Hotline Cesena

Bryan Corradino
Andrea Zammarchi
Federico Campanozzi
Andrea Micheli

24 aprile 2021

Indice

Capitolo 1

Analisi

L'applicazione Hotline Cesena emula il noto top-down shooter "Hotline Miami", videogioco d'azione in cui il giocatore, equipaggiato con un'arma da fuoco, deve muoversi all'interno di una mappa popolata da svariati nemici e portare a termine determinate missioni. Ogni nuova partita prevede la generazione casuale del layout della mappa e delle entità presenti al suo interno. Il gioco termina al completamento di tutte le missioni o alla morte del giocatore. Inoltre, i punteggi ottenuti alla fine di ogni partita verranno registrati in una classifica generale.

1.1 Requisiti

1.1.1 Requisiti funzionali

- Il giocatore dovrà essere in grado di spostarsi liberamente ed eseguire azioni in base agli input dell'utente;
- Come in un top-down shooter, la telecamera di gioco dovrà seguire dall'alto gli spostamenti del giocatore e centrare la visuale su di esso;
- I nemici, tramite l'utilizzo di una propria strategia di movimento, saranno in grado di attraversare la mappa e reagire alla presenza del giocatore;
- Il giocatore dovrà avere la possibilità di raccogliere e utilizzare oggetti e armi presenti nel mondo di gioco;
- Le armi dovranno sparare proiettili in grado di viaggiare lungo la mappa e danneggiare le entità con le quali entreranno in contatto;

- La partita dovrà terminare al completamento di tutte le missioni o alla morte del giocatore.

1.1.2 Requisiti non funzionali

- Il gioco calcolerà il punteggio totale ottenuto a fine partita tenendo conto di alcuni parametri;
- L'utente potrà essere in grado di modificare le impostazioni dell'applicazione;
- Ad ogni azione eseguita dal giocatore o dai nemici dovrà essere associata la riproduzione di una risorsa audio;
- Il gioco dovrà risultare fluido, senza cali di prestazioni.

1.2 Analisi e modello del dominio

In Hotline Cesena, le varie entità interagiscono fra loro all'interno di un mondo di gioco generato in modo casuale all'inizio di ogni partita. La mappa si compone di stanze collegate tra loro e al cui interno si possono trovare nemici, armi e/o oggetti.
Il giocatore è in grado di spostarsi liberamente (a meno che non entri in collisione con ostacoli o pareti), di sparare con la propria arma e di interagire con gli oggetti sulla mappa in risposta ai comandi impartiti dall'utente. Inoltre, dovrà portare a termine alcune missioni le cui condizioni di completamento verranno calibrate in base alla composizione della mappa.
I nemici sono suddivisi per tipologia, ciascuno con un proprio schema di movimento, e sono in grado di rilevare la presenza del giocatore qualora quest'ultimo entri nel loro campo visivo o produca forti rumori (ad esempio, sparare o correre).
Gli oggetti sparsi nell'ambiente di gioco possono conferire bonus al giocatore o essere richiesti per alcune particolari missioni.
A fine partita, verrà calcolato un punteggio relativo alle abilità del giocatore e al tempo trascorso.

Nella **Figura 1.1** è possibile trovare una rappresentazione schematica degli elementi del dominio appena descritti.

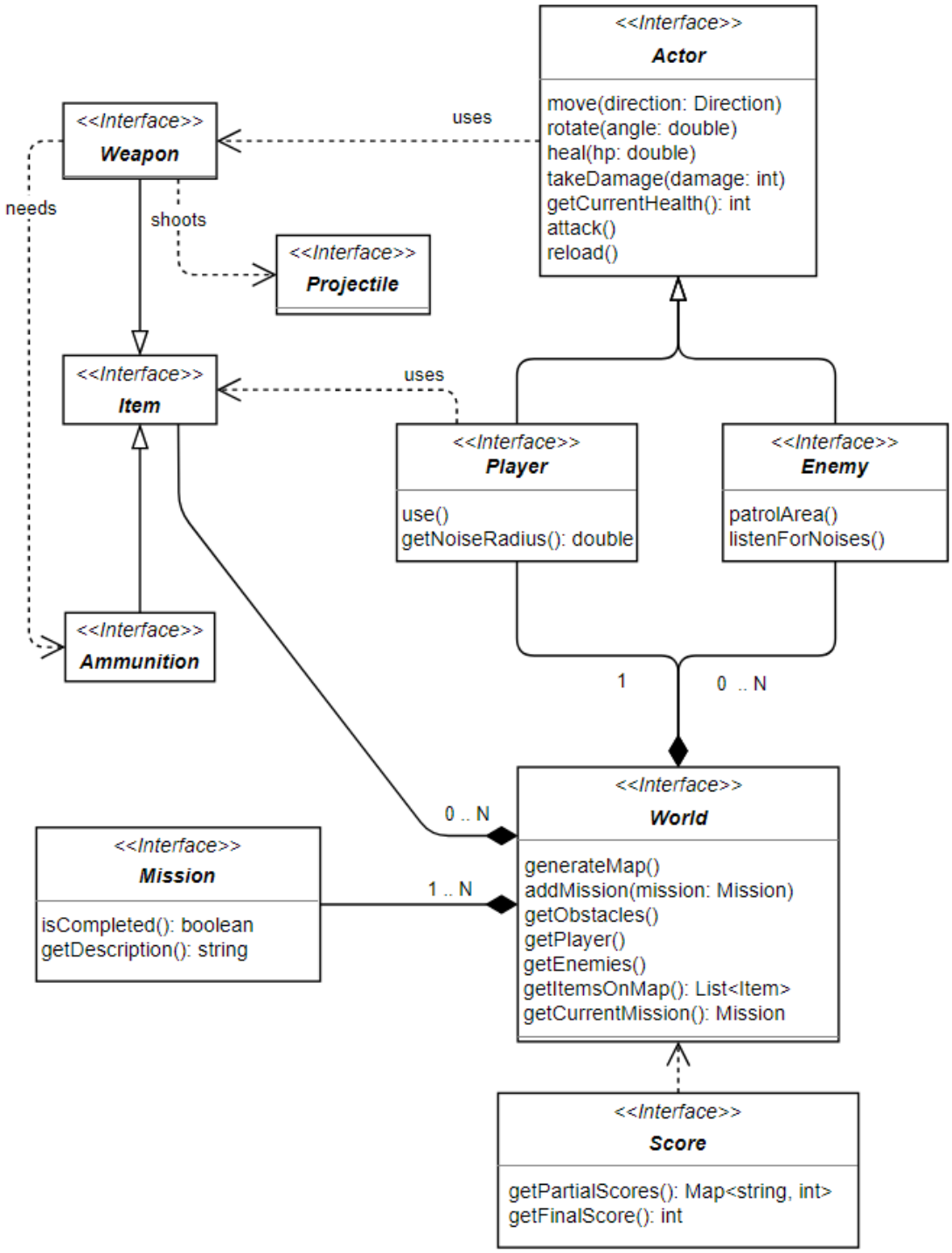


Figura 1.1: Rappresentazione UML del dominio applicativo.

Capitolo 2

Design

2.1 Architettura

La struttura dell'applicazione segue il pattern architetturale **MVC** (Model-View-Controller).

Interazione Model-Controller

Come mostrato in Figura 2.1, il punto di ingresso del Model è rappresentato dall'interfaccia `DataAccessLayer`, il cui ruolo è quello di istanziare ciascuna entità del dominio e di renderle tutte sempre disponibili non solo alle entità stesse, ma anche ai vari componenti del Controller: di conseguenza, `DataAccessLayer` è stato progettato come *singleton*.
All'avvio della partita, su richiesta del `WorldController`, `DataAccessLayer` crea la mappa di gioco tramite il componente `WorldGeneratorBuilder`: sarà compito di quest'ultimo generare un layout della mappa sempre diverso. Ultimato questo processo, il `WorldController` distribuisce le entità del dominio ai rispettivi controller specializzati.

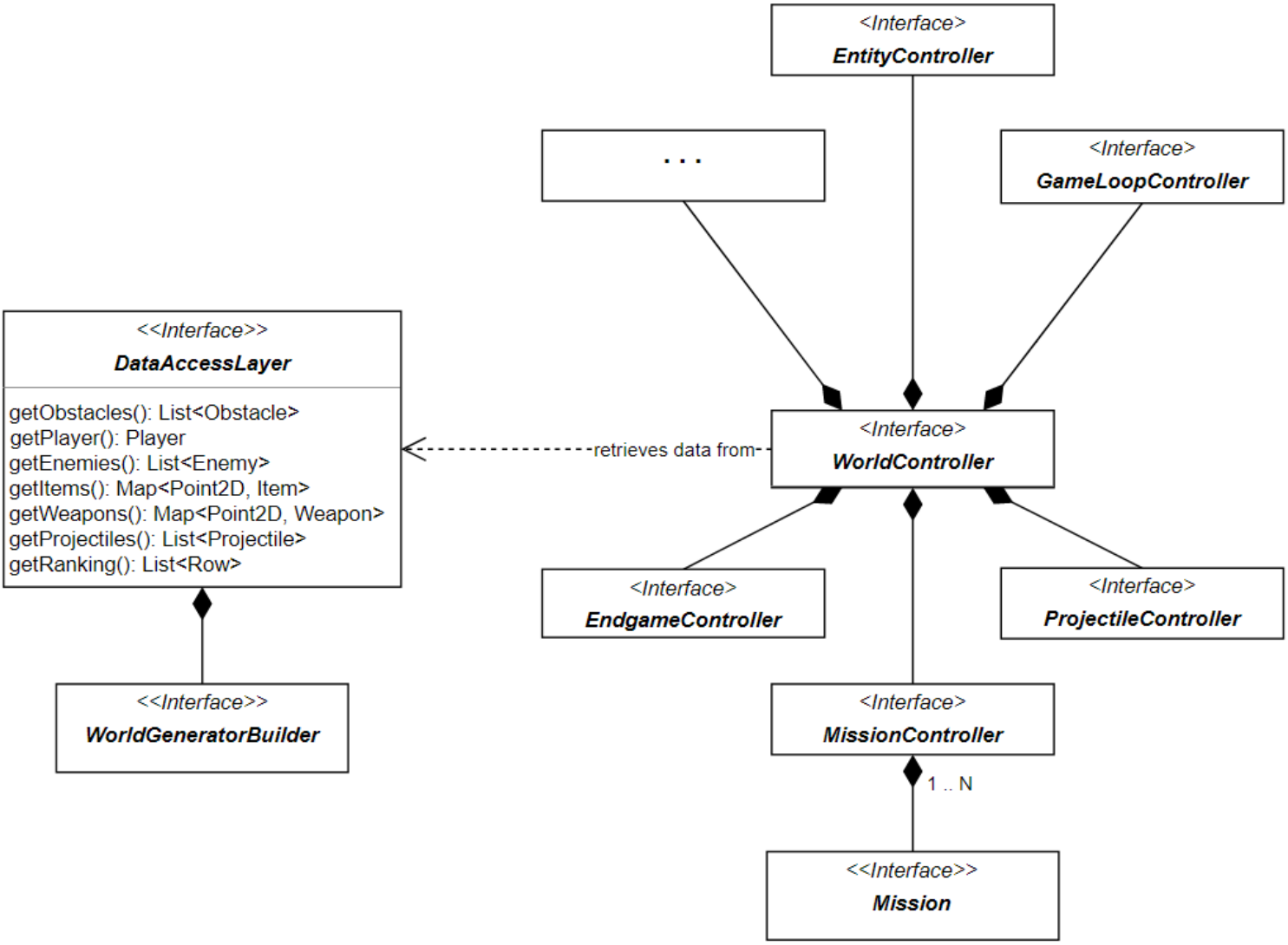


Figura 2.1: Coordinamento tra Model e Controller. Si è scelto di riportare solo le interfacce del Model che hanno subìto cambiamenti dopo la fase di analisi.

