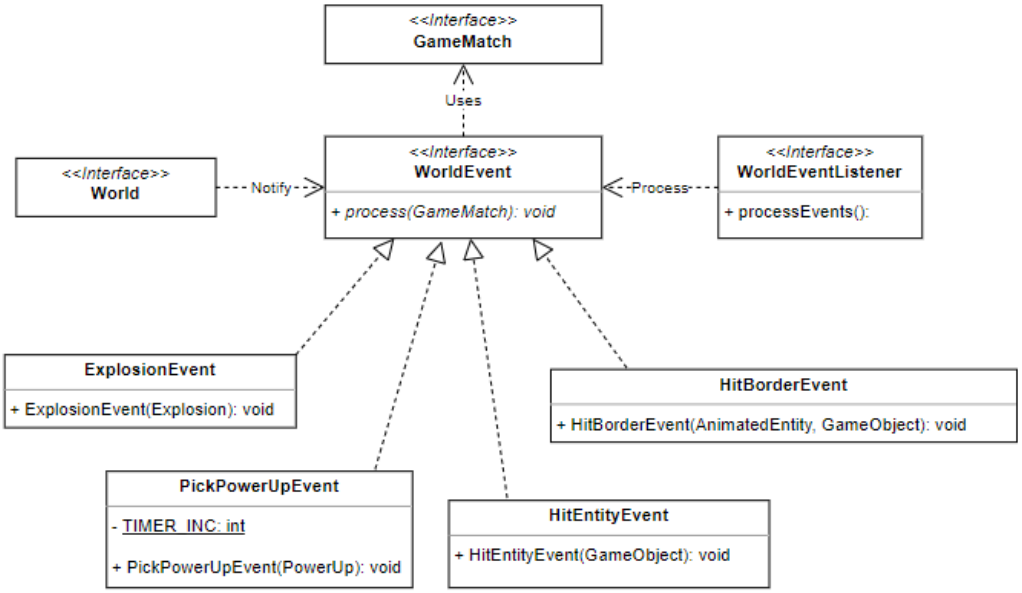

Figura 2.4: Schema UML della gestione degli eventi.

Timer

Per la realizzazione del **Timer** della partita è stata sviluppata una semplice classe `Time` che permette una comoda e personalizzata gestione del tempo; nello specifico permette di ottenere data una quantità totale di secondi, il tempo corrispondente nel formato *minuti:secondi*. Ovviamente vi è un'entità `Timer` con metodi per decrementare e aumentare una determinata istanza di `Time`.
Il `TimerThread` eseguirà poi secondo dopo secondo, una *decTime()* sul `Timer` di cui si compone.

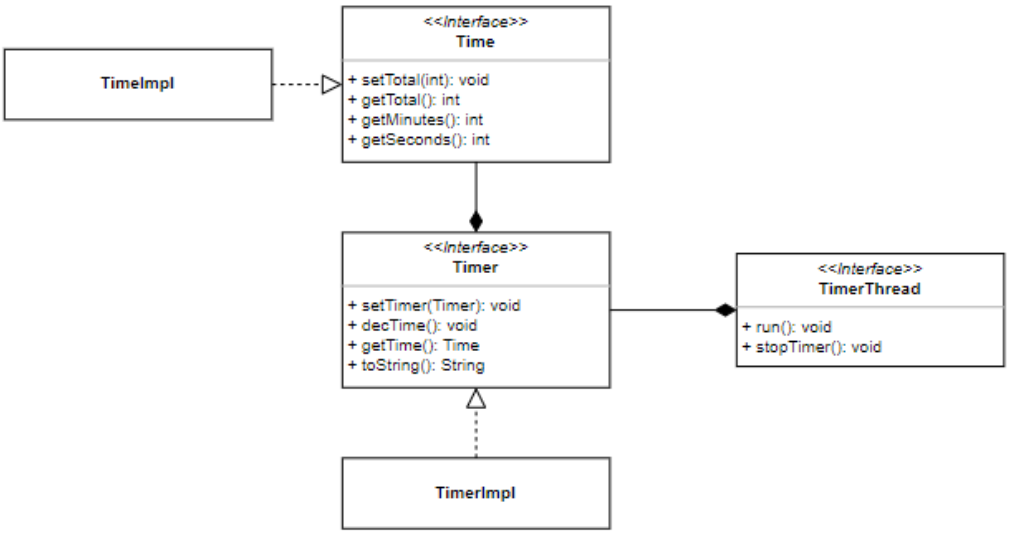

Figura 2.5: Schema UML della realizzazione del Timer.

MatchController

Mi sono occupato inoltre, della realizzazione del **MatchController**, ovvero il Controller associato alla partita, contenente il **GameLoop** dove ad ogni frame (60Fps):

- Elabora gli input notificati dalla `MatchView`.
- Processa gli eventi notificati dal `Model`.
- Richiama l'*update()* sul `GameMatch`.
- Richiama la *render()* sulla `MatchView`.

In fase di analisi, ho deciso che, volendo seguire il principio di *SRP*, ovvero *Single Responsibility Principle*, avrei sviluppato componenti separate, per la gestione degli input ricevuti dalla `View` e per la gestione degli eventi notificati dal `Model`.
Infatti, il `MatchController` si compone di:

- un **WorldEventListener**, il quale corrisponde all'*Observer* lato-Model, grazie al quale il World riesce a notificare l'avvenimento dei vari `WorldEvent`. Come anticipato prima l'ascoltatore ogni volta in cui gli viene notificato un `WorldEvent`, lo accoda in una apposita lista, per poi andare a processare ogni elemento di quest'ultima ad ogni frame.

- un **CommandListener**(approfondito successivamente), definibile come l'*Observer* lato-View, il quale si interfaccerà con la `MatchView`, e ad ogni tasto premuto verrà settato nel suo componente `PlayerBehavior`, il relativo tasto premuto e a seconda di quest'ultimo verrà processato il relativo comando da eseguire.

Il `MatchController` fornisce inoltre alla `MatchView` metodi necessari ad ottenere tutte le informazioni utili a quest'ultima, andandole a prendere all'interno del `Model`.
Infine, il `MatchController` è in grado di riconoscere la fine della partita e, una volta arrivata, terminare autonomamente la sua esecuzione.

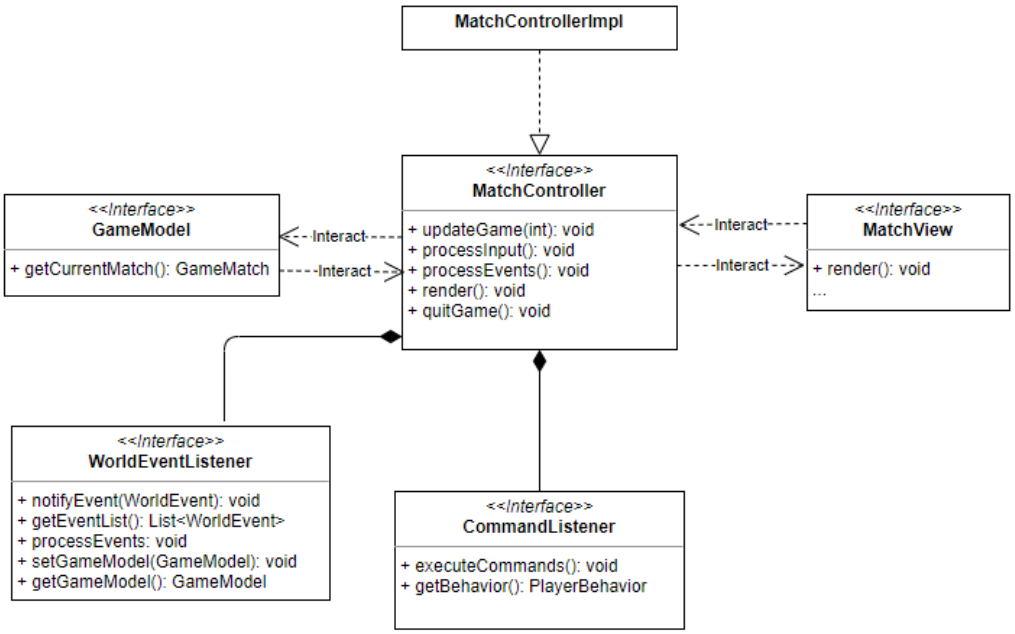

Figura 2.6: Schema UML della realizzazione del MatchController e delle entità da lui coinvolte.

MatchView

La **MatchView** si occupa del rendering della partita, e per farlo in una maniera ottimizzata, utilizza principalmente due **Canvas**, il primo con cui disegna il background, ovvero gli oggetti che non cambiano mai durante la partita, mentre con il secondo, va a disegnare ad ogni frame, gli oggetti in base al loro stato aggiornato.

I metodi principali che caratterizzano la `MatchView` sono:

- *drawGame()*: prepara la partita, disegnando la Scorebar e il Background.
- *render()*: Renderizza la schermata a seconda dello stato della partita ad ogni frame. Quindi aggiorna la scorebar gli oggetti del gioco, i quali vengono disegnati secondo un ordine specifico in modo tale da non essere sovrapposti causando "bug" grafici.

Per renderizzare i vari oggetti nel canvas utilizza il suo **GraphicsContext** con il quale disegna le varie `Image` collezionate nelle enumerazioni `GameImages` e `AnimatedObjectsSprite`.

Utilizza i metodi forniti dal `MatchController` per accedere alle informazioni utili alla realizzazione della schermata e, si occupa anche di notificare al Controller, gli input da tastiera ricevuti dall'Utente tramite la sua **ControlsMap**, la quale ad ogni `KEY_CODE` ha associato un `Consumer<Boolean>` che andrà a cambiare il determinato valore del `PlayerBehavior`.
A seconda dei controlli scelti dall'utente, la `ControlsMap` viene settata con dei `KEY_CODE` specifici.

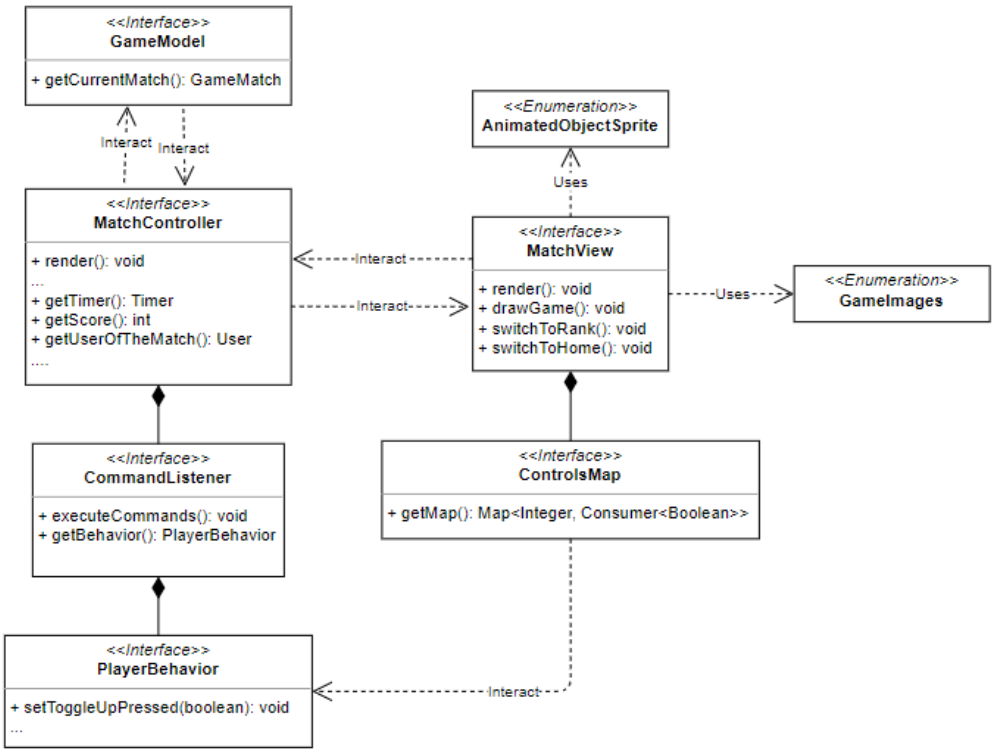

Figura 2.7: Schema UML della realizzazione della MatchView e delle entità da lei coinvolte.