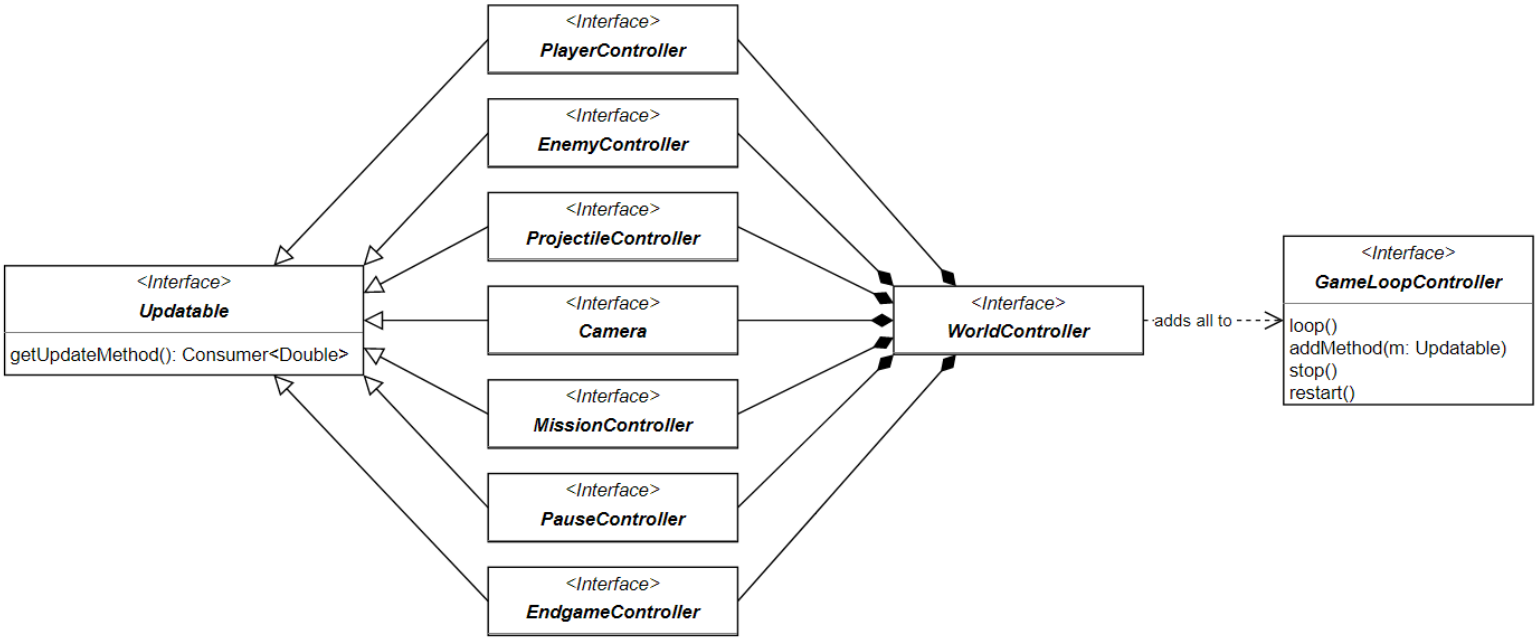


Figura 2.2: Controller del *Game Loop*.

Il *Game Loop* è una parte fondamentale per il funzionamento dell'applicazione. Il compito dell'interfaccia `GameLoopController`, rappresentata in fig. 2.2, è richiamare ciclicamente ogni controller per il quale sia necessario effettuare aggiornamenti di stato ad ogni frame.

Interazione Controller-View

In Figura 2.3 è rappresentata l'interazione tra le componenti di Controller e quelle di View.
Contestualmente alla generazione del mondo di gioco, il `WorldController` ordina all'interfaccia `WorldView` di disegnare e visualizzare la scena riflettendo la composizione della mappa. A ciascuna `Entity` del dominio viene associata una rappresentazione grafica nella View attraverso l'interfaccia `Sprite`; questa è stata progettata in modo da consentire di aggiornarne in tempo reale la posizione, la rotazione e l'immagine visualizzata.
Nel rispetto dei requisiti elencati in fase di analisi, si è deciso di progettare la telecamera di gioco facendole seguire i movimenti di uno specifico `Sprite` che, a conti fatti, sarà sempre quello del giocatore. Ad ogni spostamento, `Camera` traslerà la scena di gioco per simulare il comportamento della telecamera di *Hotline Miami*.
Per quanto riguarda la gestione degli input dell'utente, le pressioni di tutti

i tasti in un dato istante vengono registrate dall'interfaccia `InputListener` e distribuite con facilità tramite il `WorldController` alle componenti che ne necessitano.

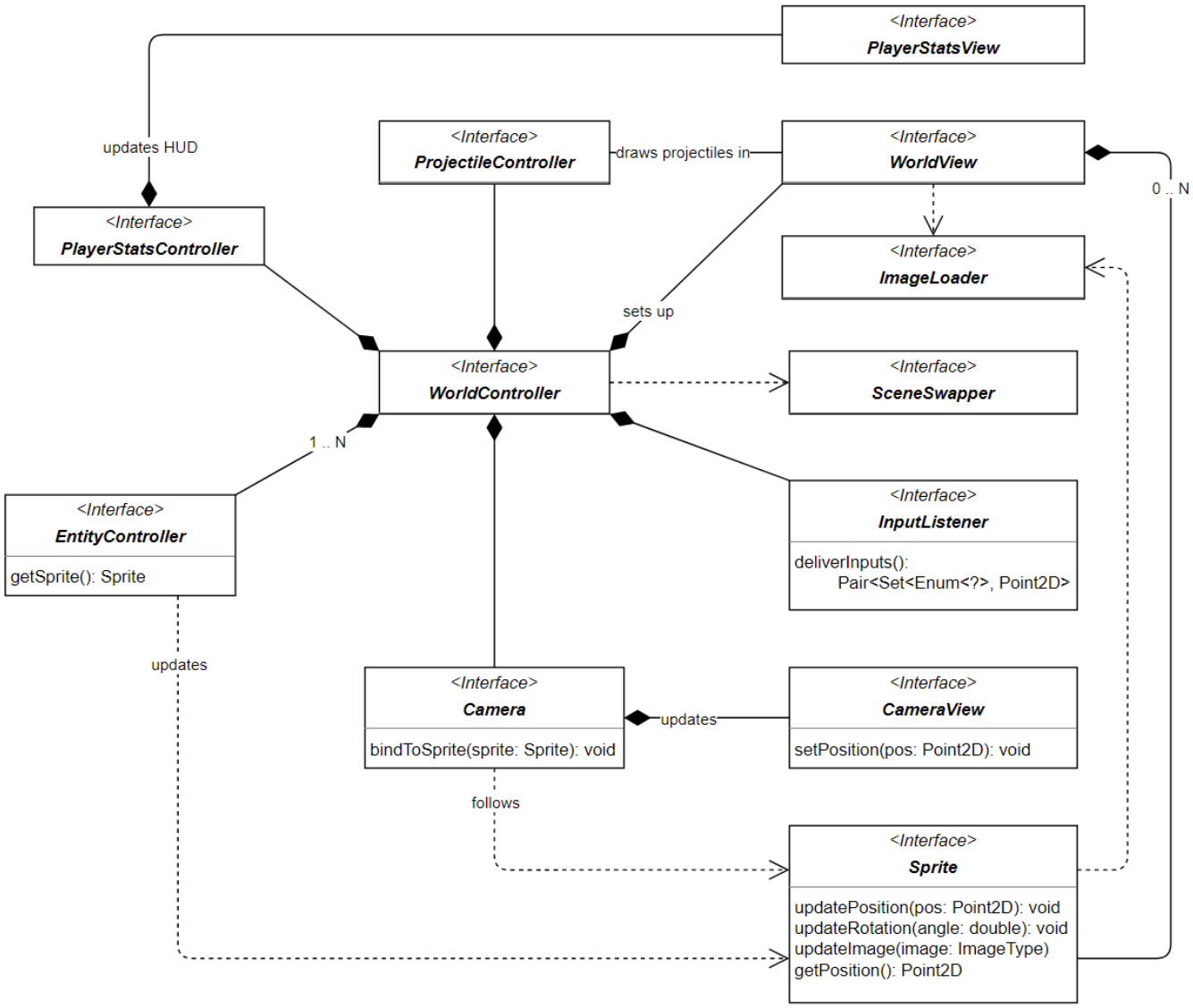


Figura 2.3: Coordinamento tra Controller e View.

Una prima problematica da subito riscontrata in questa fase di progettazione è stata la gestione di eventi asincroni quali la riproduzione di effetti sonori e gli aggiornamenti delle immagini di gioco. Tramite l'applicazione del pattern *Publish-subscribe* (fig. 2.4), è stato possibile associare a ogni azione delle entità del Model un evento ad-hoc, permettendo al Controller di aggiornare in modo semplice e rapido la View o di riprodurre clip audio quando necessario.

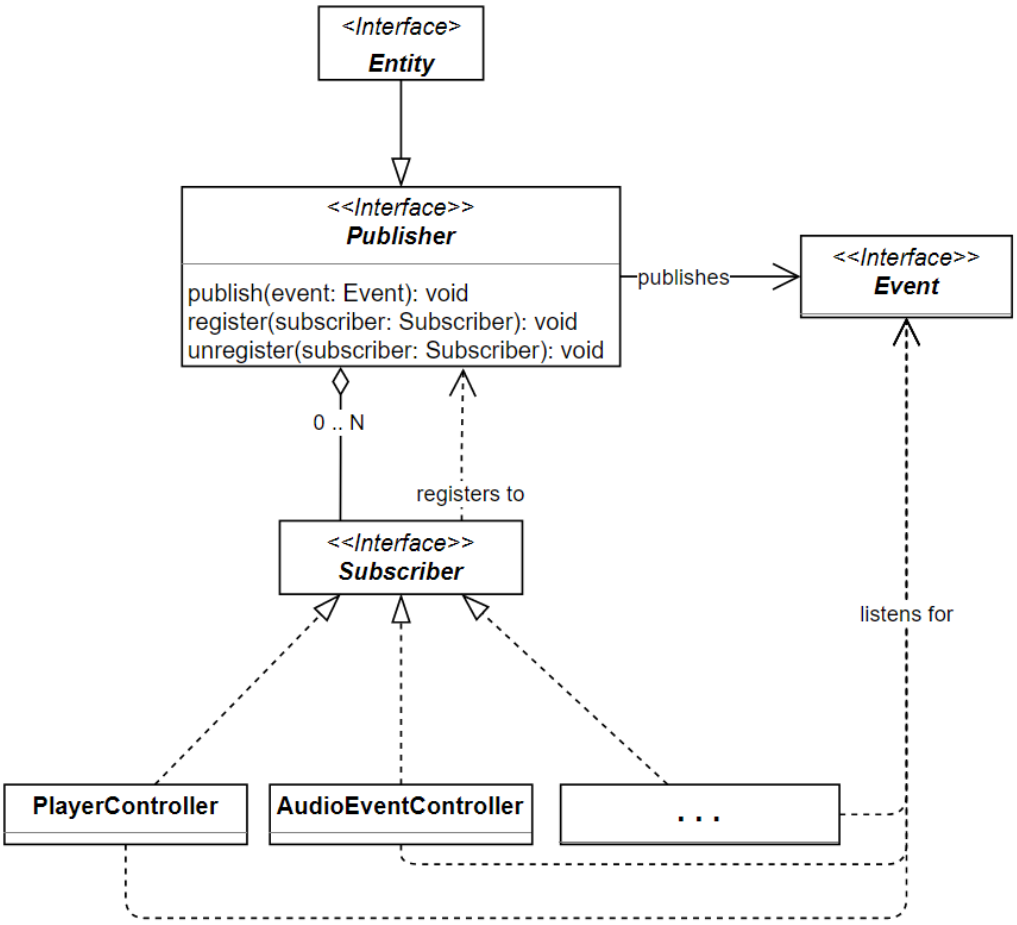


Figura 2.4: Pattern *Publish-subscribe*.

Il componente a cui viene affidato il caricamento delle diverse schermate dell'interfaccia (menu iniziale, menu di pausa, ecc.) è il `SceneSwapper` (fig. 2.5), in grado di fornire una fluida transizione tra scene e un buon grado di estensibilità nel caso in cui si vogliano aggiungere nuovi tipi di menu.

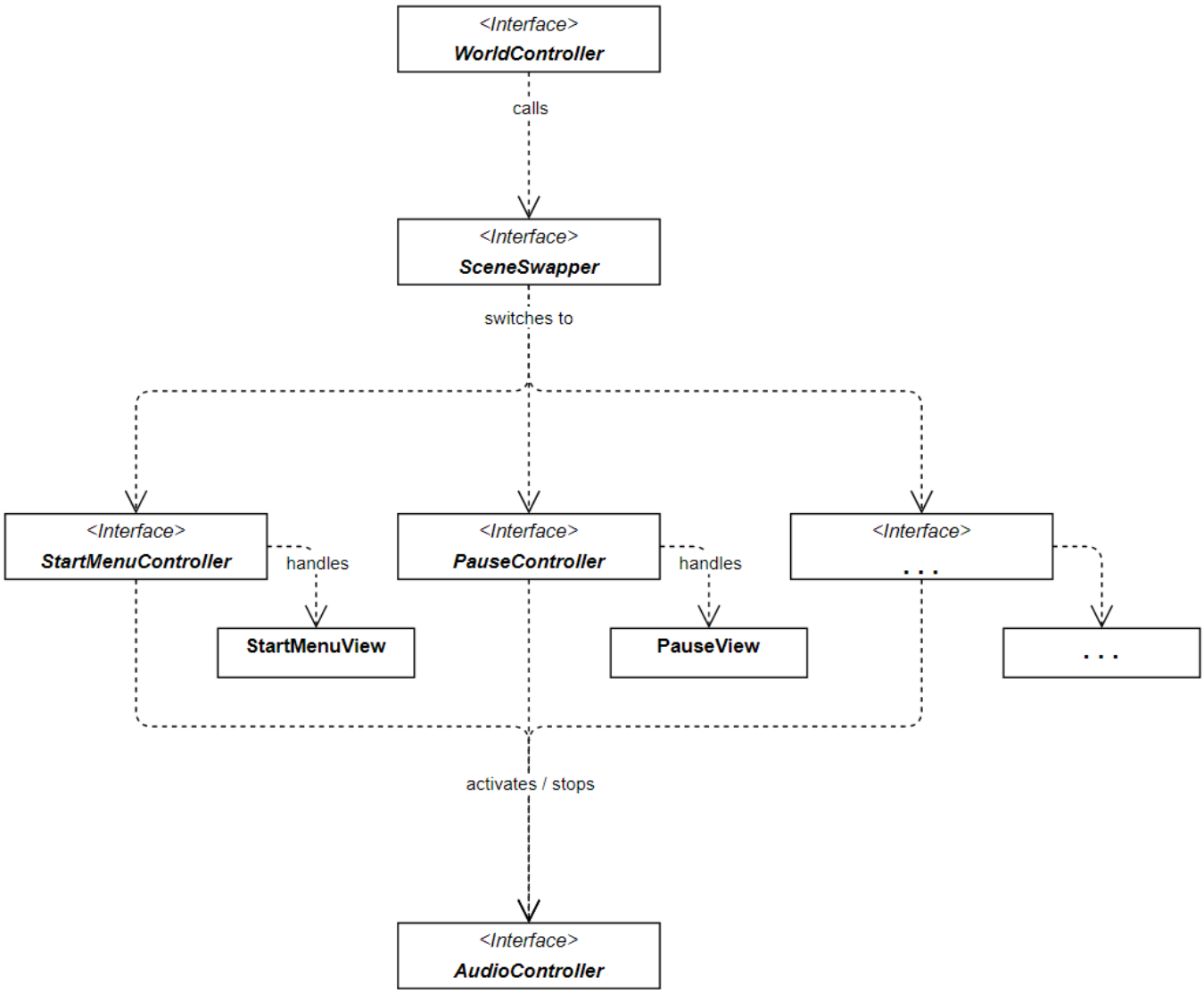


Figura 2.5: Ruolo di SceneSwapper all'interno della View.

2.2 Design dettagliato

2.2.1 Bryan Corradino

Il mio ruolo all'interno del gruppo prevedeva principalmente la realizzazione delle entità di gioco e del giocatore, del sistema di input, della telecamera e del calcolo del punteggio.

Gerarchia delle entità

Di comune accordo con gli altri membri del gruppo, si è deciso di modellare le entità del gioco secondo una gerarchia di interfacce, come mostrato in **Figura 2.6**.
`Entity` è l'interfaccia di base che caratterizza tutte le entità di gioco (eccezion fatta per gli `Item`, di Andrea Zammarchi) con aspetti fondamentali quali una posizione sulla mappa, una larghezza e un'altezza. Tutte le specializzazioni di `Entity` aggiungono nuovi tipi di azioni, come la possibilità di muoversi, di attaccare, ecc.
Al fine di consentire un riutilizzo massimo del codice nelle implementazioni dei nemici (di Andrea Micheli) e dei proiettili delle armi (di Andrea Zammarchi), ho fornito implementazioni complete per tutti i metodi di tutte le specializzazioni di `Entity` mostrate nello schema, tranne che per il metodo `MovableEntity::move`: questo è stato reso un *Template Method* e illustrato più nel dettaglio in un paragrafo successivo.

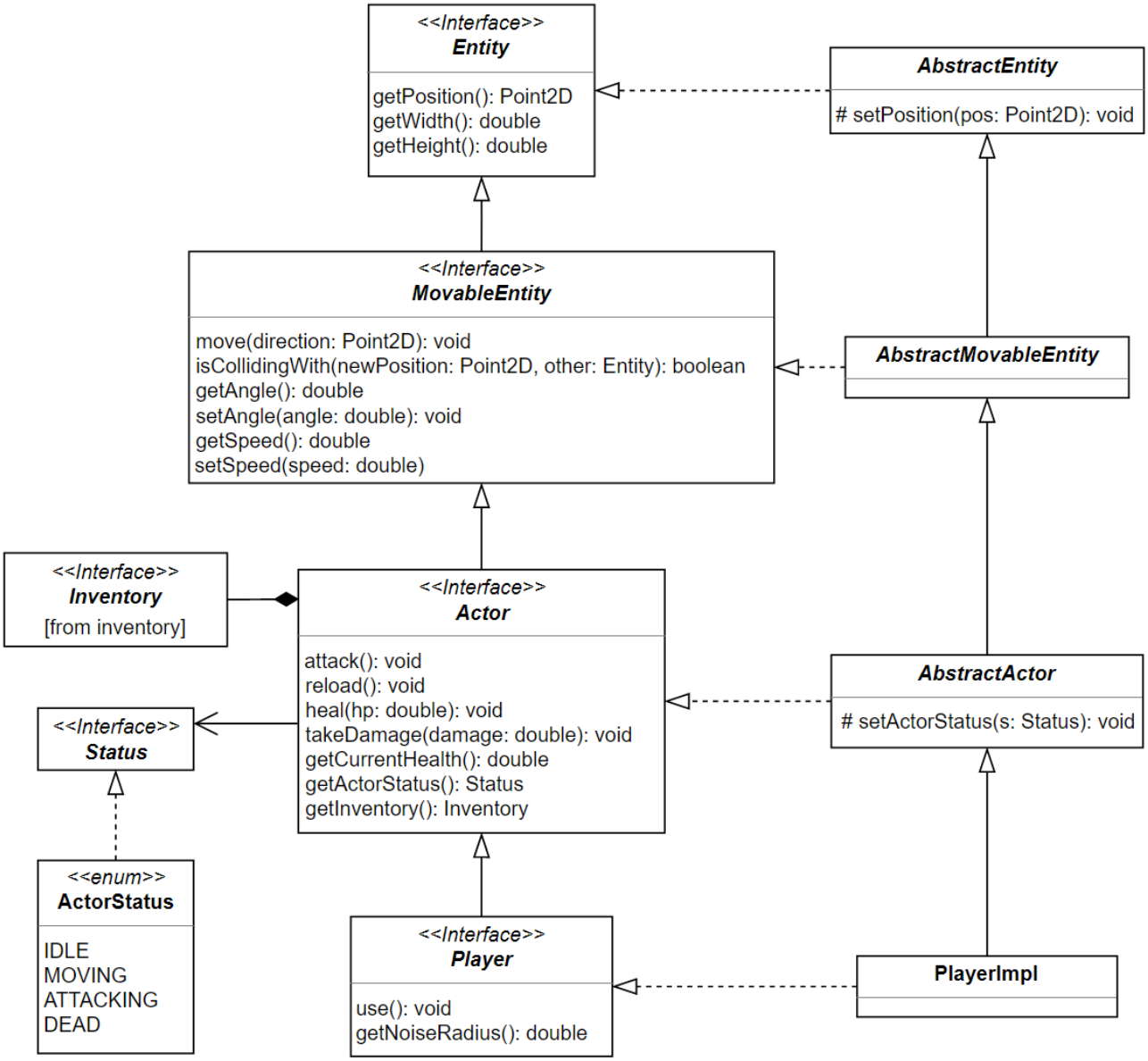


Figura 2.6: Gerarchia delle entità.

Come già anticipato in fase di analisi del dominio, un `Actor` è caratterizzato da un certo quantitativo di punti salute e può subire danni se colpito da un proiettile. Per notificare alle altre entità il proprio stato in un dato istante (ad esempio, se in movimento o se morto), un `Actor` mantiene al suo interno un campo `Status`, che viene modificato all'esecuzione di determinate azioni.

Oltre ad avere accesso a tutte le azioni eseguibili da `Actor` (come già mostrato in fig. 2.6), `Player` può raccogliere armi e oggetti presenti sulla mappa di gioco. Inoltre, poiché ci aspettiamo che i nemici reagiscano ai suoni prodotti dal giocatore, ho fornito anche un metodo che, dipendentemente dallo stato corrente del `Player` (indicato dal suo `Status`), restituisce il raggio del rumore prodotto.

Template Method

In Figura 2.7 è stata schematizzata l'applicazione del pattern *Template Method* per il metodo `MovableEntity::move`. L'impiego di questo pattern si è reso necessario poiché tutte e tre le implementazioni concrete di `MovableEntity` richiedono una logica di movimento diversa e una gestione delle collisioni personalizzata.

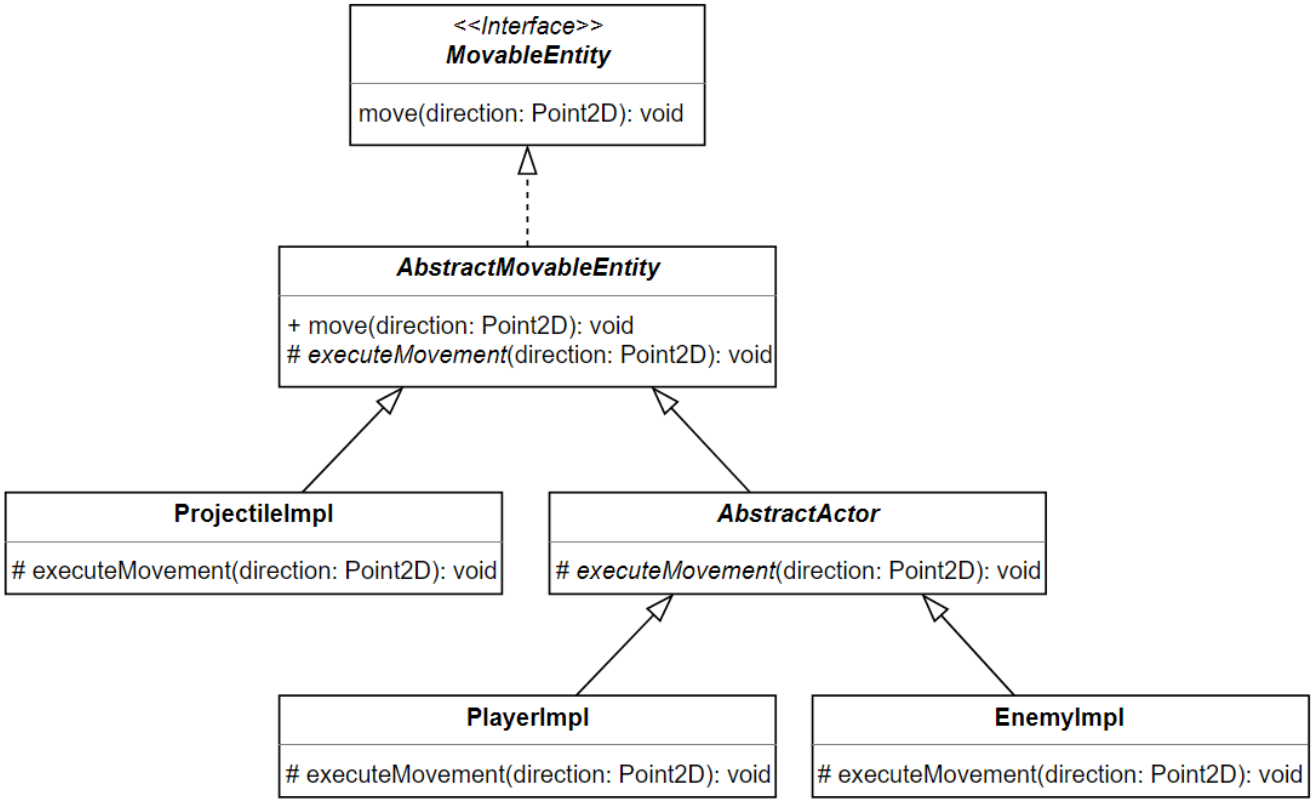


Figura 2.7: Applicazione del pattern Template Method.

Inventario

Poiché tutte le implementazioni di `Actor` fanno uso di armi e devono dunque avere la possibilità di trasportare munizioni e oggetti collezionabili, ho deciso di delegare la gestione dell'inventario e dell'azione di ricarica delle armi a un componente separato, chiamato `Inventory` (Figura 2.8).
L'interfaccia espone a tutti gli `Actor` anche metodi che, allo stato attuale delle cose, vengono sfruttati solo da `Player`: ad esempio, se l'azione di raccogliere oggetti sulla mappa la si volesse estendere anche ai nemici, `Inventory` e la sua implementazione `NaiveInventoryImpl` consentirebbero di farlo senza alcun problema.
Per motivi di bilanciamento della difficoltà del gioco, `NaiveInventoryImpl` permette di trasportare solo un'arma per volta, sebbene l'interfaccia preveda la possibilità di possederne anche di più.