Vari Tools ad accesso unificato

Per la realizzazione dei vari tools del Software, anzichè creare intere classi statiche, sono state sviluppate delle *Utility Class* tramite l'utilizzo del pattern **Singleton**, questo per garantire che abbiano comunque una unica istanza, accessibile globalmente e senza doversi preoccupare di fornirne il riferimento a chi lo richiede.
Sono state realizzati insieme agli studenti Gustavo Mazzanti e Tommaso Brini.
Nello specifico:

- **DirectoryLoader**: controlla ad ogni avvio del programma, se nella *Home Directory* dell'utente vi è presente la cartella *'.bomberOne/'* con relativi file di testo, utilizzati per memorizzare i punteggi delle partite, e se così non fosse, li creerebbe.

- **RankLoader**: legge e scrive sui file presenti nella cartella *'.bomberOne/'*, salvando partita dopo partita i punteggi ottenuti. La serializzazione avviene tramite liste di User(approfondito successivamente), tenute separate, così come i file di testo, una per ogni Difficoltà disponibile del gioco.

- **ResourceLoader**: carica dalla cartella *"resources/"* varie risorse, nonchè il Font e le Immagini(file *".png"*) usate dalle View, e le mappe di gioco(disposizione dei muri, per ora una sola) tramite file *".csv"*.
  Quest'ultimo, comunica con le seguenti enum, usate per collezionare le varie risorse caricate:

  - `GameImages`: enumerazione i cui valori contengono semplici immagini, dagli oggetti inanimati al logo dell'applicazione.
  - `AnimatedObjectsSprite`: enumerazione i cui valori contengono matrici di immagini, utili alla rappresentazione grafica degli oggetti animati del gioco.
  - `Maps`: enumerazione i cui valori contengono liste di strighe ottenute dalla lettura di file ".csv"

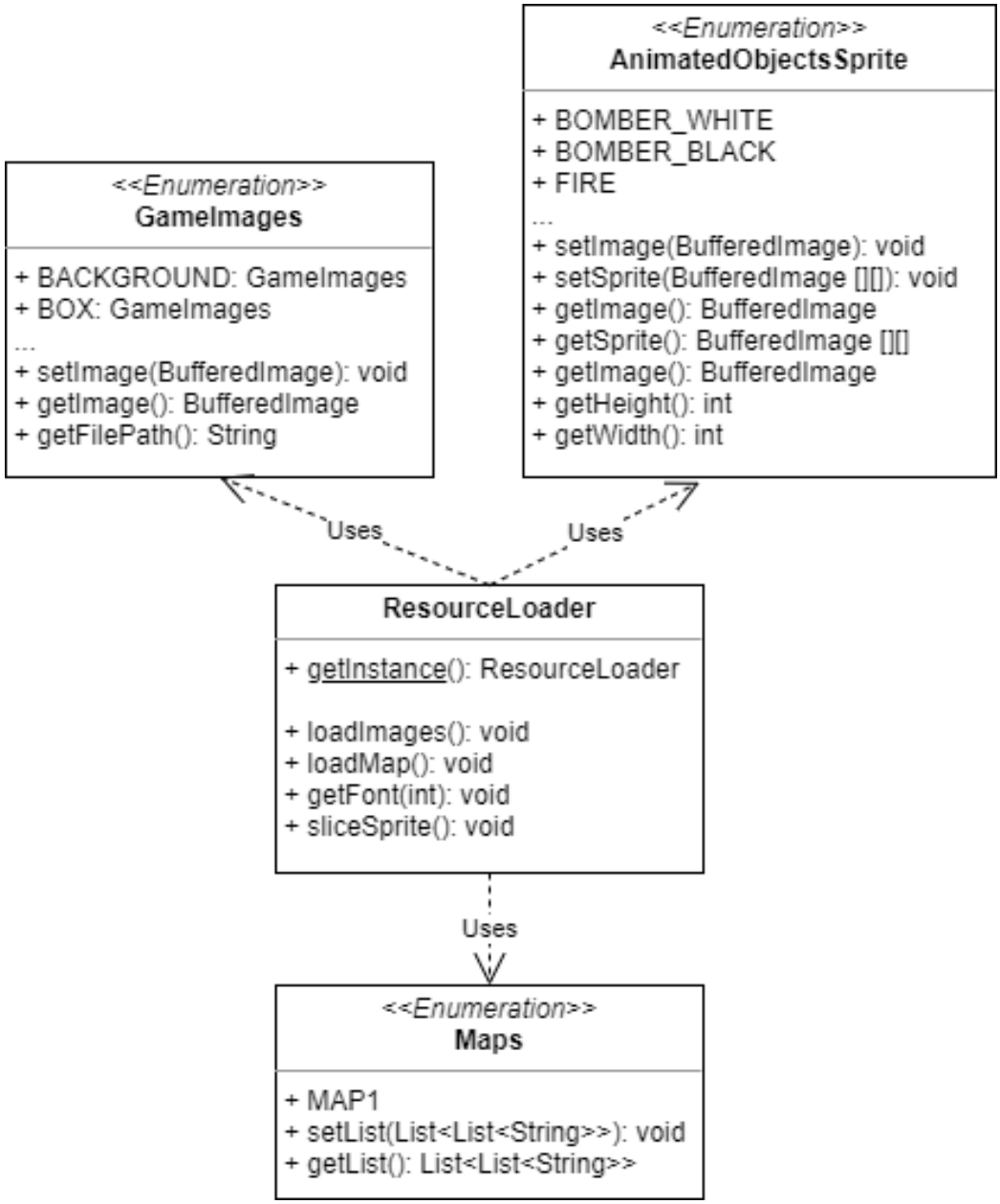

Figura 2.8: Schema UML della realizzazione del tool ResourceLoader e dall'entità che lo coinvolgono.

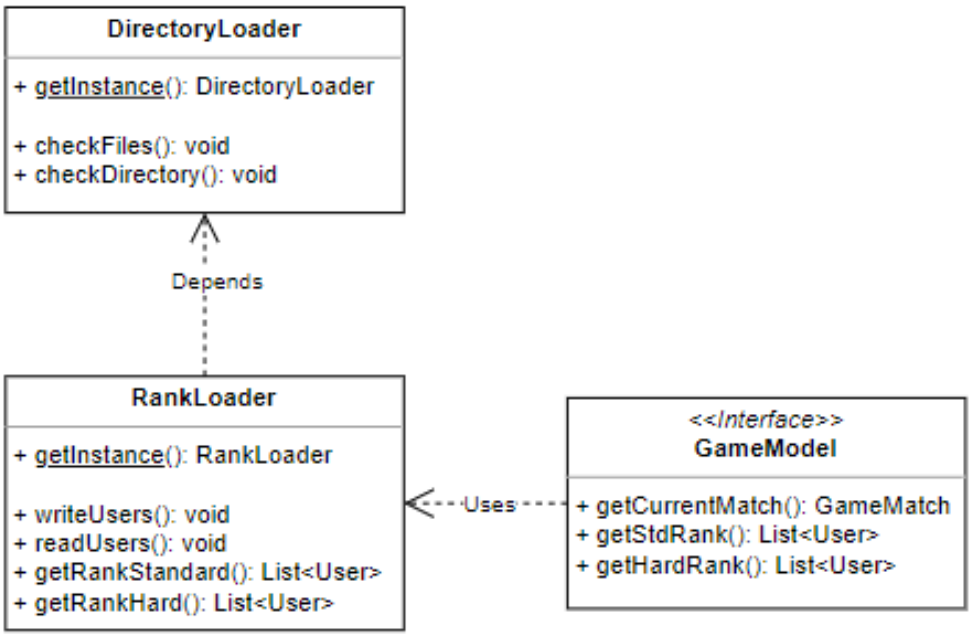

Figura 2.9: Schema UML della realizzazione dei Tools DirectoryLoader e RankLoader, da sottolineare che l'esecuzione del primo, permette al secondo di poter leggere/scrivere sui vari Files.

2.2.2 Gustavo Mazzanti

Oggetti di gioco

In fase di progettazione abbiamo pensato di creare una classe generale GameObject che potesse essere estesa per creare i vari singoli oggetti del gioco. Questa, sfruttando il pattern **Template Method**, possiede un metodo abstract **update()** che ci ha "obbligato" a implementarlo in tutte le sue sottoclassi. I due oggetti di gioco non-Moveable che ho implementato sono **Bomb** e **Fire**, che riguardano le bombe che il **Bomber** può sganciare nella mappa e il fuoco generato dall'**Explosion** di queste.

- **Bomb**: `GameObject` esteso, in grado di generare una `Explosion` con le dovute impostazioni, in base ai `PowerUp` posseduti dal `Bomber`. Il metodo `update()` è implementato in modo tale che gestisca il timing per l'`Explosion` e l'animazione di pulsazione precedente a questa.

- **Fire**: semplice `GameObject`, generato dalla `Explosion`, nel quale l'estensione si limita ad implementare il metodo `update()`, in modo tale che gestisca il tempo di vita del fuoco e la relativa animazione.

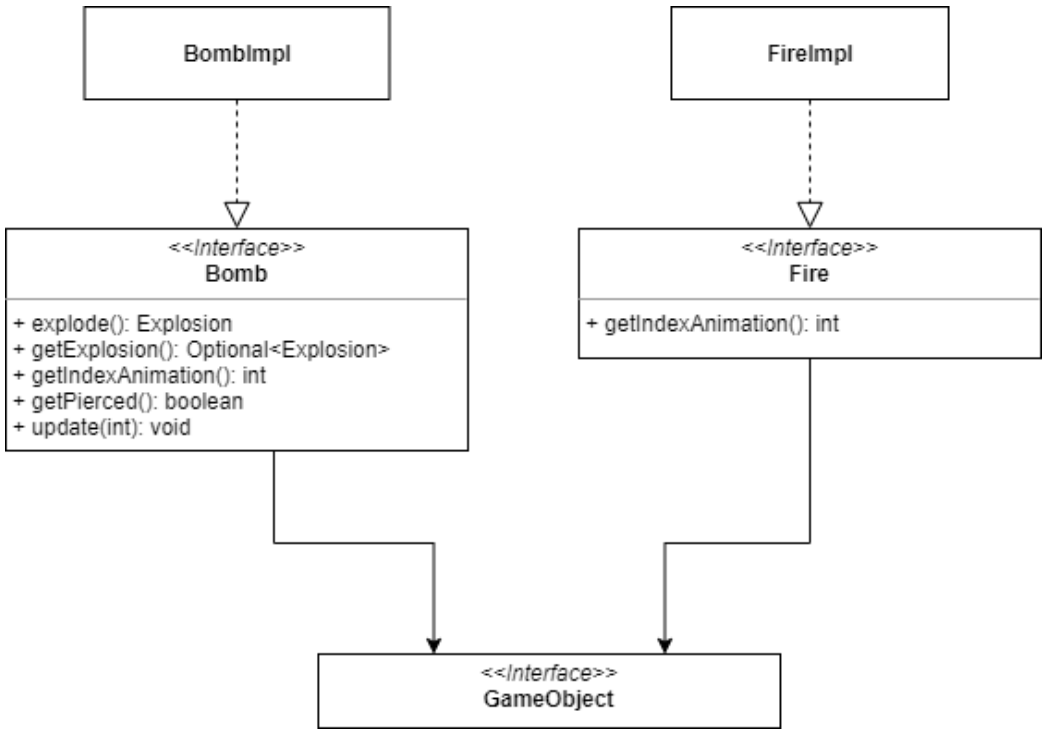

Figura 2.10: Schema UML dei GameObject Bomb e Fire.

MoveableObjects

In seguito mi sono occupato della realizzazione di `MoveableObject`, `GameObject` esteso, con dei metodi aggiuntivi per la gestione dei movimenti nella mappa:

- **moveDir()**: 4 metodi, uno per ogni movimento bidimensionale, implementati tenendo in considerazione il tempo trascorso, in modo tale da avere una animazione di movimento più fluida.
- **metodi vari**: implementati alcuni metodi di impostazione e di controllo per la gestione delle animazioni di tipo `Sprite`, ovvero una matrice di immagine per le animazioni di movimento dei personaggi del gioco.
- **update()**: piccola estensione del metodo `update()` di `GameObject`, si occupa di reimpostare il `collider` del personaggio in questione ed

aggiorna il tempo passato dall'ultima volta in cui è stata richiamata la `update()`.

Questa classe si è resa utile per implementare le due estensioni **Bomber** e **Enemy**, unici due `MoveableObject` del gioco.

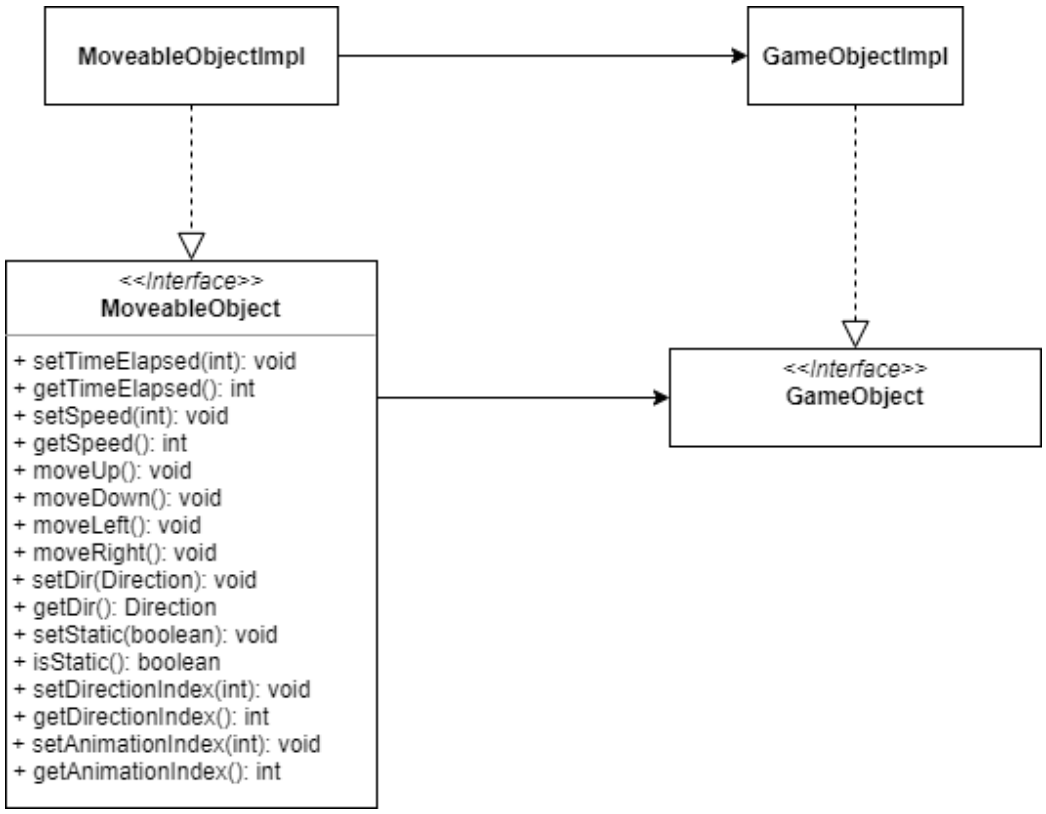

Figura 2.11: Schema UML di MoveableObject.

In particolar modo mi sono occupato anche della realizzazione del **Bomber**, il protagonista del videogioco.

Bomber

L'oggetto protagonista del gioco, in grado di sganciare `Bomb`, prendere e gestire i `PowerUp`, distruggere le `Box` e gli `Enemy` tramite l'`Explosion` delle `Bomb` piantate.

I suoi metodi principali sono:

- **costanti e caratteristiche**: sono presenti diverse costanti utilizzate per inizializzare il `Bomber`, per incrementare i settaggi al raccoglimento dei `PowerUp` e altre costanti di controllo. Mentre i settaggi contengono le caratteristiche del `Bomber` modificate grazie ai `PowerUp` raccolti.
- **respawn()**: metodo che riporta il bomber alla posizione iniziale ripristinando tutte le sue caratteristiche (annullamento `powerUp`), resettandoli ai valori iniziali.
- **metodi vari**: metodi utilizzati per impostare e verificare i settaggi come: le vite, le munizioni, la velocità, la potenza di fuoco e altre caratteristiche.
- **plantBomb()**: metodo che, se disponibile, genera una `Bomb` nella posizione corrente del `Bomber`.
- **Gestione PowerUp**: i `PowerUp` vengono applicati al `Bomber` tramite un metodo che prende in input il tipo di `PowerUp` e si occupa di richiamare il metodo per impostare corretamente i settaggi del `Bomber`.
- **moveDir()**: essendo un `MoveableObject` vengono estesi anche i metodi `move` descritti in precedenza, con qualche controllo in più.
- **update()**: questo metodo è stato esteso per gestire anche gli indici relativi all'immagine, per via della sua sprite image (immagine divisa in matrice), tante piccole immagini per generare le animazioni del protagonista, ad ogni movimento, l'update(), si occuperà di gestire questi indici.

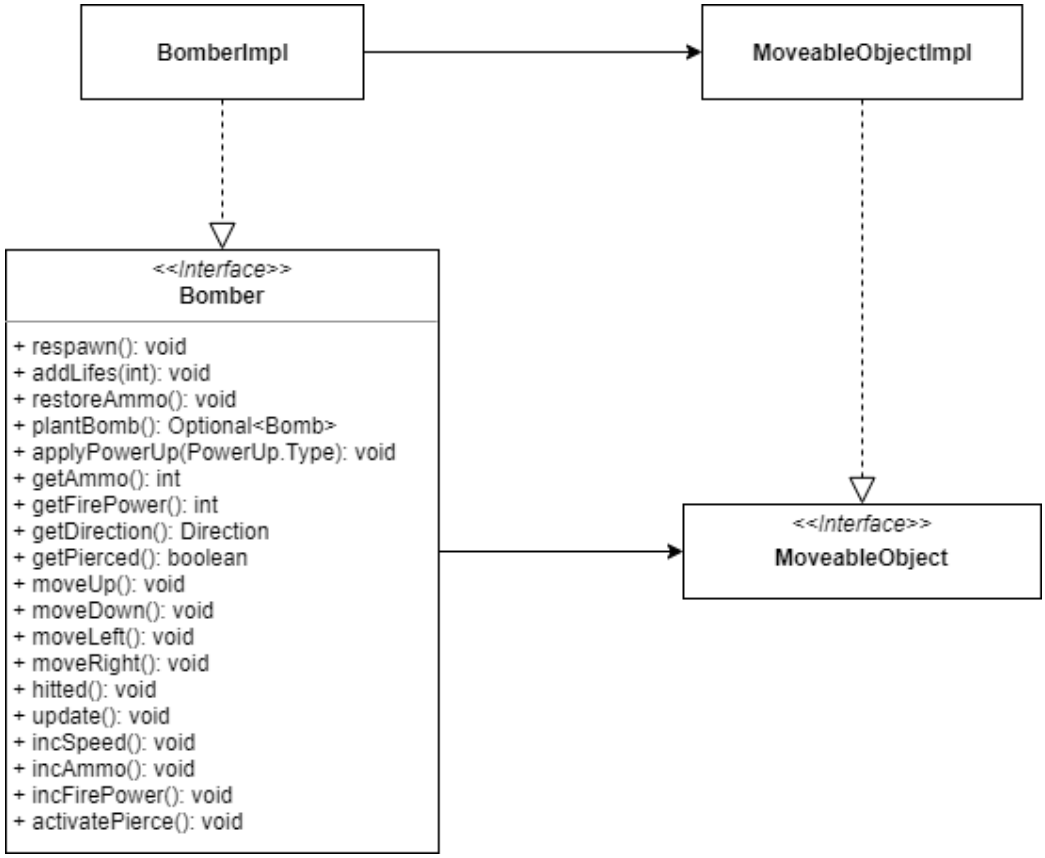

Figura 2.12: Schema UML di Bomber.

Package Input

Per tutti i movimenti del **Bomber** è stato creato un nuovo package contenente una classe **PlayerBehavior**, questa tiene traccia dei comandi premuti per controllare il **Bomber** con l'utilizzo di 5 variabili boolean, uno per ogni comando del player.
Al suo interno troviamo 2 metodi per ogni variabile, una per verificarne lo stato e l'altro per impostarlo. Oltre a questa è stata implementata una classe, con la relativa intefaccia, in grado di poter rimanere in ascolto su **PlayerBehavior**.
Questa classe, **CommandListener**, possiede principalmente un solo metodo per eseguire i comandi sul **Bomber**.

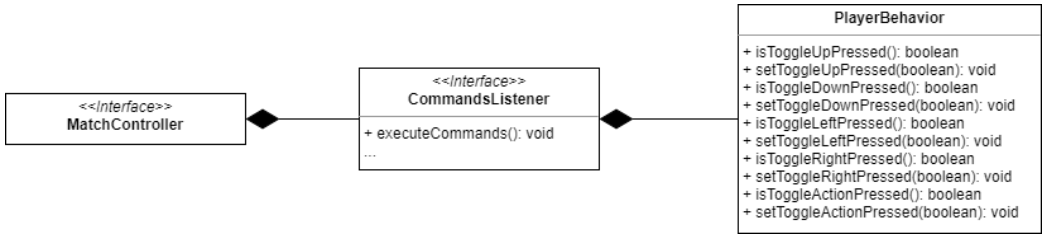

Figura 2.13: Schema UML del package Input.

Package Commands

Il package Commands si trova all'interno del package Input e contiene tutti i comandi che il **Bomber** può eseguire.
Al suo interno troviamo una interfaccia **Command** che possiede due soli metodi, `execute(GameMatch)` e `dir()`, rispettivamente per eseguire il comando nel **GameMatch** in input e per ricevere la **Direction** corrente del **Bomber**.
Questa interfaccia è stata implementata in 5 classi differenti, una per ogni comando possibile al **Player**: **MoveUp**, **MoveDown**, **MoveLeft**, **MoveRight** e **PlantBomb**, ognuno di questi contiene i metodi descritti per l'interfaccia **Command**, il metodo `execute()` non farà altro che richiamare il metodo giusto per il movimento sul **Bomber** passando tramite il **GameMatch**.
Tutte queste classi di **Commands** verranno poi richiamate dal **CommandsListener** descritto in precedenza.

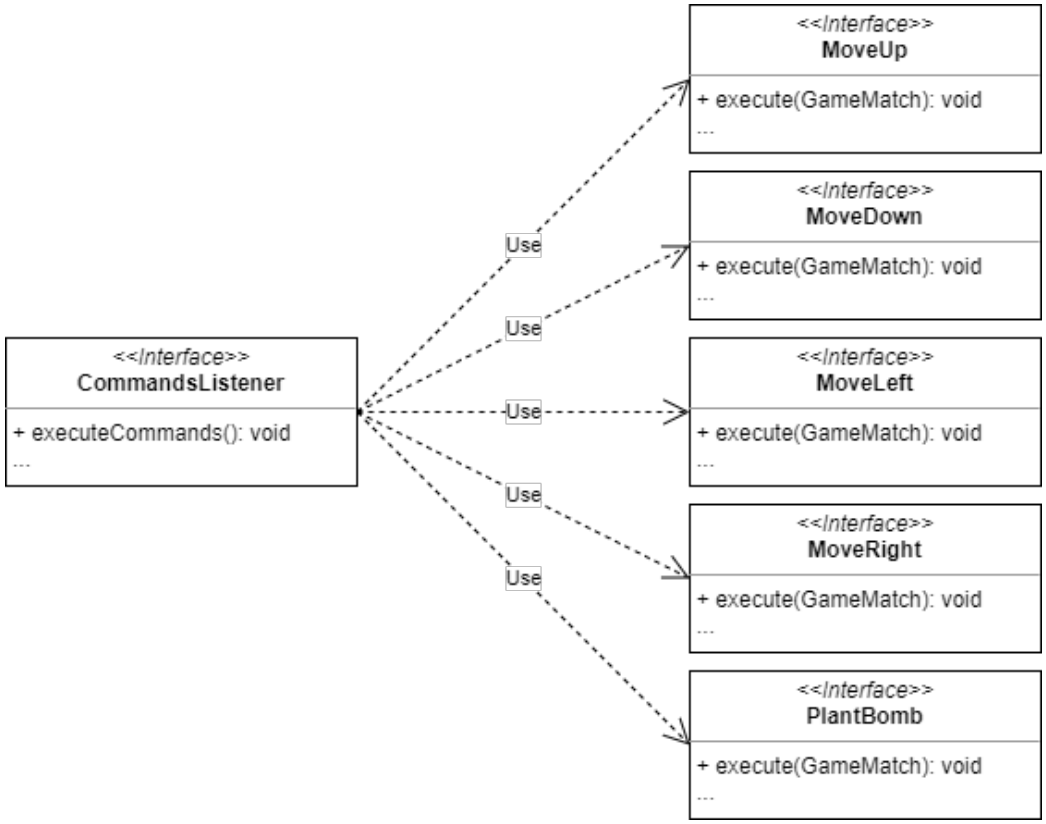

Figura 2.14: Schema UML dei Commands.