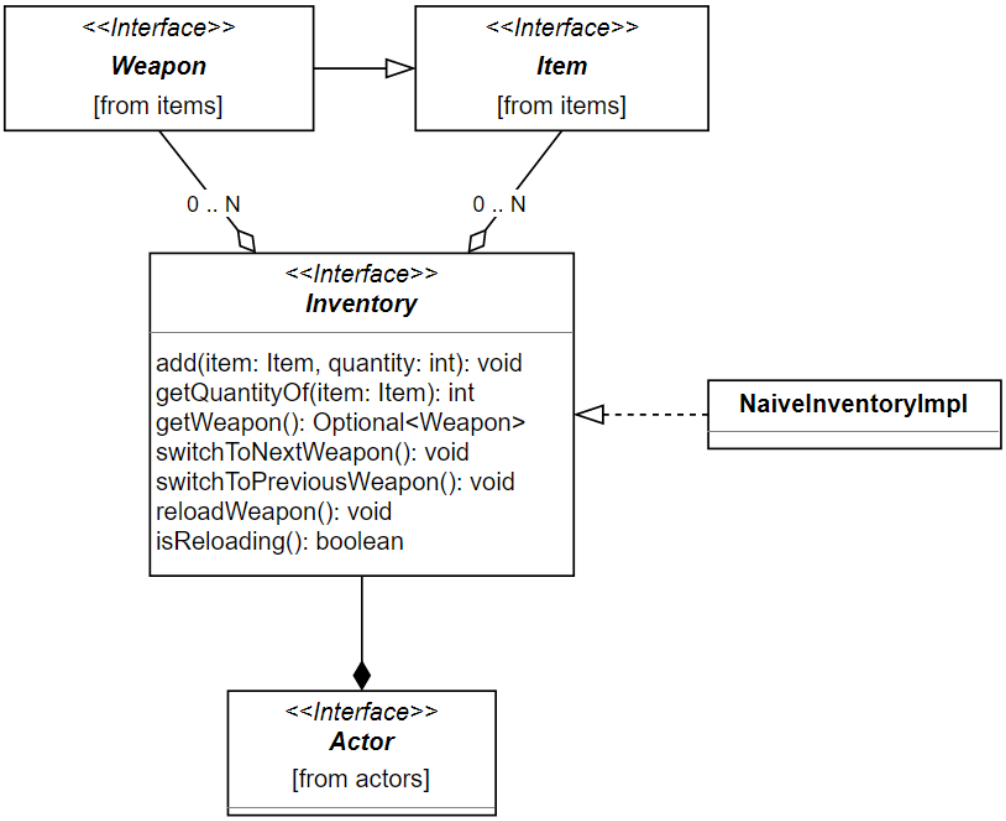


Figura 2.8: Schema dell'inventario.

Factory per l'inventario

In fig. 2.9 viene mostrato l'utilizzo del pattern *Factory* per facilitare la creazione di inventari contenenti insiemi predefiniti o personalizzati di armi e oggetti.

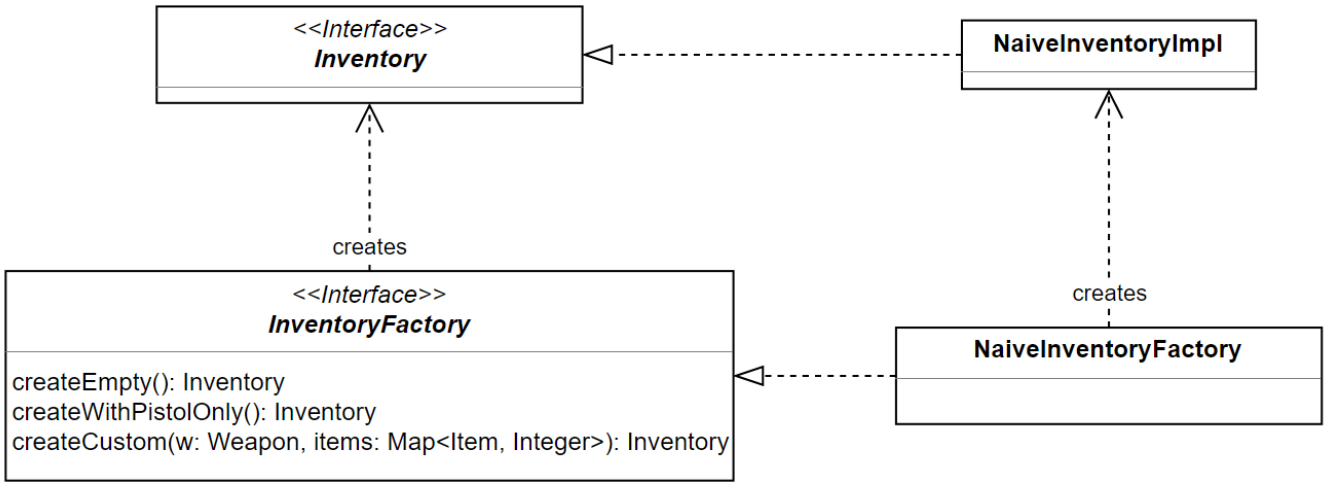


Figura 2.9: Applicazione del pattern *Factory* per l'inventario.

PlayerController e *Command*

Per la gestione del giocatore, la difficoltà principale risiedeva nel tradurre gli input impartiti dall'utente facendo attenzione a non mescolare aspetti puramente di Controller e di View con la parte di Model. Inoltre, necessitavo di un design modulare che mi consentisse di testare singolarmente le varie componenti del `PlayerController` senza che queste presentassero tra di loro dipendenze imprescindibili.
Al fine di disaccoppiare nel modo più marcato possibile le interfacce e minimizzare il numero di dipendenze interclasse, il processo di ascolto e consegna di input "grezzi" viene delegato all'`InputListener`, mentre il passaggio di effettiva interpretazione degli input è stato reificato nell'interfaccia `InputInterpreter`. (**fig. 2.10**).
L'`InputListener` è un componente di View e viene utilizzato all'interno del progetto come punto centrale di raccolta di tutti gli input inviati dall'utente per mezzo di mouse e tastiera. Mantenendo un riferimento a un oggetto di questa classe all'interno del `PlayerController`, posso ottenere con facilità l'insieme di tutti i tasti premuti e le coordinate del mouse sulla finestra di gioco.

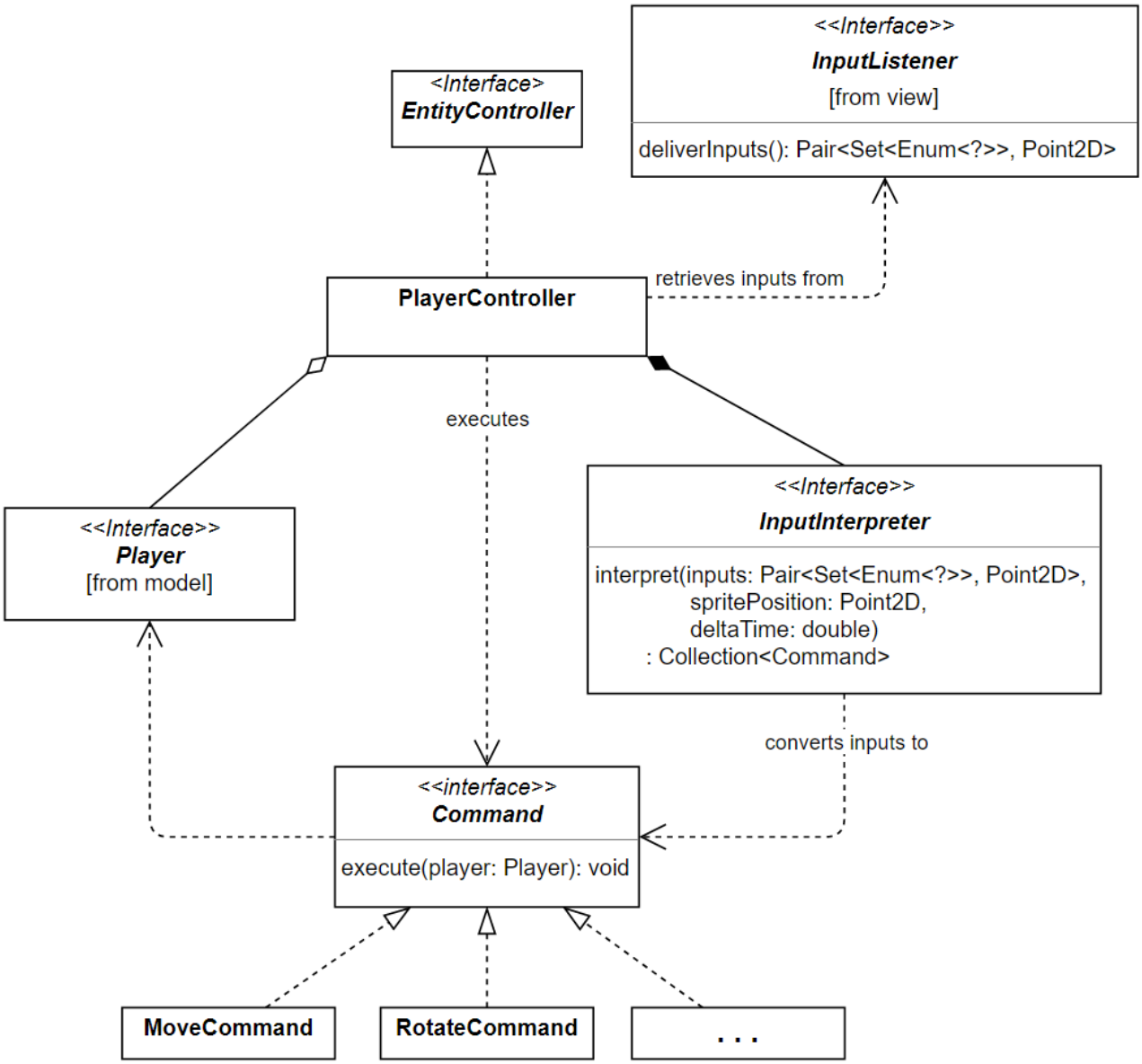


Figura 2.10: PlayerController e applicazione del pattern *Command*.

L'`InputInterpreter`, invece, mediante l'applicazione del pattern *Command*, si occupa di trasformare gli input raccolti dall'`InputListener` in comandi che incapsulino i metodi dell'interfaccia `Player`, di consegnarli al `PlayerController` e lasciare a quest'ultimo il compito di eseguirli uno a uno all'interno del *game loop*.
Questo approccio introduce due importanti vantaggi:

1. è possibile implementare facilmente una funzionalità di personalizzazione degli assegnamenti dei tasti ai vari comandi (cosa che, purtroppo, non è stata fatta per mancanza di tempo);

2. la transizione a una diversa libreria grafica non impatta né l'implementazione del `PlayerController`, né quella dell'`InputInterpreter`.

Infine, l'ultima funzione svolta dal `PlayerController` è quella di riflettere i cambiamenti di stato del giocatore anche sulla View. Dunque, nel rispetto dell'architettura del progetto, si è scelto di rendere il `PlayerController` un *Subscriber* dell'entità `Player` (fig. 2.11): ciò significa che, in reazione a un evento di movimento, di raccolta di una nuova arma, ecc., lo `Sprite` relativo al giocatore verrà aggiornato di conseguenza.

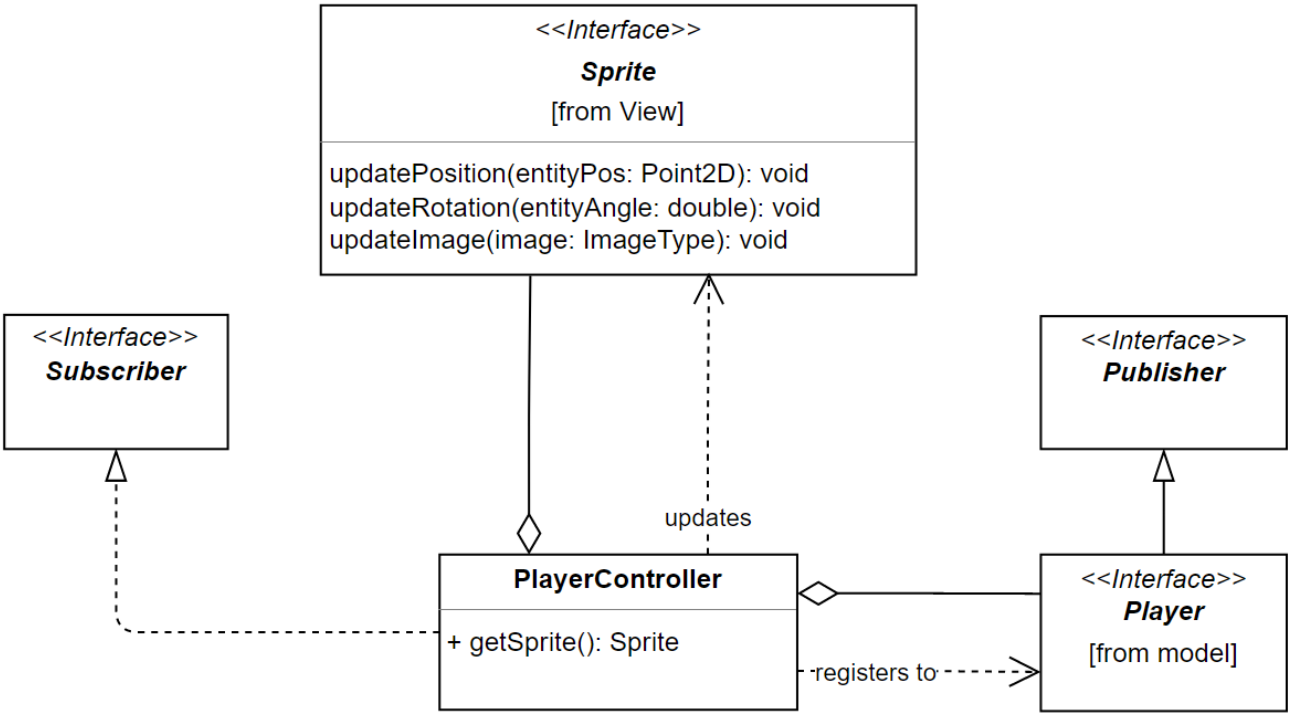


Figura 2.11: Interazione tra PlayerController e View.