Explosion

Questa classe è stata pensata come generatore di esplosione, in modo tale che la `Bomb` potesse generare una `Explosion`, in grado, tramite ExplosionEvent (spiegato in precedenza), di generare tanti `Fire` in base ai `PowerUp` di tipo FirePower presi dal `Bomber`.
La scelta di non estenderlo come `GameObject` è stata presa in quanto questo oggetto non aveva bisogno dei metodi comuni di GameObject come ad esempio il metodo per avere un collider, ma si limita solamente a generare i `Fire` necessari, tramite `ExplosionEvent`.
Gli unici metodi che possiede questa classe sono dei controlli sul tipo di `Bomb` che ha generato l'`Explosion` (`getPierced()` e `getFirePower()`), e un metodo per ricevere la posizione centrale dell'`Explosion`, nonchè la posizione in cui si trovava la `Bomb` che ha generato l'`Explosion`.

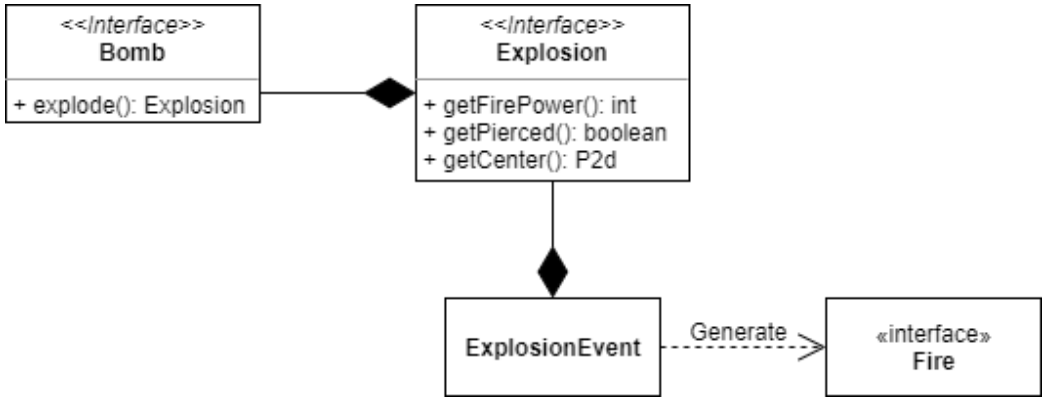

Figura 2.15: Schema UML di Explosion.

HomeView

La **HomeView** è la schermata iniziale del gioco, strutturata per poter accedere direttamente alla **RankView**, alla **RulesView** oppure per avviare una partita, quindi alla**SetUpView**. All'interno di questa schermata sono stati inseriti 3 bottoni, relativamente alle 3 possibili operazioni, oltre a ciò è anche presente un piccolo EasterEgg, una sorta di riconoscimento per gli sviluppatori. I suoi principali metodi sono `init()` che si limita a richiamare il metodo `drawHome()`, quest'ultimo invece si occupa di caricare tutte le immagini della view corrente. Ci sono altri metodi come: `switchToRank()`, `switchToRules()` ed altri che servono semplicemente per cambiare la view visualizzata.

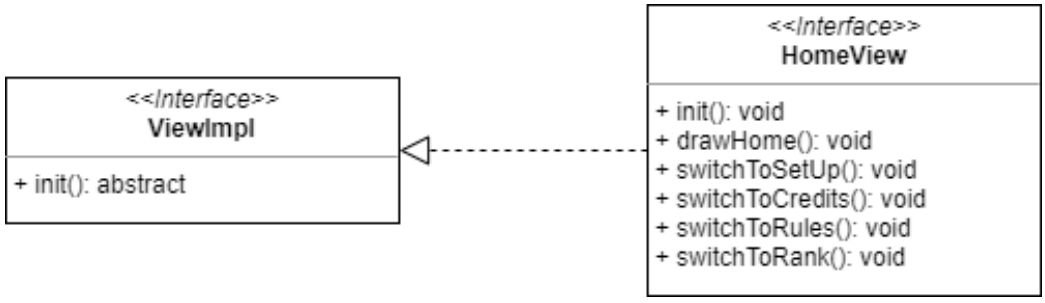

Figura 2.16: Schema UML della HomeView.

RulesView

La **RulesView** è una semplice schermata contenete le regole del gioco, come per la **HomeView** sono presenti i metodi `init()` e `drawRules()` i quali svolgono gli stessi compiti che hanno nella **HomeView**. Anche qui è presente un metodo `switchToHome()` in grado di poter riportare la view alla `Home` del gioco.

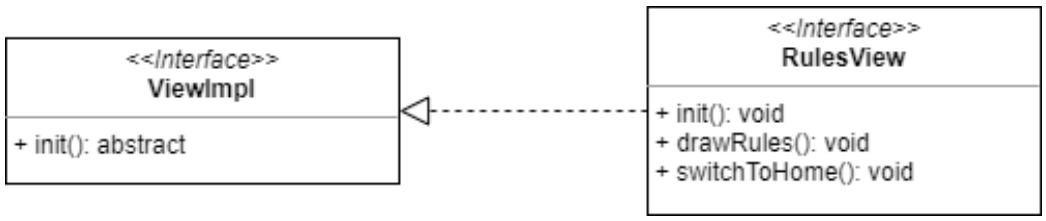

Figura 2.17: Schema UML della RulesView.

CreditsView

La **CreditsView** è una schermata nascosta all'interno della **HomeView**, si tratta di un semplice EasterEgg che ha gli stessi metodi e comportamenti della **RulesView**.

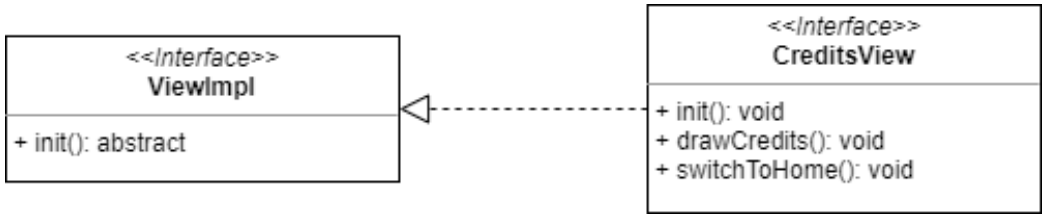

Figura 2.18: Schema UML della CreditsView.

Controller delle view

Per tutte e tre le View che ho implementato, non avendo richieste particolari per il controller, mi sono limitato ad implementare una sola classe chiamata **HomeController** la quale estende la classe **Controller** senza aggiungere nulla se non il metodo `init()` lasciato vuoto.

Figura 2.19: Schema UML della HomeController.

Tool Audio

Mi sono occupato della gestione dell'audio del gioco, sviluppando 3 classi:

- **Sounds**: enumeratore contenente i tipi di suono: `AUDIO` e `EFFECT`;
- **GameSounds**: enumeratore contenente tutti i suoni del gioco, possiede un costruttore per inizializzare i suoni con il tipo di suono, il volume e la path relativa al file, inoltre sono stati implementati alcuni metodi come getVolume(), getMediaPath() e getType(), che ritornano rispettivamente il volume del GameSounds, la path del file e il tipo di GameSounds (`AUDIO` o `EFFECT`).
- **SoundsHandler**: classe implementata sfruttando il pattern **Singleton**, grazie al quale è possibile creare una sola istanza per eseguire gli audio del gioco. Questa classe al suo interno possiede 2 `MediaPlayer`, dei semplici lettori per poter eseguire i file audio, `playerAudio` relativo agli `AUDIO` e `playerEffects` relativo agli `EFFECT`, il suo costruttore si occupa di caricare tutti i GameSounds nelle due chache (`chacheAudio` e `chacheEffects`). I suoi metodi sono: `getIstance()` che ritorna l'istanza per gli audio (Pattern Singleton), `start(GameSounds)` si occupa di far partire l'`AUDIO` o l'`EFFECT` in input nel giusto player mettendo il loop infinito gli `AUDIO` oppure eseguendo una sola volta gli `EFFECT`, infine ci sono i metodi `replayAudio()` e `stopAudio()` che fanno rispettivamente ripartire e stoppare l'ultimo `AUDIO` eseguito sul `playerAudio`.

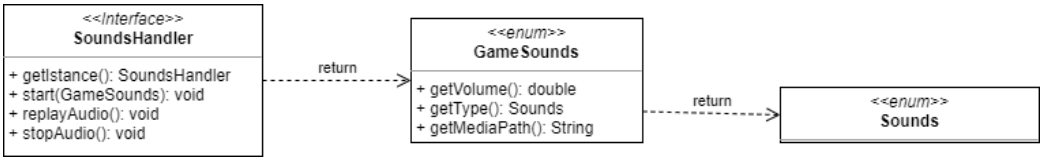

Figura 2.20: Schema UML dei tool audio.

Font

Per l'importazione di un font arcade ho sfruttato la classe **ResourceLoader** implementata da Luigi Borriello (spiegata in precedenza) e mi sono limitato a creare un nuovo metodo chiamato `getFont(int size)` che si occupa di restituire il font caricandolo da res/ nella dimensione richiesta in input.

ResourceLoader UML

2.2.3 Tommaso Brini

GameObjectCollection

In fase di progettazione abbiamo pensato a come implementare una collezione che contenga tutti gli oggetti di gioco presenti e che rendesse efficace e immediata la loro creazione e eliminazione. Per questo motivo, abbiamo deciso di creare l'interfaccia GameObjectCollection che grazie all'uso di liste e alcuni metodi specializzati ha il compito di gestire e contenere tutti gli oggetti di gioco. La sua relativa implementazione, `GameObjectCollectionImpl`, consiste infatti in una lista di GameObject e dei seguenti metodi:

- **metodi vari** per la restituzione di liste contententi solo gli oggetti appartenti a una specificata categoria, per esempio Bombe, Nemici e tutti gli altri GameObject. Questi metodi consentono di filtrare la lista principale contenente tutti i GameObject e restituiscono una nuova lista contenente gli oggetti della categoria richiesta.
- **spawn(GameObject)**: aggiunge alla lista principale il GameObject passato come argomento del metodo stesso.
- **despawn(GameObject)**: elimina dalla lista principale il GameObject passato come argomento del metodo stesso.
- **getDespawnedObject()**: ritorna una lista contenente tutti i GameObject "morti", cioè quelli su cui il metodo isAlive restituisce false, ottenuta filtrando la lista principale.

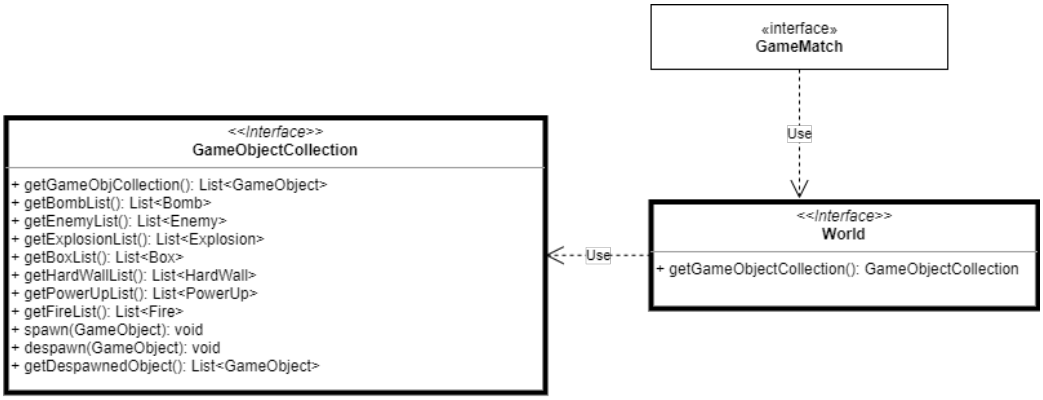

Figura 2.21: Schema UML della `GameObjectCollection`, contenente tutti gli oggetti di gioco.

WorldFactory

Per la creazione del mondo di gioco, è stato scelto il pattern Factory, infatti ogni istanza del GameMatch sarà dotata di una WorldFactory, usata per generare il mondo a seconda della difficoltà scelta dall'utente. La sua relativa implementazione, `WorldFactoryImpl`, contiene due metodi che resituiscono il World generato, il primo con la difficoltà settata a Easy, il secondo con modalità Hard.

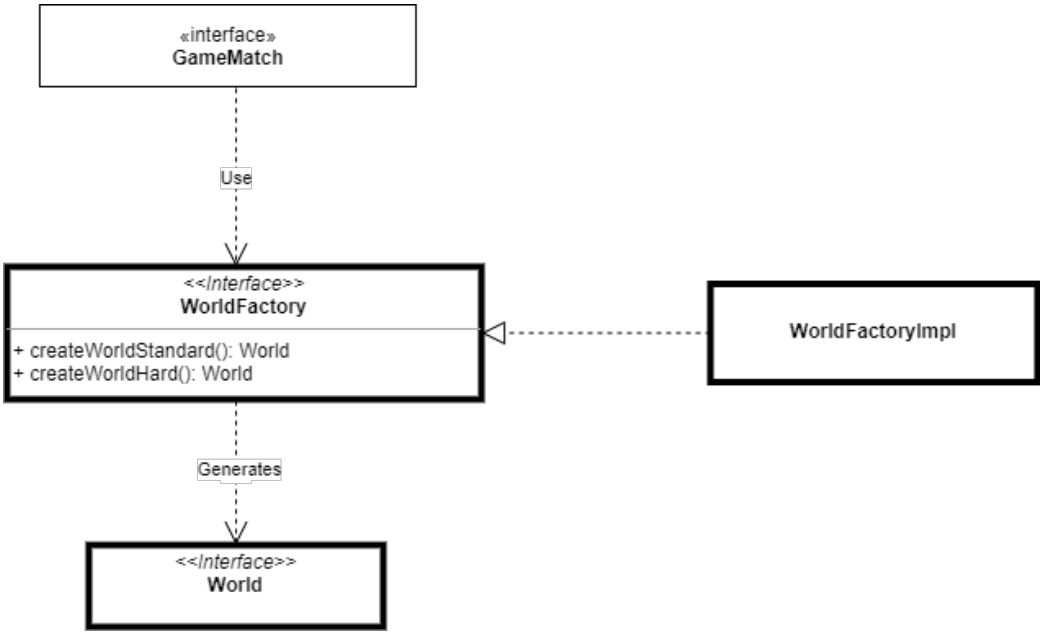

Figura 2.22: Schema UML di `WorldFactory`.

World

L'interfaccia World modella l'intero mondo di gioco. La sua implementazione, `WorldImpl`, contiene infatti:

- una istanza di **GameObjectCollection** che rappresenta l'insieme di tutti gli oggetti.
- una istanza di **ObjectFactory** per poter generare i vari oggetti.

Il costruttore della WorldImpl, richiamato attraverso il pattern Factory da WorldFactory, prende in input la difficoltà e la skin scelta dall'utente; il costruttore imposta la difficoltà, genera il Bomber con la skin passatagli come argomento, imposta la mappa e richiama i metodi privati per settare i nemici, le casse e i muri. I metodi principali sono:

- alcuni metodi getter e setter che restituiscono/settano i campi privati del World;
- **updateState()**: richiama l'update su ogni oggetto contenuto nella Collection, resetta la posizione dei nemici nel caso in cui il Bomber dovesse perdere una vita, elimina gli oggetti colpiti dalle esplosioni delle bombe richiamando il metodo despawn del GameObjectCollection.

- **checkCollision()**: per ogni oggetto "colpibile" (Bomber, Box, Enemy) contenuto della collezione, controlla se è stato colpito dal fuoco di una esplosione; se così fosse, segnala l'evento attraverso il listener. Per evitare bug di gioco, abbiamo deciso che i nemici e il bomber al momento dello spawn non sono colpibili per alcuni secondi. Controlla infine se il bomber viene colpito da uno dei nemici o se invece raccoglie un powerUp, segnalando l'evento attraverso il listener.

- **checkRespawn()**: se i nemici hanno la possibilità di respawnare al momento della loro morte, questo metodo ha il compito di spawnarli nuovamente al centro della mappa attraverso il metodo spawn della GameObjectCollection. Come già detto in precedenza, i nemici possono respawnare solo nella modalità Hard e solo se è ancora presente almeno una cassa.

- **checkBoundary()**: controlla che i nemici e il bomber non vadano a sbattere contro muri o casse, e eventualmente segnala l'evento attraverso il listener.

- **checkExplosion()**: per ogni bomba presente nella collezione, controlla se sta per esplodere e segnala l'evento attraverso il listener.

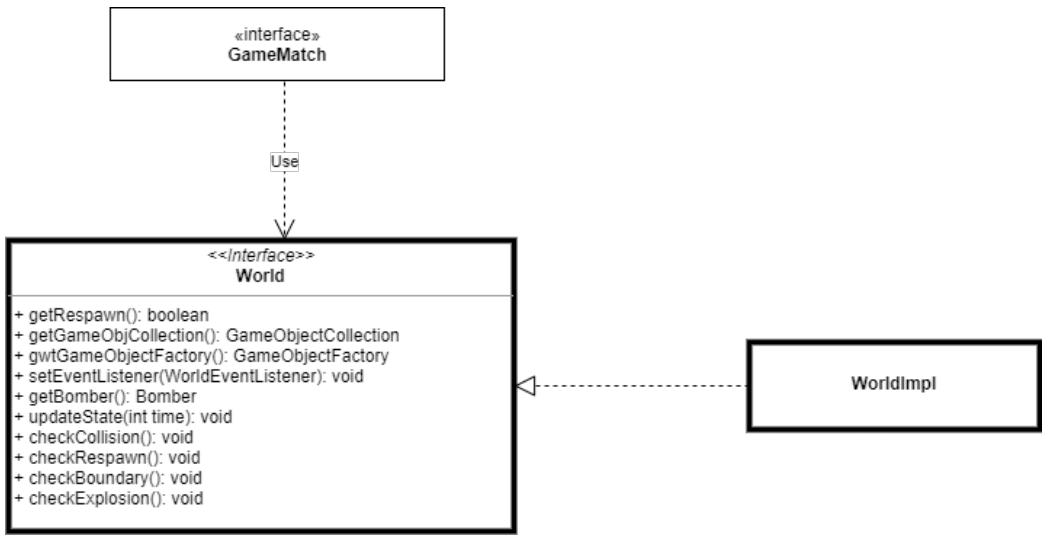

Figura 2.23: Schema UML di `World`.

GameMatch

La classe GameMatch rappresenta l'istanza della partita. All'interno della sua implementazione, **GameMatchImpl**, abbiamo deciso di tenere traccia dell'User, del World, del punteggio, del TimerThread e del gameOver. I metodi principali sono:

- **init()**: è il metodo da richiamare per iniziare effettivamente la partita. Crea il World in base alla difficoltà attraverso il pattern Factory del Worldfactory già descritto precedentemente e fa partire il timer.
- alcuni metodi getter e setter per il World, il punteggio, la difficoltà e l'User.
- **checkGameOver()**: imposta a true il booleano gameOver se è finito il tempo, se il bomber ha perso tutte le vite oppure se il giocatore ha vinto eliminando tutte le casse e i nemici.
- **updateGame()**: si occupa semplicemente di richiamare l'update del World e successivamente di richiamare il metodo checkGameOver.

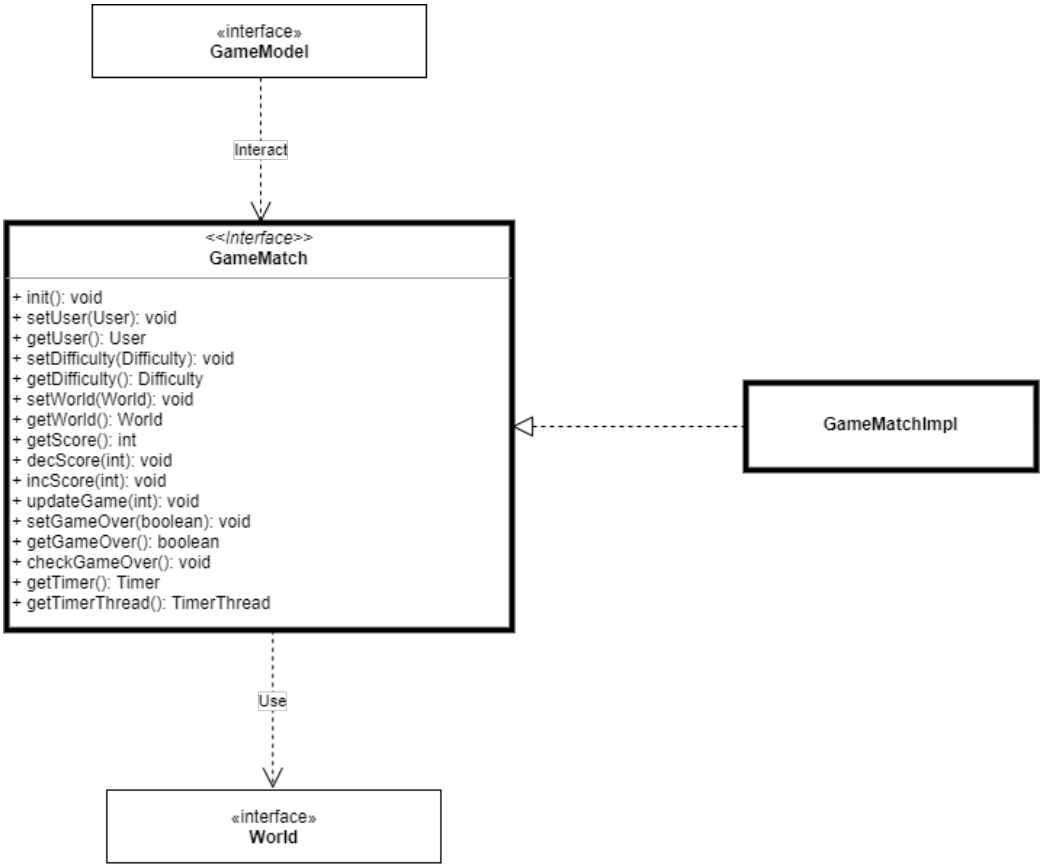

Figura 2.24: Schema UML di `GameMatch`.

Maps

Per caricare la mappa di gioco, ho deciso di creare un metodo in Resource-Loader che per ogni valore dell'Enum Maps legge il relativo file ".csv" e ne salva il contenuto grazie all'utilizzo di liste di stringhe. Queste liste verrano poi salvate dentro l'Enum stessa. Ho deciso di usare un Enum per facilitare l'inserimento futuro di possibili altre mappe, sarà necessario solamente aggiungere un altro file ".csv".

GameModel

Mi sono occupato della realizzazione del GameModel, ovvero una classe che tiene traccia della partita in corso e delle classifiche. Permette ai Controller di accedere alla struttura del Model. Contiene dei metodi per restituire

le classifiche (caricate attraverso il RankLoader, come già precedentemente descritto), un metodo per restituire la partita in corso e uno per iniziarla.

SetUpController

Mi sono occupato della realizzazione del SetUpController, ovvero il Controller associato alle impostazioni. Il SetUpController elabora i dati notificati dalla SetUpView, si occupa di regolare tutte le impostazioni della partita e dell'User scelte dall'utente. Ho pensato di utilizzare il pattern Builder per creare l'user, in modo da generarlo solo dopo aver scelto il nome, la skin e la modalità per i controlli. Dopo aver creato l'user, averlo settato per la partita corrente e aver settato la difficoltà scelta, può creare la partita.

SetUpView

La **SetUpView** consiste in un insieme di Button che andranno a settare le impostazioni per la partita da iniziare. I metodi principali sono:

- **drawSetUp()**: prepara la View delle impostazioni, disegnando i vari tasti e le label.
- **switchToGame()** e **switchToHome()**: tramite il ViewsSwitcher cambiano la Views, lanciando rispettivamente la MatchView e la HomeView. Il metodo switchToGame(), che lancia la MatchView e fa iniziare la partita, viene chiamato quando viene cliccato il Button "Play", ma solo se tutte le impostazioni sono state opportunamente settate.

2.2.4 Francesco Agostinelli

Enemy

In questo progetto, mi sono occupato di progettare ed implementare i nemici e la loro relativa IA (Intelligenza Artificiale). In particolare, in base alla difficoltà di gioco scelta dall'utente, i nemici hanno differenti "comportamenti", che sono stati implementati utilizzando uno strategy pattern. Nella difficoltà facile (easy mode) i nemici saranno pressochè innocui, poichè si muoveranno casualmente nella mappa di gioco, mentre nella difficoltà difficile (hard mode), i nemici si muoveranno casualmente finché non si viene individuati, infatti se il giocatore si avvicinasse troppo e venisse visto dai nemici, questi inizieranno a seguirlo.