Factory per il PlayerController

In fig. 2.12 viene illustrata l'applicazione del pattern *Factory* per la creazione del `PlayerController`. La factory istanzia per il `PlayerController` una nuova `InputInterpreterImpl` e vi assegna una mappa contenente associazioni predefinite di tasti-comandi.

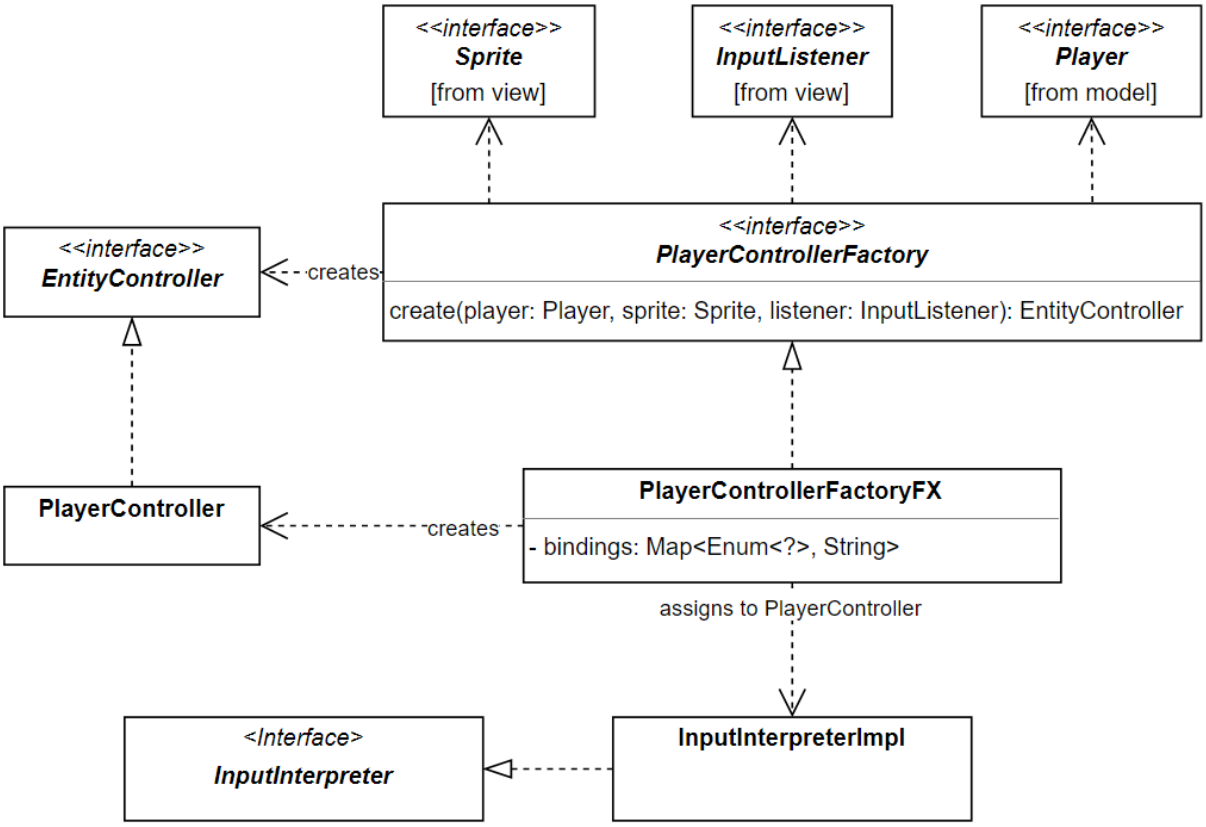


Figura 2.12: Schema di funzionamento della `PlayerControllerFactory`.

Publish-subscribe

Per l'applicazione del pattern *Publish-subscribe*, ho cercato di fornire un design che fosse intuitivo e facilmente utilizzabile dagli altri membri del gruppo. La **fig. 2.13** si rifà allo schema già mostrato nella sezione di architettura del progetto, con qualche aggiunta.
Per assicurare una maggiore separazione dei ruoli tra i vari elementi del Model, ciascuna `Entity` delega la pubblicazione degli eventi a un componente esterno, chiamato `GameBus`, di cui esiste un'istanza per ogni `Publisher`. Le entità hanno a disposizione una serie di classi concrete di eventi (non riportate in figura) pubblicabili in base al tipo di azione di cui si vuole notificare l'avvenimento.
Notare che `Event` fornisce un metodo per ottenere i nomi delle interfacce implementate dalle classi Publisher: l'intento è quello di venire incontro alla

necessità di determinati Subscriber di filtrare gli eventi in arrivo nel caso in cui siano registrati a più Publisher contemporaneamente.
Nonostante le uniche entità a ricoprire il ruolo di Publisher siano, all'atto pratico, solo `Player` ed `Enemy`, il design di questo sistema permette facilmente di rendere Publisher una qualsiasi entità del Model, anche quelle non discendenti da `AbstractEntity`, nel caso in cui ciò sia richiesto per l'aggiunta di nuove funzionalità: basterà che deleghino per composizione a `GameBus` la pubblicazione degli eventi.

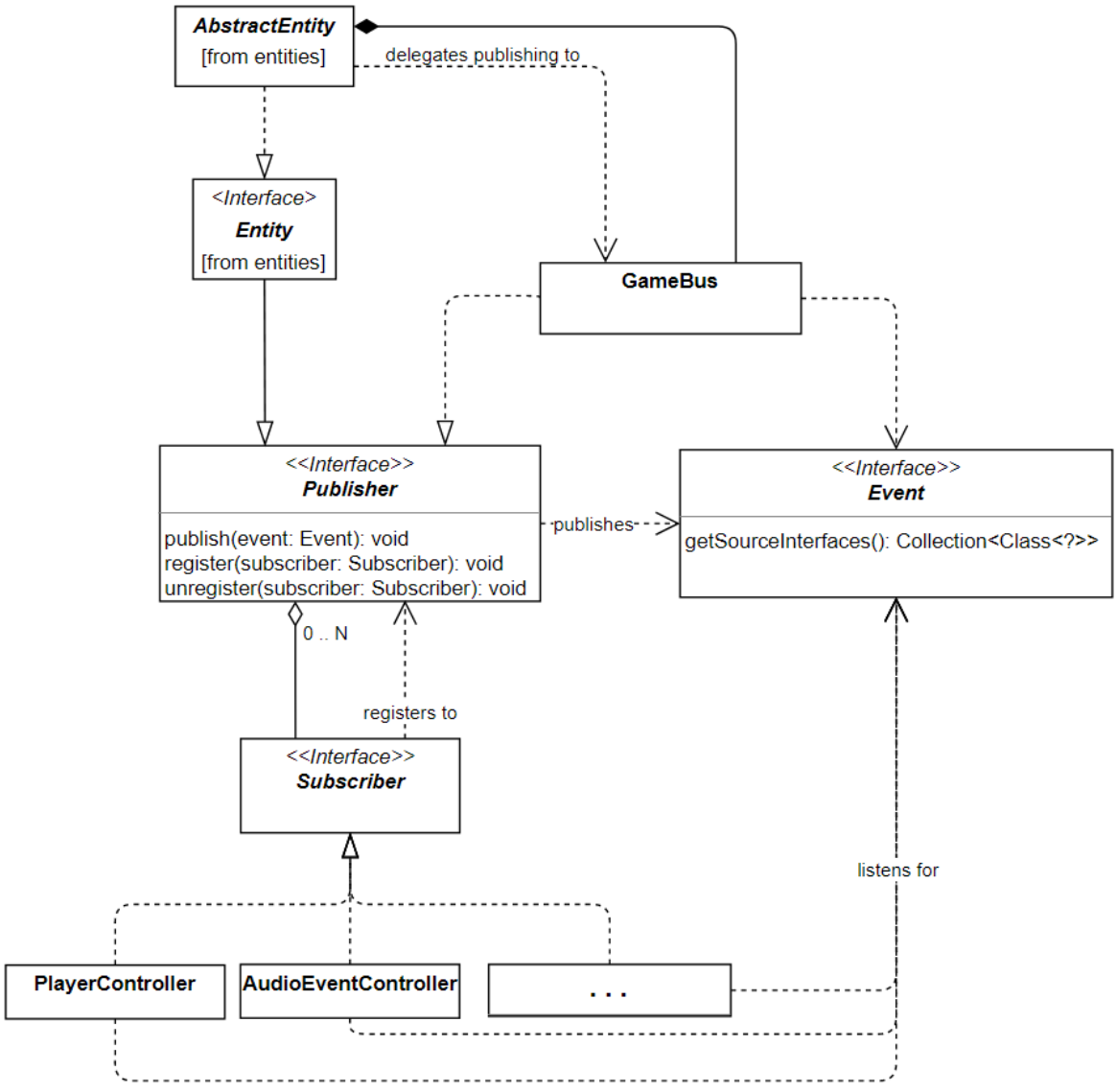


Figura 2.13: Applicazione del pattern *Publish-subscribe*.

Punteggio

Il calcolo del punteggio a fine partita è parte integrante del nostro gioco. Poiché la quantità di punti assegnata all'utente dipende da più fattori che tengono conto delle sue performance e del suo stile di gioco, mi è sembrato logico progettare il tutto nel seguente modo: il calcolo si compone di più algoritmi, identificati in fig. 2.14 dall'interfaccia `PartialScore`, che gestiscono autonomamente piccole parti del punteggio complessivo.
Terminata la partita, `Score` viene richiamato da un controller esterno per ottenere sia i punteggi parziali che quello totale.

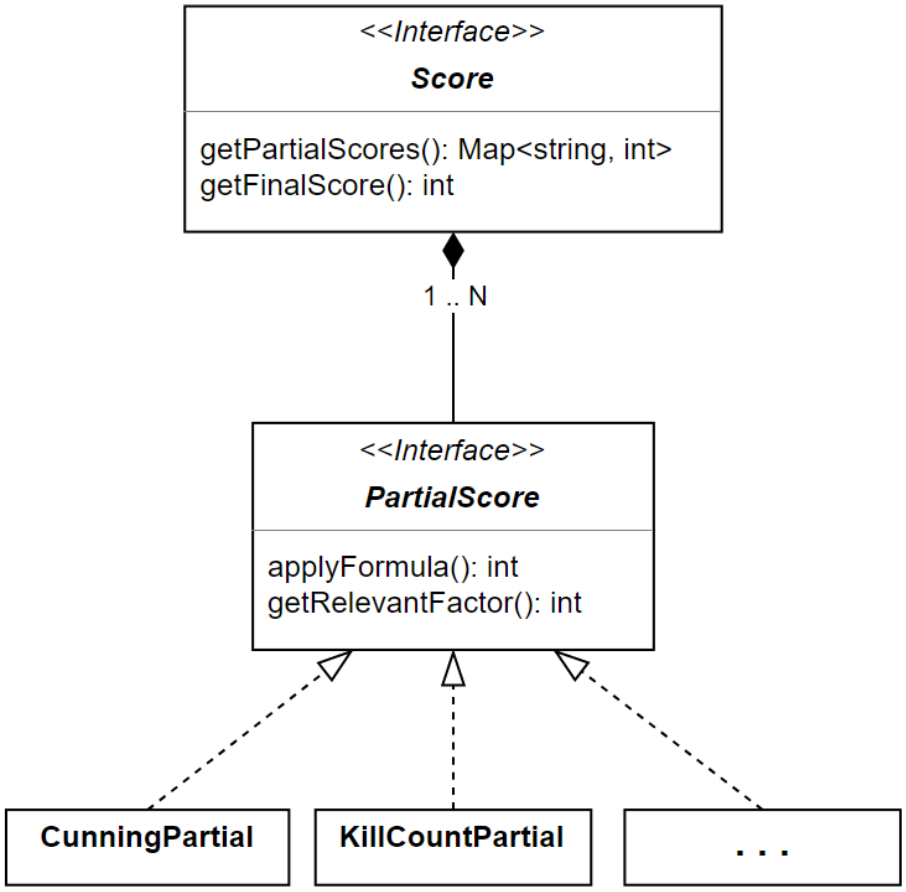


Figura 2.14: Gestione del punteggio.