Nella classe `EnemyImpl`, che estende `MoveableObject`, ho realizzato solamente metodi che modificano le sprite e che gestiscono le animazioni, poichè

semanticamente mi sembrava la scelta più idonea da fare, essendo EnemyImpl un'implementazione specifica, solamente espandibile e poco indicata ad essere riusabile, infatti in un game engine che rispetti ci si focalizza sulla riusabilità e la genericità di classi base come potrebbe essere, nel nostro caso, `GameObject` o MoveableObject per definire poi, i reali oggetti di gioco.

AbstractActions

I vari "behavior" sono stati implementati utilizzando uno strategy pattern, dato che è una soluzione molto efficente ed efficace che mi permette di assegnare a runtime, qualora lo ritenessi opportuno, un behavior differente dal precedente. Inoltre, mi permette di dividere l'implementazione in classi differenti, quindi rende il codice più indipendente, manutenibile e aperto ad espansioni.

Tutte i vari comportamenti sono estensioni della classe astratta `AbstractActions`, che ho deciso di implementare poichè tutti i behaviors hanno alcuni metodi che sarebbero comuni.

I metodi di `AbstractActions`, sono:

- **setEnemy(Enemy)**: questo metodo assegna un oggetto Enemy al behavior.
- **getEnemy()**: questo metodo restituisce l'oggetto Enemy che ha assegnato.
- **nextMove()**: modifica la posizione del nemico sulla base della sua direzione.
- **doActions()**: questo metodo , di cui deve essere eseguito l'override necessariamente (perché è un metodo stub), è il più importante di ogni behavior, infatti deve avere tutta la logica "decisionale" del nemico, cioè deve computare il prossimo movimento ed eseguirlo richiamando `nextMove()`.

Ogni classe che estende `AbstractActions` deve necessariamente avere scritto solamente l'override del metodo `doActions()`.

BasicBehavior e HardBehavior

Queste due classi sono i comportamenti corrispondenti alle difficoltà del gioco.

`BasicBehavior` permette al nemico di muoversi randomicamente sulla mappa di gioco, senza tenere conto del giocatore. Tale comportamento viene assegnato al nemico nella modalità facile.

`HardBehavior` permette al nemico di seguire il giocatore, solo se il giocatore si avvicina troppo o viene visto dal nemico. Un oggetto di questa classe, utilizza un algoritmo di pathfinding per calcolare, qualora ci sia, il percorso più breve per raggiungere il giocatore.

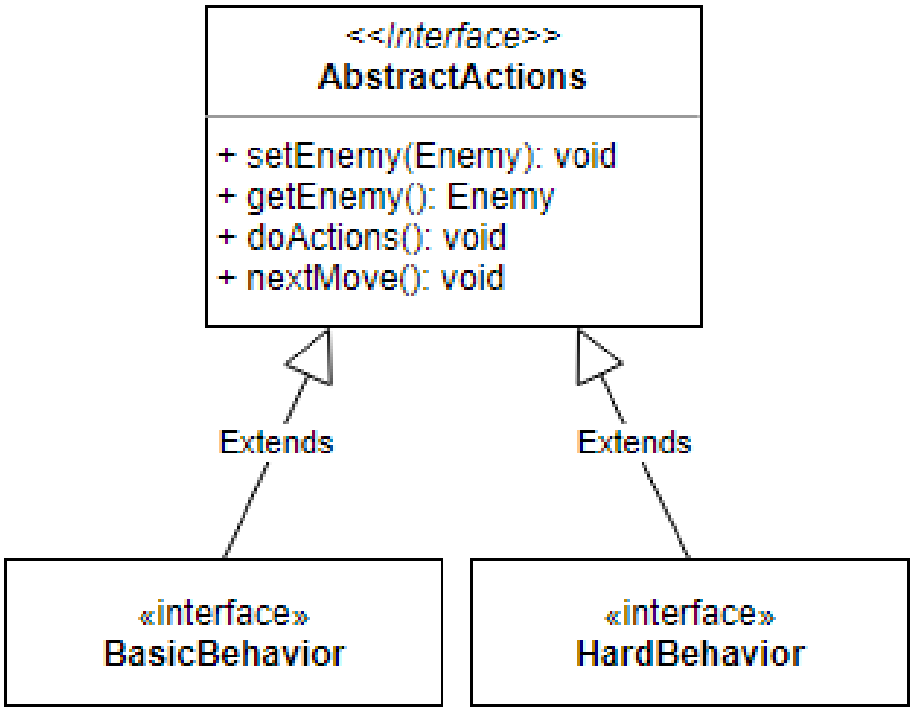

Figura 2.25: Schema UML di `AbstractActions`, `BasicBehavior` e `HardBehavior`

Il package Pathfinding

Una feature importante, quantomeno a mio avviso, alla quale ho dedicato molto impegno è sicuramente il pathfinding, quindi permettere al nemico di trovare un percorso, qualora fosse possibile, per arrivare al player. Ho sviluppato un package apposito con varie classe che collaborano tra loro per raggiungere tale obiettivo. Ho creato la classe `GameBoard` che contiene lo stato della mappa di gioco, in particolare ogni punto della mappa è un oggetto `BoardPoint`, che rappresenta niente di meno che un punto in un sistema di

riferimento bidimensionale, con un valore dell'enumeratore `Markers` che rappresenta lo stato attuale del punto all'interno della mappa. Il pathfinding è stato implementato nell'oggetto `BFSSearch` che costruirà un grafo, composto da oggetti della classe `Node`, man mano che viene eseguita la ricerca ed in caso di successo restituisce una lista di direzioni che portano al player.

Markers

Questo enumeratore ha come valori i possibili stati che un punto della mappa può assumere durante la partita. Gli oggetti `BoardPoint` utilizzano tale enumeratore.

BoardPoint

Questa classe, come già spiegato, costituirà la `GameBoard` e rappresenta un punto in uno spazio bidimensionale, dato che il gioco è in 2D.

Questi sono i metodi:

- **getX()**: restituisce un intero che rappresenta il valore della prima coordinata del punto.
- **getY()**: restituisce un intero che rappresenta il valore della seconda coordinata del punto.
- **getMarker()**: restituisce un valore di Markers che rappresenta lo stato del punto.
- **setX(int)**: modifica il valore della prima coordinata con il valore del parametro passato.
- **setY(int)**: modifica il valore della seconda coordinata con il valore del parametro passato.
- **setMarker(Markers)**: modifica il valore del marker con il parametro passato.
- **setPoint(int, int, Markers)**: modifica le coordinate del punto ed il marker.
- **isEquals(BoardPoint)**: restituisce true se il punto ha le stesse coordinate del `BoardPoint` passato come parametro.

Accessibility

Questo enumeratore è stato creato ad hoc per alcuni metodi che all'interno di `GameBoard` che devono controllare l'accessibilità di certe righe e colonne della mappa di gioco. Ho pensato di svilupparlo perché rende il codice più leggibile e semanticamente robusto, al contrario di un semplice boolean.

GameBoard

Dato che tali algoritmi devono analizzare una struttura dati, ho pensato di implementare la classe `GameBoard`. Con questa classe ho potuto memorizzare lo stato della mappa ad ogni update del gioco, cioè:

- Quante casse sono ancora presenti e che posizione hanno.
- La posizione del giocatore.

La mappa viene costruita come una lista di liste di oggetti BoardPoint (spiegati in seguito), quindi una pseudo matrice dinamica.

Ho deciso di utilizzare una lista di liste (ArrayList), al posto di un grafo, per motivi di complessità spaziale. Infatti un grafo di GameObjects è sicuramente più pesante in memoria, quindi per ridurre lo spazio occupato e ridurre i tempi di accesso ho deciso di optare per questa struttura dati.

I metodi di GameBoard, sono:

- **getGameBoard()**: restituisce un oggetto ArrayList⟨ArrayList ⟨BoardPoint⟩⟩ che rappresenta la mappa di gioco.
- **setGameBoard(ArrayList⟨ArrayList⟨BoardPoint⟩⟩** imposta la sua mappa corrente uguale a quella passata come parametro.
- **getRowsQuantity()**: restituisce il numero di righe della mappa.
- **getColumnsQuantity()**: restituisce il numero di colonne della mappa.
- **findSpotLocation()**: restituisce un oggetto Optional⟨BoardPoint⟩, che in caso di successo della ricerca, contiene il BoardPoint, cioè la posizione del marker spot, altrimenti un oggetto Optional vuoto.
- **isLegal(int, int)**: restituisce true se le coordinate passate come parametri rappresentano una posizione della mappa, altrimenti false.

- **getItem(int, int)**: restitusce un oggetto Optional⟨BoardPoint⟩, che contiene il BoardPoint specificato alla posizione indicata dai due parametri, in caso essa fosse una posizione valida, altrimenti ritorna un oggetto Optional vuoto.
- **setItem(BoardPoint)**: imposta l'elemento passato come parametro alla posizione specificata, quindi ritorna true se l'oggetto indica una posizione valida, altrimenti false.
- **setItems(List⟨BoardPoint⟩)**: imposta gli elementi della lista alle posizioni specificate, qualora siano valide.
- **isAccessible(int, int)**: restituisce true se la posizione passata come paramentro è valida e ha un marker il cui valore sia `ACCESSIBLE` oppure `SPOT`.
- **checkAccessibility(BoardPoint, BoardPoint, Accessibility)**: verifica se i BoardPoint passati come parametro sono sulla stessa riga o colonna. In base al valore del terzo parametro, controlla che le posizioni che intercorrono tra i due punti siano tutte accessibili, quindi restituisce il valore true, altrimenti false.
- **isSpotVisible(int, int)**: restitusce true se la posizione passata come parametri è nella stessa riga o colonna del marker `SPOT` e verifica se i punti di differenza siano accessibili, altrimenti false.
- **setSpotLocation(int, int)**: imposta lo spot nella posizione specificata.
- **searchMarker(Markers)**: cerca la prima occorrenza del marker passato come parametro. Se il marker viene trovato, viene restituito un oggetto Optional⟨BoardPoint⟩che contiene tale marker, altrimenti un oggetto Optional vuoto.
- **changeItem(Markers, Markers)**: modifica la prima occorrenza del marker passato come primo parametro con il secondo parametro, quindi se il primo marker esiste e viene modificato, viene restituito true, altrimenti false.
- **changeAllItems(Markers, Markers)**: modifica tutte le occorrenze del primo marker passato come parametro con il secondo.
- **changeAllItems(List⟨BoardPoint⟩)**: modifica tutti i punti passati come parametro, qualora siano validi.

- **resetItem(Markers)**: imposta la prima occorrenza del marker passato come parametro con il valore `ACCESSIBLE`.
- **resetItem(int, int)**: imposta il marker del `BoardPoint` specificato dalle coordinate passate come parametro con il valore `ACCESSIBLE`.
- **resetAllItems(Markers)**: imposta il valore `ACCESSIBLE` per ogni occorrenza del marker passato come parametro.

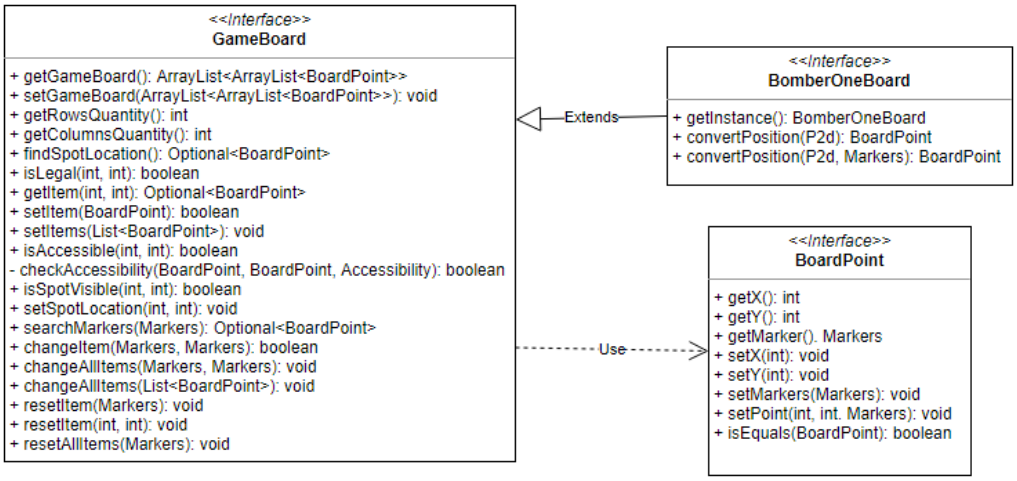

Figura 2.26: Schema UML della `GameBoard`, `BomberOneBoard` e `BoardPoint`

Node

Questa classe viene utilizzata durante l'esecuzione della ricerca del percorso per creare un grafo non ordinato, per tenere conto di tutti i percorsi analizzati.