Factory per il punteggio

La creazione dell'insieme di tutti i punteggi parziali viene demandata a una Factory esterna, rappresentata in fig. 2.15 dall'interfaccia `PartialScoreFactory`. Questo mi permette di disaccoppiare `Score` dalle singole implementazioni dei punteggi parziali e assicurare al tempo stesso che la creazione di questi da parte di controller esterni sia rapida e intuitiva.

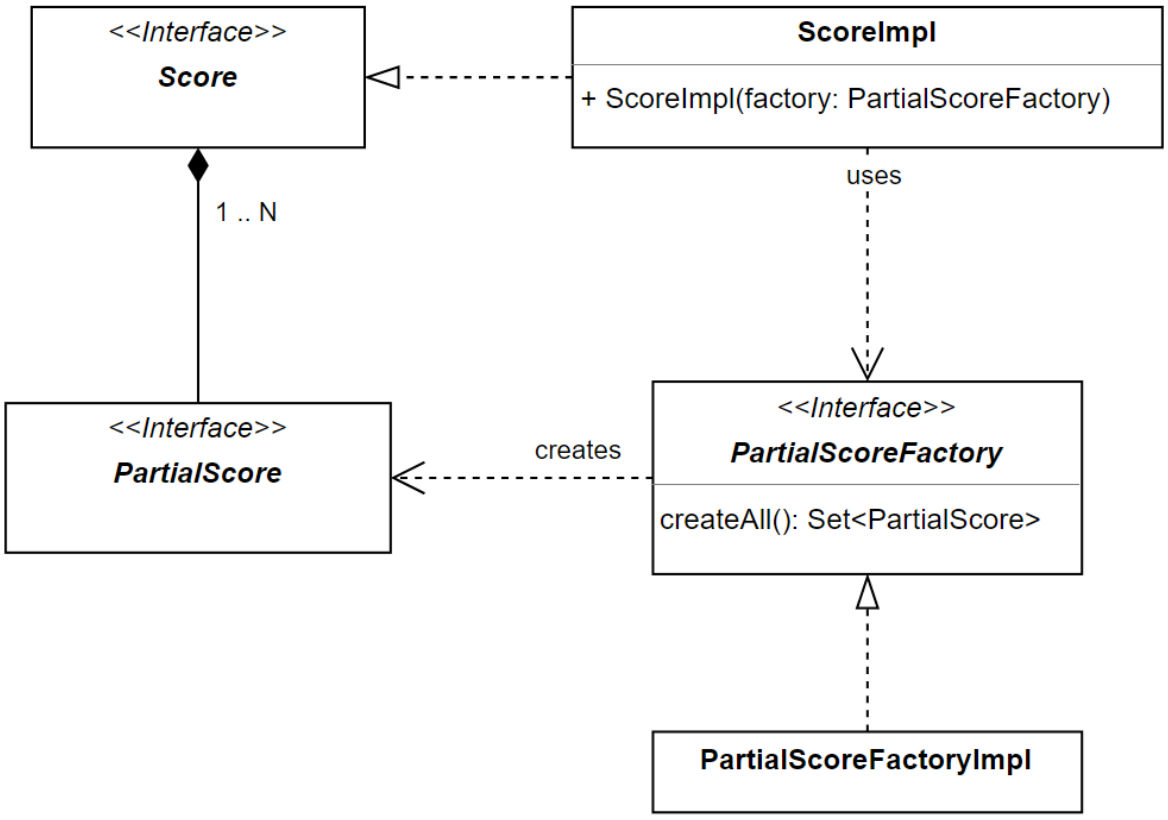


Figura 2.15: Applicazione del pattern *Factory* per il punteggio.

Telecamera

La gestione della telecamera è racchiusa nell'interfaccia `Camera` (fig. 2.16), la quale, seguendo i movimenti di un singolo `Sprite` specificato a inizio partita dal `WorldController`, aggiorna durante l'esecuzione del *game loop* la propria controparte di View, `CameraView`. Sebbene allo stato attuale la telecamera segua esclusivamente i movimenti del giocatore, l'interfaccia consente

di assegnarvi un qualsiasi `Sprite` in previsione di nuove funzionalità molto specifiche: ad esempio, in caso di sconfitta del giocatore, sarebbe possibile far puntare la telecamera al nemico responsabile della sua morte.

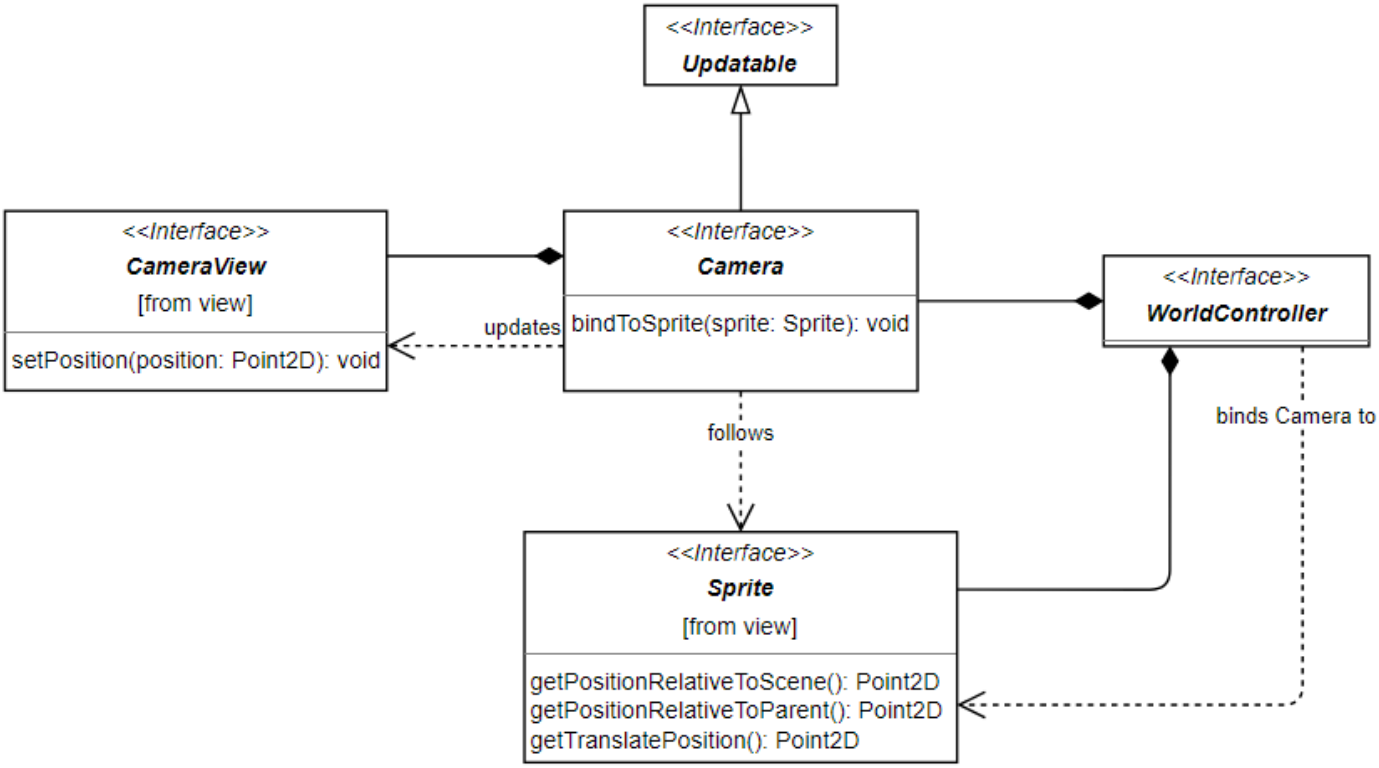


Figura 2.16: Schematizzazione della telecamera.

2.2.2 Federico Campanozzi

Data Access Layer

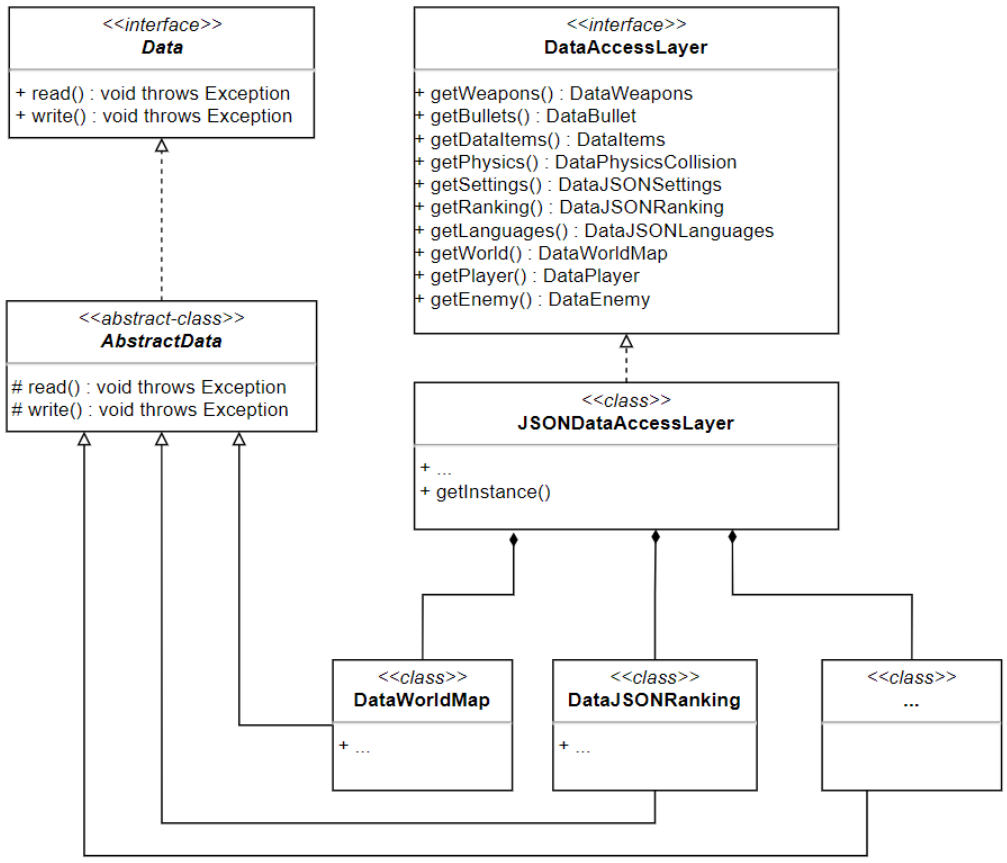


Figura 2.17: Data Access Layer UML

Il Data Access Layer e' stato il primo aspetto che ho realizzato del progetto, essendo una parte fondamentale per il funzionamento di esso. Ho pensato subito di applicare il pattern **Singleton** per avere un modo semplice e comodo per accedere ai dati. Una seconda motivazione e' legata alla condivisione, infatti applicando questo pattern abbiamo potuto condividere i dati a tutte le parti dell'applicazione senza inutili passaggi di argomenti o copiature cosi' da evitare errori di sincronizzazione e aggiornamento degli stessi. Per non violare **SRP**(Single Responsability Principle) la classe *JSONDataAccessLayer* si compone di molteplici sotto-classi lasciando a quest'ultime la responsabilita' di modellare una piccola parte del dominio applicativo di loro competenza,

come possiamo notare dallo schema (fig. 2.17). Per avere la massima flessibilita', ovvero la possibilita' di aggiungere con estrema facilita' altre classi, ho creato l'interfaccia *Data* e la classe astratta *AbstractData*, che ogni entita' del package `hotlinecesena.model.dataccesslayer.datastructure` estende, per avere un'implementazione di default. Ho usato questa tecnica per massimizzare il riutilizzo del codice, infatti grazie a *AbstractData* una classe che la estende puo' fare ovveride solo dei metodi che necessita. Un'altra possibilita' che questa gestione offre e' il passaggio ad una tecnologia piu' avanzata di memorizzazione strutturata, come ad esempio un database. Per fare questo cambiamento infatti basta estendere la classe *JSONDataAccessLayer* e sostituire quelle poche classi che leggono e scrivono su file JSON con delle classi che leggono e scrivono su tabelle di un database.

Procedural Generator

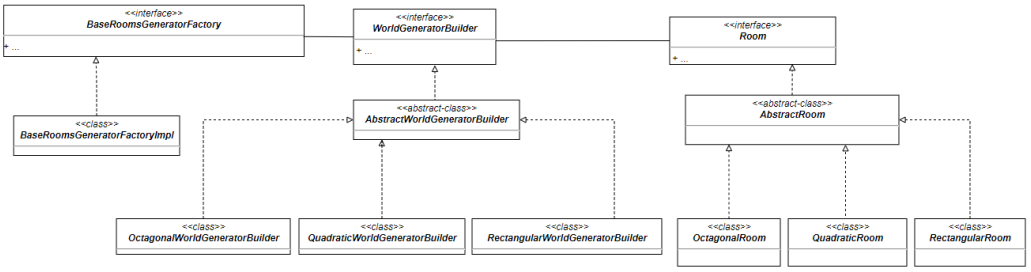


Figura 2.18: Schema generale del generatore

In fig. 2.18 e' mostrato come ho impostato il lavoro per la generazione casuale delle mappe. Per rendere il piu' customizzabile possibile e facilmente estendibile il generatore ho pensato di applicare un **template method** sul metodo *AbstractWorldGeneratorBuilder::generateRooms*. Per ridurre il codice ho creato una classe astratta *AbstractWorldGeneratorBuilder* che si interpone tra l'interfaccia *WorldGeneratorBuilder* e le singole implementazioni per racchiudere in essa tutte le parti comuni dei generatori e lasciare ai singoli il compito di implementare le parti che differiscono. Questa cosa si puo' verificare dalle classi *RectangularWorldGeneratorBuilder*, *OctagonalWorldGeneratorBuilder*, e *QuadraticWorldGeneratorBuilder*. Lasciando comunque aperta la possibilita' in futuro di creare altri generatori, ampliando quelli gia' esistenti oppure creandone di nuovi estendendo *AbstractWorldGeneratorBuilder*. In fig. 2.19 e' riassunto quanto appena detto :

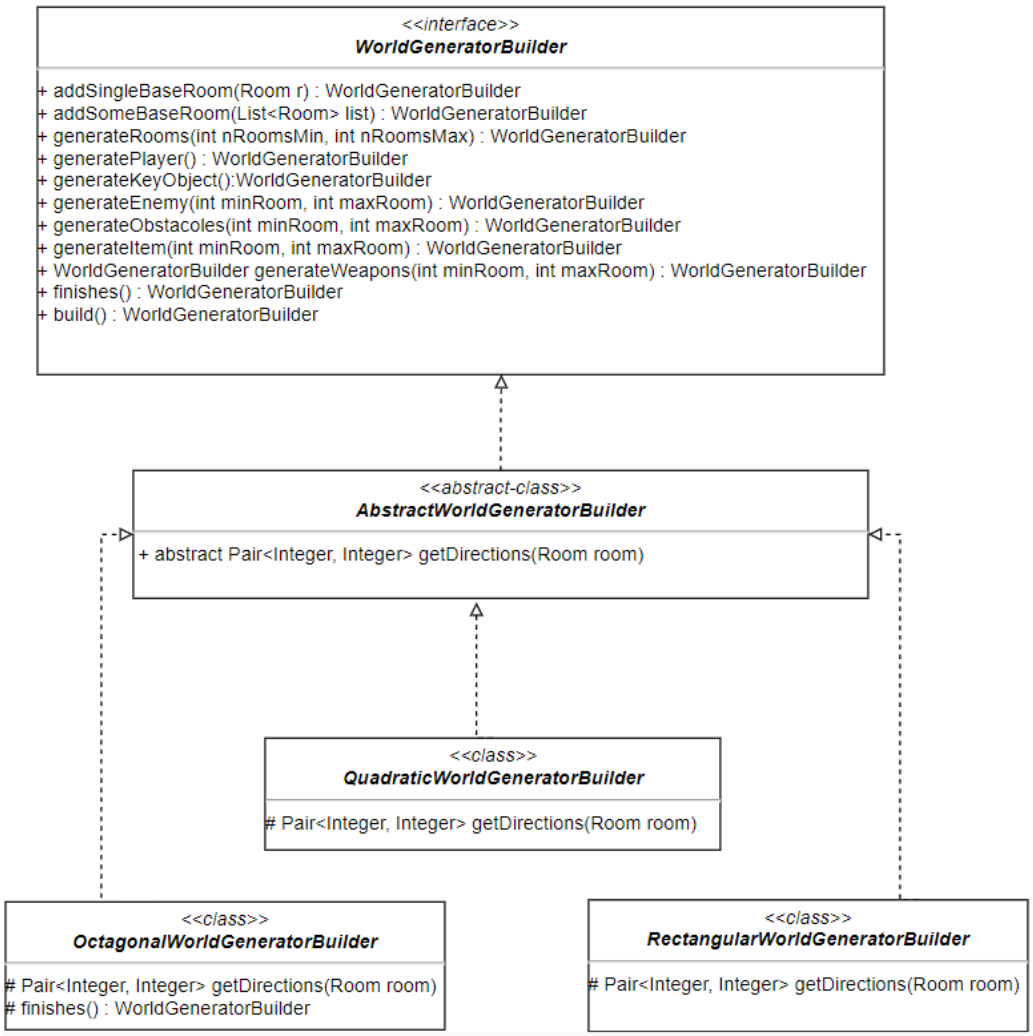


Figura 2.19: Schema dettagliato del *WorldGeneratorBuilder*

Tutto questo e' racchiuso all'interno di un pattern **builder** per gestire al meglio l'alto numero di parametri che il generatore necessita e per dare, in futuro, all'utente un modo per creare delle mappe con o senza determinati oggetti.

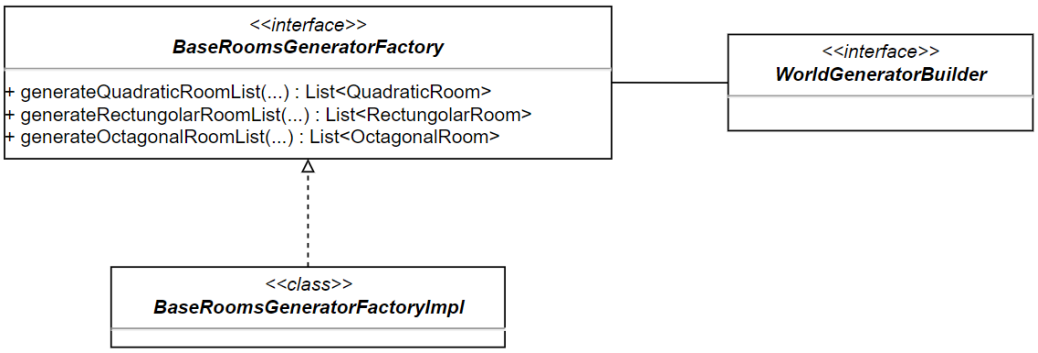


Figura 2.20: Schema dettagliato della *BaseRoomsGeneratorFactory*

Nell'interfaccia *WorldGeneratorBuilder* sono presenti due metodi *WorldGeneratorBuilder::addSingleBaseRoom* e *WorldGeneratorBuilder::addSomeBaseRoom* che aggiungono rispettivamente, una stanze e un insieme di stanze, da cui il generatore parte per unirle tra di loro e creare cosi' la mappa finale. Per facilitare la creazione di tali liste ho creato una factory, utilizzando il pattern **simple factory** sulla classe *BaseRoomsGeneratorFactory* come si puo' notare dallo schema UML in fig. 2.21. Questo consente in futuro di ampliare la factory generando, ad esempio, altre figure geometriche.

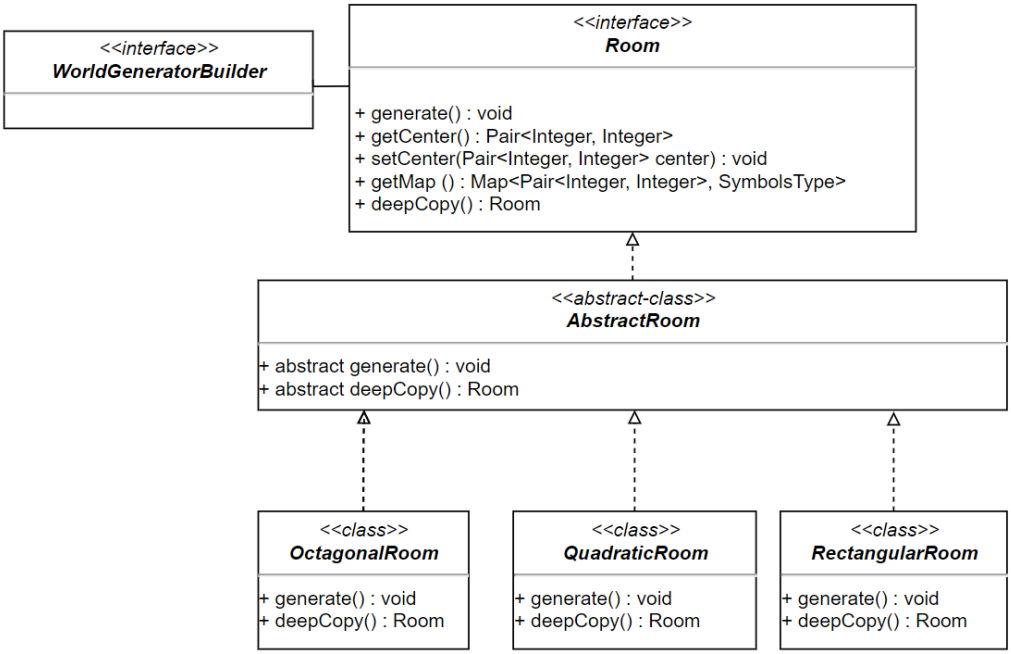


Figura 2.21: Schema dettagliato delle *Room*

Come si puo' notare dalla fig. 2.19 ho applicato la stessa logica del **template method** anche per la generazione delle singole stanze creando infatti l'interfaccia *Room*, la classe astratta *AbstractRoom* e le relative implementazioni *OctagonalRoom*, *QuadraticRoom* e *RectangularRoom*. In questo caso abbiamo pero' ben due template method, uno sul metodo *Room::generate* e l'altro su *Room::deepCopy*, il primo si occupa della generazione della mappa che descrive una singola stanza il secondo si occupa di creare una copia esatta della stanza istanziata.

Mission

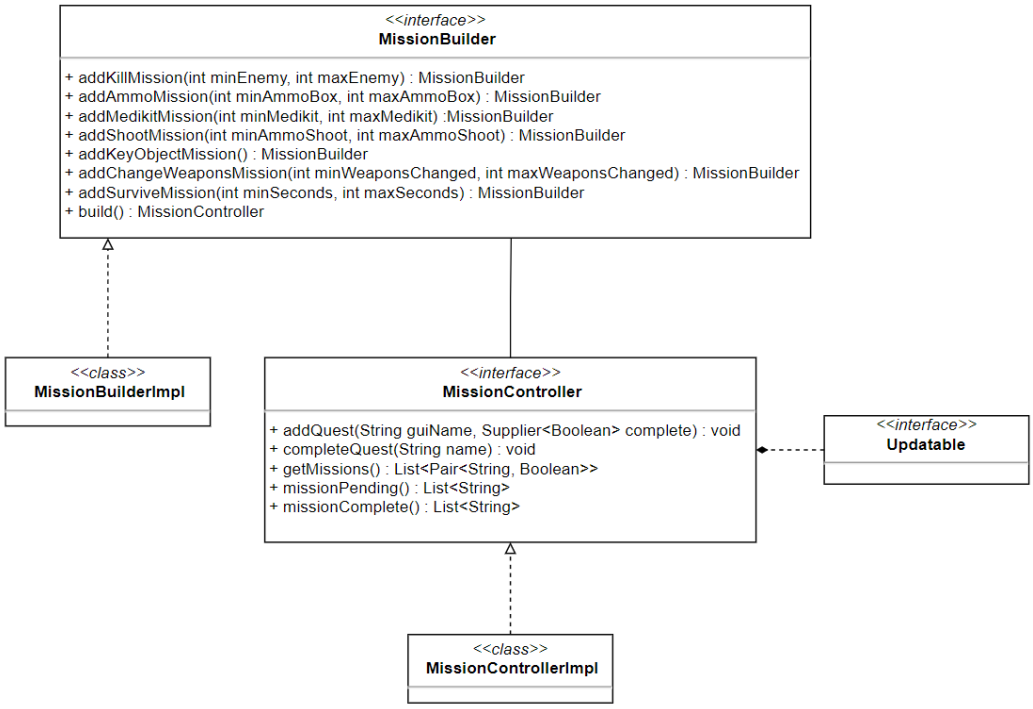
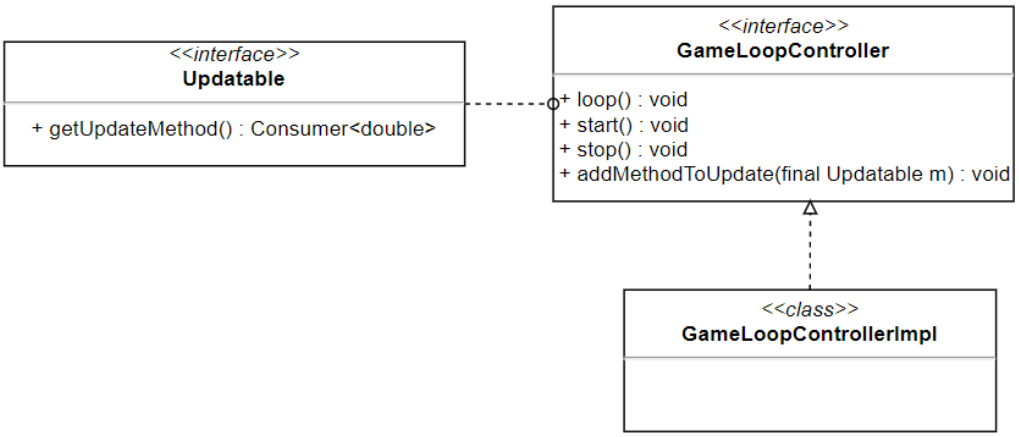


Figura 2.22: Mission Builder & Controller

Come si vede dalla (fig. 2.22) ho creato anche qui un pattern **builder** per la generazione delle missioni dovendo gestire un alto numero di di parametri. Come gia' detto in fase di analisi anche le missioni sono create in base al numero, e tipo, di oggetti presenti sulla mappa e alla composizione di quest'ultima.
Ho inoltre creato l'interfaccia *MissionController* che serve per gestire lo stato di avanzamento delle missioni e del loro completamento. Il metodo *MissionController::addQuest* e' quello che usa internamente il *MissionBuilder* per aggiungere le missioni ed e' volutamente public proprio per lasciare aperta la possibilita' in futuro di gestire missioni dinamiche, ad esempio in risposta a determinati eventi del player o degli enemy o di qualsiasi altro oggetto.