I metodi sono:

- **getDirection()**: restituisce un valore di Direction che è la direzione con cui è stata percorsa una certa posizione in fase di path finding.
- **getPosition()**: restituisce un oggetto BoardPoint che rappresenta la posizione attraversata durante la ricerca.
- **getParent()**: restituisce il nodo padre del nodo corrente.
- **getPath()**: restituisce una lista di Direction che rappresenta il percorso tra la posizione del nemico fino alla posizione del player.

PathFinder e BFSSearch

Per rendere più espandibile e generico possibile, ho implementato l'interfaccia `PathFinder` all'interno del package `navigation` che rappresenta un generico algoritmo di ricerca di un percorso. Dopodichè ho creato la classe `BFSSearch` che implementa tale interfaccia, ed effettua una ricerca di tipo `Breadth-First Search` (oppure `Ricerca in ampiezza`) per trovare il percorso più breve.

L'uso di un interfaccia favoreggia sia l'espandibilità ma anche la possibilità di assegnare a runtime un algoritmo differente da quello già instaziato, quindi sono riuscito ad utilizzare lo `Strategy` pattern. Riconosco che nel progetto non sono presenti altre classi che implementano questa interfaccia, poiché non erano utili al fine dello sviluppo (cioè ne bastava solamente uno), però sono convinto che sarebbe stato un peccato fare diversamente e che molto potenziale sarebbe andato sprecato.

I metodi dell'interfaccia PathFinder, sono:

- **explored(BoardPoint)**: restituisce true se l'oggetto passato come parametro è già stato analizzato durante la ricerca, altrimenti false.
- **addTargets(Node)**: contrassegna come "scoperti" i nodi vicini al nodo passato come parametro.
- **searchPath(BoardPoint)**: ricerca un percorso che dalla posizione passata come parametro, porta alla posizione della `GameBoard` che ha marker uguale a `SPOT`.

`BFSSearch`, considerato che la implementa, ha gli stessi metodi, e come già spiegato effettua una ricerca di tipo BFS. Ho scelto questo tipo di ricerca perché era la più semplice da realizzare e una delle più efficienti in questo contesto.

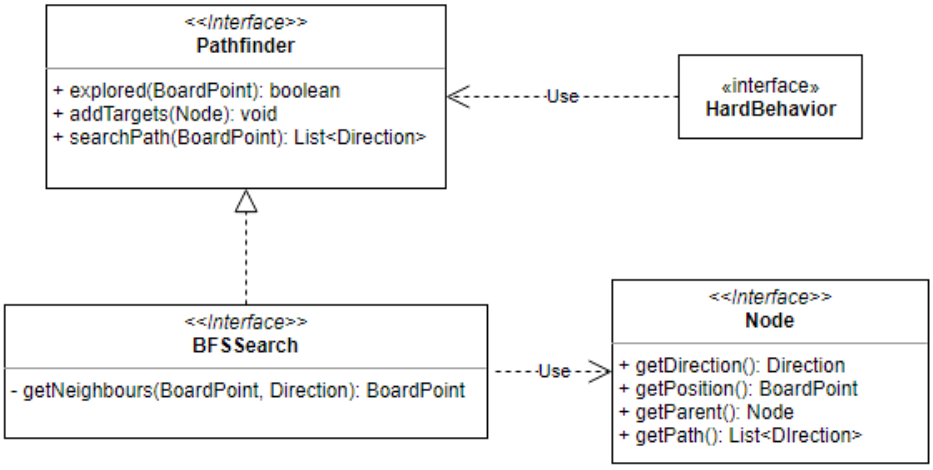

Figura 2.27: Schema UML di `PathFinder` e delle sue interazioni

Come è possibile notare, tutte queste classi sono state sviluppate per cercare di essere più generali possibili e quindi riusabili in altri progetti, di fatto il path finding è ricorrente nei giochi, basti pensare che lo si ha dai tempi di Pacman e di tantissimi altri che sono stati creati in seguito. Per cui, ho pensato che la riusabilità di questo package fosse molto importante, infatti penso si possa vedere un tentativo di creazione di un piccolo framework. Sostengo, però, che un framework indipendente del genere debba essere implementato assieme al game engine, per riuscire ad analizzare meglio tutta la soluzione e ricavarne una maggiore efficienza.

BomberOneBoard

Considerato che, nel progetto è necessaria una sola GameBoard instanziata e condivisibile da tutti i nemici per poter trovare il percorso che porta al player, ho deciso di creare un altro package in cui ho sviluppato la classe `BomberOneBoard` che estende GameBoard e rappresenta una delle mappe create di questo gioco.

Nello specifico, questa classe è stata implementata utilizzando un `Singleton` pattern.

Questa classe ha i seguenti metodi:

- **getInstance()**: restituisce l'instanza dell'oggetto GameBoard.
- **convertPosition(P2d)**: converte il parametro passato in un oggetto BoardPoint e lo restituisce.

- **convertPosition(P2d, Markers)**: converte il primo parametro in un oggetto BoardPoint, assegnandogli il marker, e lo restituisce

La scelta di implementare questa classe in un altro package è per tenere fede alla riusabilità e genericità del package `pathfinding`.

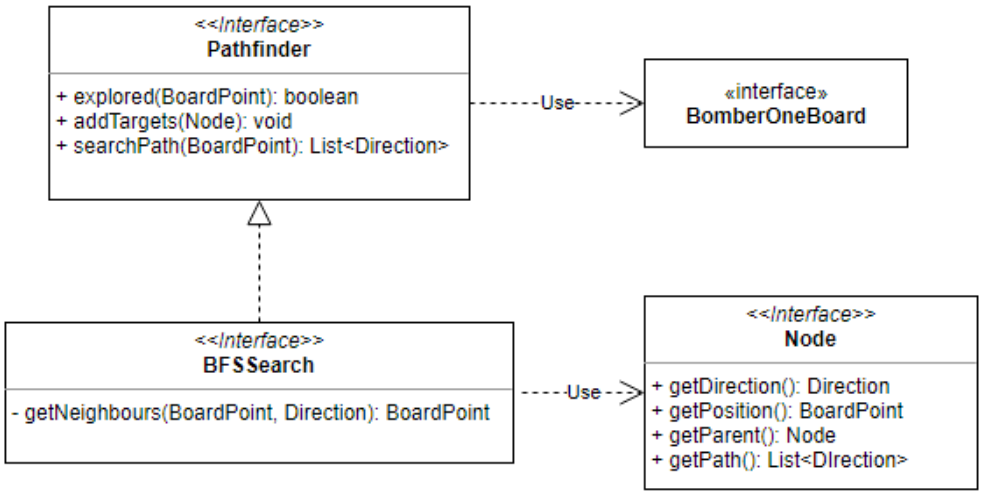

Figura 2.28: Schema UML del path finding framework

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per verificare il corretto funzionamento delle singole parti, riguardanti il software, sono stati realizzati svariati test attraverso l'utilizzo della libreria **JUnit5**, in particolare, abbiamo testato:

- La gestione degli eventi(`WorldEvent`).
- I vari tools.
- Il corretto funzionamento degli oggetti di gioco(esempio. il Fire dopo *"tot"* frame, deve despawnare).

La gestione degli input e la corretta visualizzazione dei vari componenti del gioco sono stati testati manualmente, ovviamente su diversi sistemi operativi e hardware, in quanto non completamente necessari nell'ambito di questo progetto.

3.2 Metodologia di lavoro

Nella fase iniziale del progetto, abbiamo svolto tutti insieme una fase di analisi, nella quale abbiamo lavorato alla progettazione delle parti fondamentali del nostro Software, tramite l'utilizzo di diagrammi UML, in modo tale da poter definire mano a mano, i vari componenti dei quali si sarebbe poi composta la nostra architettura.
Questa fase ha richiesto svariate ore, anche se tuttavia riteniamo fossero fondamentali per la progettazione del Sofwtare.
Una volta terminata la fase di analisi, ci siamo divisi il lavoro cercando di fare

in modo che tutti potessero toccare sia le parti di Model, che di Controller e View.
In seguito, abbiamo proseguito con lo sviluppo individuale, ognuno delle proprie parti, restando sempre e comunque in costante aggiornamento, riuscendo ad aiutarci nei momenti in cui si riscontavano delle difficoltà.
Ovviamente, nelle fase finale sono state svolte molte ore di collaudo del programma.

Come **DVCS** da utilizzare, abbiamo scelto **Git**. Nello specifico, abbiamo lavorato sul branch **"develop"** e ogni tanto sono stati effettuati dei merge di "backup" sul branch "main", fino alla realizzazione della *release* finale. Quando, durante lo sviluppo, occorreva la realizzazione di una feature specifica, essa veniva sviluppata in un branch a parte chiamato **"feature-nomeFeature"**; dopodichè una volta terminata e collaudata, veniva "mergiata" sul branch develop.

3.2.1 Luigi Borriello

- Realizzazione degli oggetti di gioco e del `Timer` relativo alla partita.
- Sviluppo di varie `UtilityClass`.
- Realizzazione e gestione degli eventi(`WorldEvent`).
- Sviluppo del `MatchController`, contenente il GameLoop e sviluppo della `MatchView`.
- Implementazione delle versioni "base" di `View` e `Controller`.

3.2.2 Gustavo Mazzanti

- Realizzazione di due oggetti di gioco (non `"Moveable"`): **Bomb** e **Fire** e l'oggetto **Explosion**, utile per generare i **Fire** tramite la **Bomb**;
- Estensione di **GameObject** in **MoveableObject**, per poter avere un oggetto di gioco `"Moveable"`;
- Implementazione di **Bomber**, l'oggetto di gioco `"Moveable"` principale, il protagonista del videogame;
- Implementazione del package **Input** per la gestione dei **Commands** per tutte le azione del **Bomber**.

- Sviluppo di varie utility class tra cui il tool audio che comprende: **Sounds**, **GameSounds** e **SoundsHandler**, e l'implementazione di un metodo per caricare il font dalla res/ path, metodo implementato all'interno di ResourceLoader (classe realizzata dal collega Borriello con l'aiuto di Brini);

- Realizzazione di tre View: **HomeView**, **RulesView** e **CreditsView** con il relativo controller **HomeController** (Utilizzato per tutte le view appena elencate);

- Disegnate le risorse del gioco, con l'aiuto di un editor grafico (Adobe Illustrator), come bottoni, logo e scritte in tema con il logo.

- Prodotti gli effetti audio attraverso un tool gratuito online per la creazione dei suoni (**ChipTone by SFBGames**);

3.2.3 Tommaso Brini

- Realizzazione del `World` e della relativa Factory.

- Realizzazione della collezione degli oggetti di gioco (`GameObjectCollection`).

- Realizzazione del GameMatch e dell'enumerazione per la difficoltà (`Difficulty`).

- Sviluppo di varie `UtilityClass`(con l'aiuto dei colleghi Borriello e Mazzanti), in particolare la gestione del caricamento della mappa.

- Sviluppo del SetUpController e della SetUpView.

3.2.4 Francesco Agostinelli

- Sviluppo degli `Enemy` e dei comportamenti che loro assumono in base alla difficoltà di gioco scelta, cioè `BasicBehavior` e `HardBehavior`.

- Progettazione e sviluppo del pathfinding system (`GameBoard`, `BoardPoint`, `Markers`, `Accessibility`, `Node` e `Navigation`).

- Progettazione e sviluppo della `RankView`.

3.3 Note di sviluppo

3.3.1 Luigi Borriello

- **Lambda** la dove fosse possibile, per mantenere il codice il più possibile compatto e leggibile.
- **Programmazione Funzionale** per la realizzazione della `ControlsMap`.
- **JavaFX** per la realizzazione delle `View`.
- **Gradle** per preparare il progetto e per rendere semplice l'importazione delle Librerie.

3.3.2 Gustavo Mazzanti

- **Runnable** utilizzato per fare il replay dell'audio nel game.
- **Optional** utilizzati per evitare che il valore di ritorno fosse null.
- **JavaFX** per la realizzazione delle `View`.
- **Gradle** per preparare il progetto e per rendere semplice l'importazione delle Librerie.

3.3.3 Tommaso Brini

- **Lambda** dove possibile.
- **JavaFX** per la realizzazione delle `View`.
- **Gradle** per preparare il progetto e per rendere semplice l'importazione delle Librerie.

3.3.4 Francesco Agostinelli

- Cercare di rendere il codice più pulito, leggibile e conciso possibile.
- Sviluppo del package `pathfinding` come componente software indipendente e riusabile.
- Ottimizzazione degli algoritmi usati, per ridurre la complessità generale.

- **Optional** utilizzati come valori di ritorno per evitare di restituire un valore null oppure `Object`.
- **JavaFx** per la realizzazione della view.

3.4 Crediti

- La maggior parte delle risorse grafiche e l'idea di realizzare un **ResourceLoader** sono state prese da *questo progetto*.
- L'idea di realizzare una **ViewType** ed uno **ViewSwitcher** sono state ispirate da *questo progetto* .
- Le basi audio del videogame sono state recuperate da **questa libreria** open source.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Luigi Borriello

Per quanto mi riguarda, sono molto soddisfatto del risultato ottenuto nel complesso.
Credo di aver dato tutto me stesso e di aver messo in gioco tutte le mie qualità, ma nonostante ciò sono comunque consapevole del fatto che il nostro software non sia perfetto, ad esempio mi sarebbe piaciuto riuscire a realizzare una MatchView meglio ottimizzata, sopratutto nel rendering dei vari oggetti.
So di aver imparato tanto da questo progetto, sopratutto riguardo argomenti che magari durante il corso non sono stati particolarmente approfonditi.
Essendo la prima volta in cui potessi mettere mani su un progetto di tale dimensioni ho capito cosa voglia dire lavorare in Team, quanto sia importante la coordinazione fra i vari membri, e quanto sopratutto sia fondamentale l'analisi svolta nella prima fase.
A malincuore riconosco che nonostante tutto l'impegno dedicatoci, so di aver commesso vari errori durante lo sviluppo, che comunque in qualche modo possano aver rallentato la mia "produttività", d'altro canto però penso che in una prima esperienza sia molto difficile che vada tutto liscio.
In conclusione, credo che questo progetto, mi abbia fatto appassionare ulteriormente alla programmazione ad oggetti e in futuro approfondirò sicuramente le mie conoscenze a riguardo.

4.1.2 Gustavo Mazzanti

Sono molto soddisfatto del lavoro svolto, questa esperienza mi ha dato la possibilità di capire meglio cosa significa lavorare in gruppo, comprendendo al meglio tutti i pregi e i difetti di ciò.
Siamo partiti con una idea e, lavorando inizialmente tutti insieme, abbiamo creato la fondamenta per questo progetto che, dopo aver diviso il lavoro, ha piano piano preso forma, diventando ciò che è ora.
Ho capito che lavorare in gruppo può essere difficoltoso, ma con la giusta cooperazione, siamo riusciti ad completare questo progetto, nonostante le difficoltà incontrate durante lo sviluppo.
L'utilizzo di `Git` è stato fondamentale per la cooperazione, in quanto ha dato la possibilità a ciascuno di poter lavorare alla propria parte senza andare in conflitto con le altre, rendendo il lavoro di gruppo più semplice e ordinato.
Sono contento di tutto il lavoro complessivo, in particolar modo quello svolto dai miei colleghi.

4.1.3 Tommaso Brini

Come già detto dal collega Borriello, anche io penso che possiamo ritenerci veramente soddisfatti dalla realizzazione di questo progetto. È stato un lavoro molto lungo, ma durante questi mesi posso sostenere di avere lavorato con colleghi bravi e volenterosi di portare a termine il compito che ci eravamo prefissati di fare. Per ciò, mi sento di dire che è stata una bella esperienza che mi ha insegnato molto sia dal lato umano, visto l'aggiornamento costante con i colleghi, sia ovviamente dal lato conoscitivo e dell'apprendimento verso la programmazione ad oggetti.
Personalmente, sono consapevole del fatto che il codice scritto da me non sia perfetto e che molte scelte che ho preso possano essere migliorabili, ma essendo il primo progetto di queste dimensioni su cui ho lavorato so che è stato molto utile come punto di partenza per lavori futuri.
Dal punto di vista dell'impegno, non ho niente da rimproverarci, nè da parte mia nè da quella dei miei colleghi. È stato molto utile lavorare con loro, penso di essere cresciuto molto.
Sicuramente, un punto di forza del nostro progetto è stata che l'idea di implementare questo gioco ci ha da subito conquistati e ciò ci ha permesso di organizzarci bene e di portarlo a termine, nonostante tutti gli sforzi.

4.1.4 Francesco Agostinelli

Il progetto mi ha fatto crescere professionalmente, in particolare, oltre ad aver imparato aspetti della progettazione ed implementazione di videogame e essermi tuffato dentro il progetto più serio e complesso che io abbia mai fatto nella mia giovane vita di sviluppatore, mi ha fatto capire ed affrontare il gap che sta tra il "pensare di saper fare" ed il "saper fare" e, almeno per me, è la vera valutazione e premio del progetto. Inoltre, per la prima volta sono riuscito ad avere un team di tutto rispetto in cui si è lavorato in un clima sereno, divertente e serio. Sicuramente, un altro vanto è quello di essere riusciti, con pochissima esperienza nello sviluppo di videogame, ad aver creato un gioco, seppur semplice. In sintesi, sento che posso parlare a nome di tutti, siamo fieri di aver sviluppato questo progetto e di apparire come contributori della repository.

4.2 Difficoltà incontrate e commenti per i docenti

Questo corso è sicuramente quello più interessante e coinvolgente fatto fin'ora, l'unica nota negativa è il quantitativo di tempo impiegato per lo sviluppo che ha ampiamente superato le 80h ciascuno. Si consiglia per il futuro del corso di approfondire alcuni argomenti, come ad esempio JavaFX che nel corso è stata trattata minimamente, in quanto per lo sviluppo della parte grafica è sicuramente più completo di JavaSwing, oppure impiegare qualche lezione in più per approfondire meglio l'utilizzo di git che in alcune situazioni di merge ha causato diverse problematiche, inoltre reputiamo il programma molto utile anche per altri insegnamenti, tanto che alcuni di noi stanno gia lavorando ad altri progetti con l'aiuto di git.
Un problema riscontrato è stata la cooperazione che, a volte, è risultata difficoltosa per via di alcune dipendenze che potevano risultare tra due colleghi. Nonostante ciò, la cooperazione e il buon lavoro ha portato a una soluzione finale di cui andiamo tutti molto fieri e sicuramente questa esperienza ci ha insegnato tanto a tutti.

Appendice A

Guida utente

Al lancio dell'applicazione, si aprirà la schermata della Home. Qui l'utente troverà un menù in cui scegliere se avviare una nuova partita, vedere le classifiche oppure consultare le regole. Nella schermata della Home è stato inserito anche un piccolo EasterEgg di riconoscimento per i programmatori.

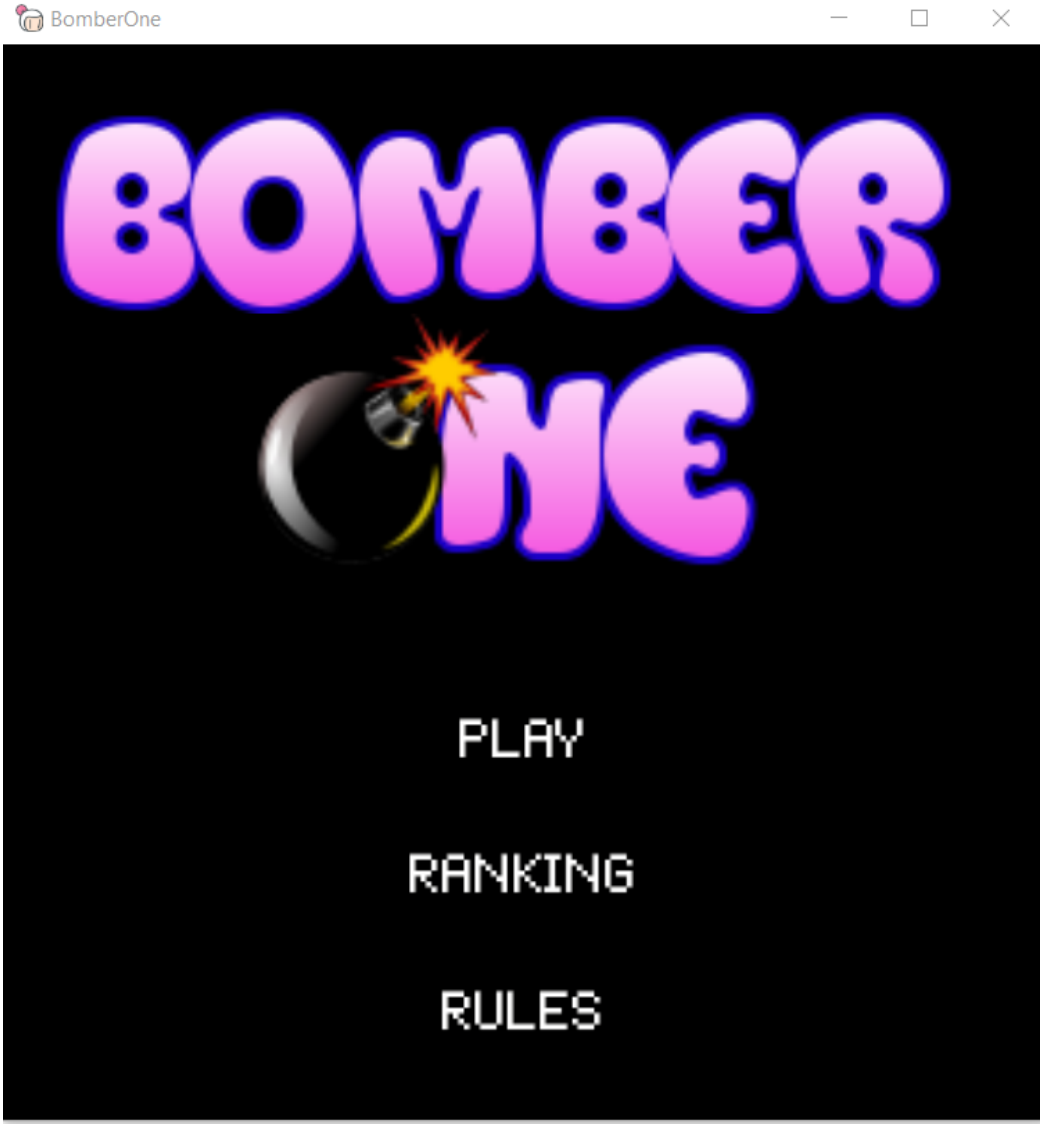

Figura A.1: View della `Home`.

Una volta spinto PLAY per iniziare la partita, verrà lanciata la schermata delle applicazioni, in cui OBBLIGATORIAMENTE sarà inserito il NickName e scelte le impostazioni per la difficoltà e per i comandi da usare (scegliere tra usare le frecce oppure i comandi WASD). Per entrambi i comandi scelti, durante la partita saranno lanciate le bombe premendo la barra spaziatrice.

Figura A.2: View della `SetUp`.

Una volta scelte le impostazioni, sarà possibile iniziare la partita. Dopo aver giocato, verranno mostrate le classifiche delle due modalità, EASY e HARD. In qualsiasi schermata, c'è la possibilità di tornare direttamente alla Home.

Figura A.3: View del `Match`.

Appendice B

Esercitazioni di laboratorio

B.0.1 Luigi Borriello

- Laboratorio 06: `https://github.com/luigi-borriello00/OOP-Lab06`
- Laboratorio 07: `https://github.com/luigi-borriello00/OOP-Lab07`
- Laboratorio 08: `https://github.com/luigi-borriello00/OOP-Lab08`
- Laboratorio 09: `https://github.com/luigi-borriello00/OOP-Lab09`
- Laboratorio 10: `https://github.com/luigi-borriello00/OOP-Lab10`

B.0.2 Gustavo Mazzanti

- Laboratorio 06: `https://github.com/9gusgusgus9/OOP-Lab06`
- Laboratorio 07: `https://github.com/9gusgusgus9/OOP-Lab07`
- Laboratorio 08: `https://github.com/9gusgusgus9/OOP-Lab08`
- Laboratorio 09: `https://github.com/9gusgusgus9/OOP-Lab09`
- Laboratorio 10: `https://github.com/9gusgusgus9/OOP-Lab10`

B.0.3 Tommaso Brini

- Laboratorio 06: `https://github.com/TommasoBrini/OOP-Lab06`
- Laboratorio 07: `https://github.com/TommasoBrini/OOP-Lab07`
- Laboratorio 08: `https://github.com/TommasoBrini/OOP-Lab08`
- Laboratorio 09: `https://github.com/TommasoBrini/OOP-Lab09`