GameLoop



Il game loop e' una parte tanto semplice quando importante, infatti essa e' la classe incaricata alla gestione degli aggiornamenti delle altre classi. Per registrare una nuova classe da aggiornare, bisogna passare al metodo *GameLoopController::addMethodToUpdate* l'istanza di quest'ultima la quale deve implementare l'ionterfaccia *Updatable*. L'interfaccia *Updatable* e' un'interfaccia funzionale che sfrutta il metodo *Updatable::getUpdateMethod* per creare di volta in volta un metodo di aggiornamento diverso evitando il proliferarsi di classi e codice ridondante. Questo si e' rivelato molto utile e comodo e ci ha fatto risparmiare molto tempo grazie al fatto che queste si prestano molto bene all'implementazione tramite lamda expression.

2.2.3 Andrea Micheli

Il primo aspetto su cui mi sono concentrato durante la progettazione e la realizzazione di questo progetto è stato quello dell'implementazione dei *nemici*, cercando di rispettare la corretta gestione del framework di sviluppo assegnatoci.

Enemy

Come mostrato dalla ***Figura 2.23***, il nemico estende *Actor* che, tramite una gerarchia di interfacce lo specializza, modellandone le funzionalità di base; ciò gli permette di attraversare il mondo di gioco, ruotarsi ed interagire con l'ambiente che lo circonda. Inoltre, l'interfaccia *Enemy* implementa proprietà fondamentali al proprio funzionamento.
L'interfaccia AI (Intelligenza Artificiale) è indispensabile al nemico, agendo come un cervello che, simulando input visivi e sonori gli permette di muoversi ed orientarsi nella mappa di gioco. Mentre, tramite l'utilizzo di un'enumerazione, è possibile generare diversi tipi di nemici, caratterizzati da un proprio stile di movimento.

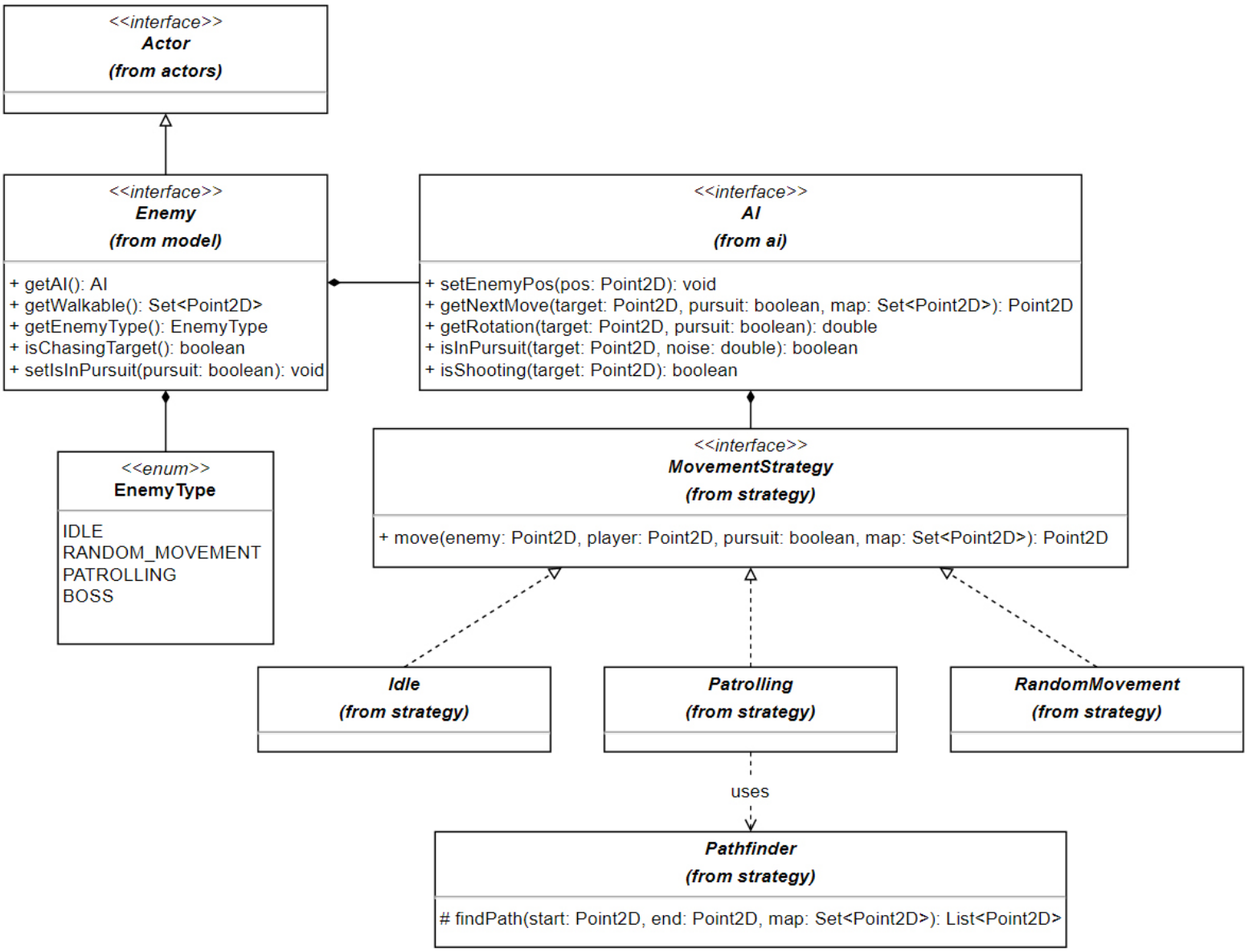


Figura 2.23: Enemy UML

Intelligenza Artificiale

L'implementazione di algoritmi che permettono ai nemici di muoversi autonomamente e reagire in modo intelligente ai comandi dell'utente è l'aspetto principale che mi ha guidato durante la progettazione.
Per implementare al meglio i diversi tipi di movimenti ho deciso di utilizzare un pattern **strategy**, permettendomi così di gestire dinamicamente l'algo-

ritmo di movimento da utilizzare per ogni tipo di nemico. Come visto in ***Figura 2.23***. L'utilizzo di questo pattern mi ha consentito di ottenere un' implementazione non caratterizzata da molteplici classi che differenziano tra loro per solo un metodo. Ciò mi ha permesso di prestare più tempo all'implementazione di algoritmi che avrebbero reso più astuto il nemico, come ad esempio A* (utilizzato in *Pathfinder*): algoritmo che permette ai nemici di raggiungere il giocatore sfruttando il percorso calcolato essere il più breve, sempre che questo esista.
Utilizzando l'interfaccia di Intelligenza Artificiale sono stato quindi in grado di separare facilmente l'implementazione della logica dei movimenti del nemico, dalla vera e propria attuazione di tali.

Enemy Controller

Per quanto riguarda la vera e propria attuazione dei movimenti, dopo diverse iterazioni, sono riuscito a trovare un modo per poter gestire separatamente la logica dalla sua effettiva applicazione. Ogni nemico viene direttamente controllato dalla propria istanza di *AI*, la quale indica ad ogni aggiornamento i comandi da eseguire. E' il nemico stesso, nel momento in cui viene istanziato, a conoscere tutte le posizioni degli ostacoli e dei muri contenuti all'interno della mappa di gioco, in modo da poter comunicare all'*AI* quali siano i movimenti attuabili.
Ad ogni aggiornamento, tramite la conoscenza della posizione del giocatore e la propria rotazione, il nemico è in grado di determinare, definendo un valore di campo visivo e capacità visiva, se il giocatore sia visibile.
Per portare il nemico ad attaccare invece è necessaria la conoscenza delle posizioni dei muri che lo circondano e la posizione del giocatore. Per fare ciò è necessario un algoritmo che verifichi che la linea di tiro sia libera. Tale controllo è reso possibile simulando un ipotetico raggio diretto tra le due entità; se il raggio dovesse spezzarsi per via di un muro, significherebbe che la visuale è oscurata.

Pattern Factory

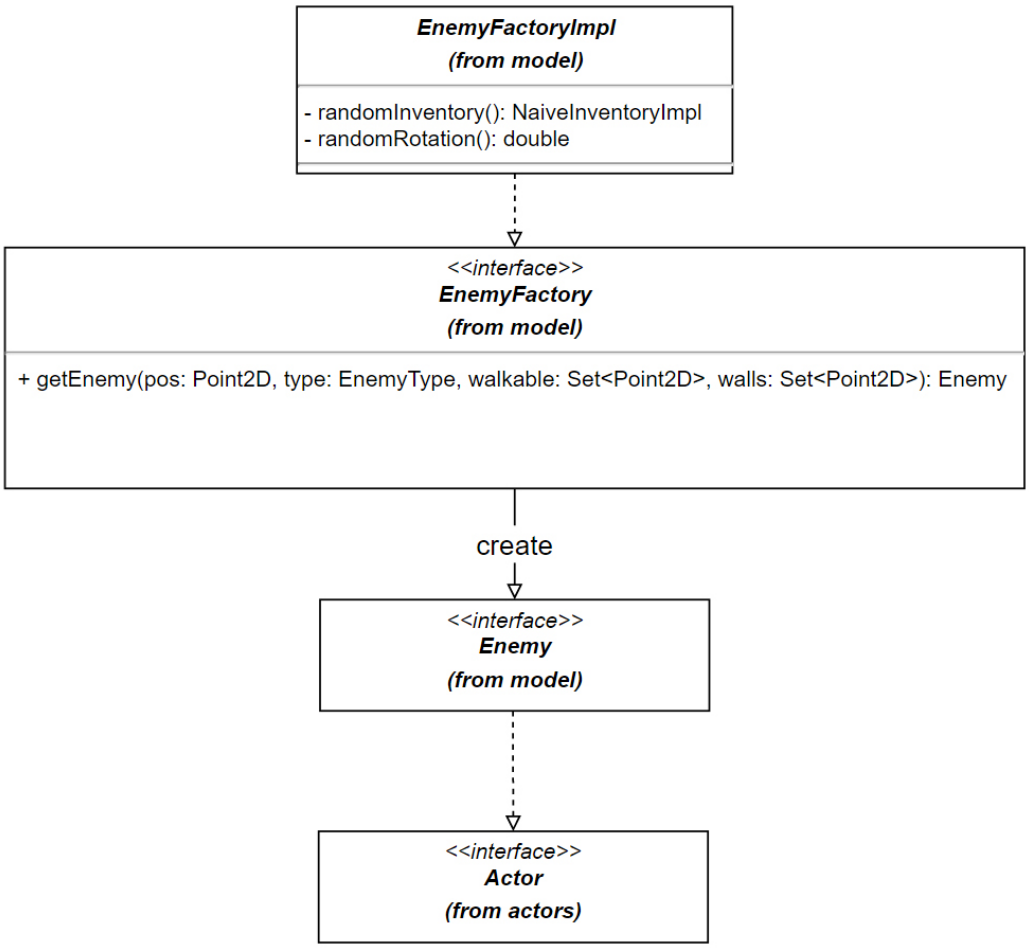


Figura 2.24: UML della factory di *Enemy*

Per la creazione di nemici come mostrato nella ***Figura 2.24*** ed in modo simile per i loro rispettivi controller di movimento ***Figura 2.25***, ho scelto di utilizzare il pattern **factory**, che mi permette di creare diversi tipi di nemici o controller. Tramite un'implementazione di tipo *simple factory*, sono stato in grado di esporre, attraverso una semplice interfaccia, il metodo per la generazione di tali. Ciò permette di facilitare la creazione di questi oggetti da parte di classi esterne.

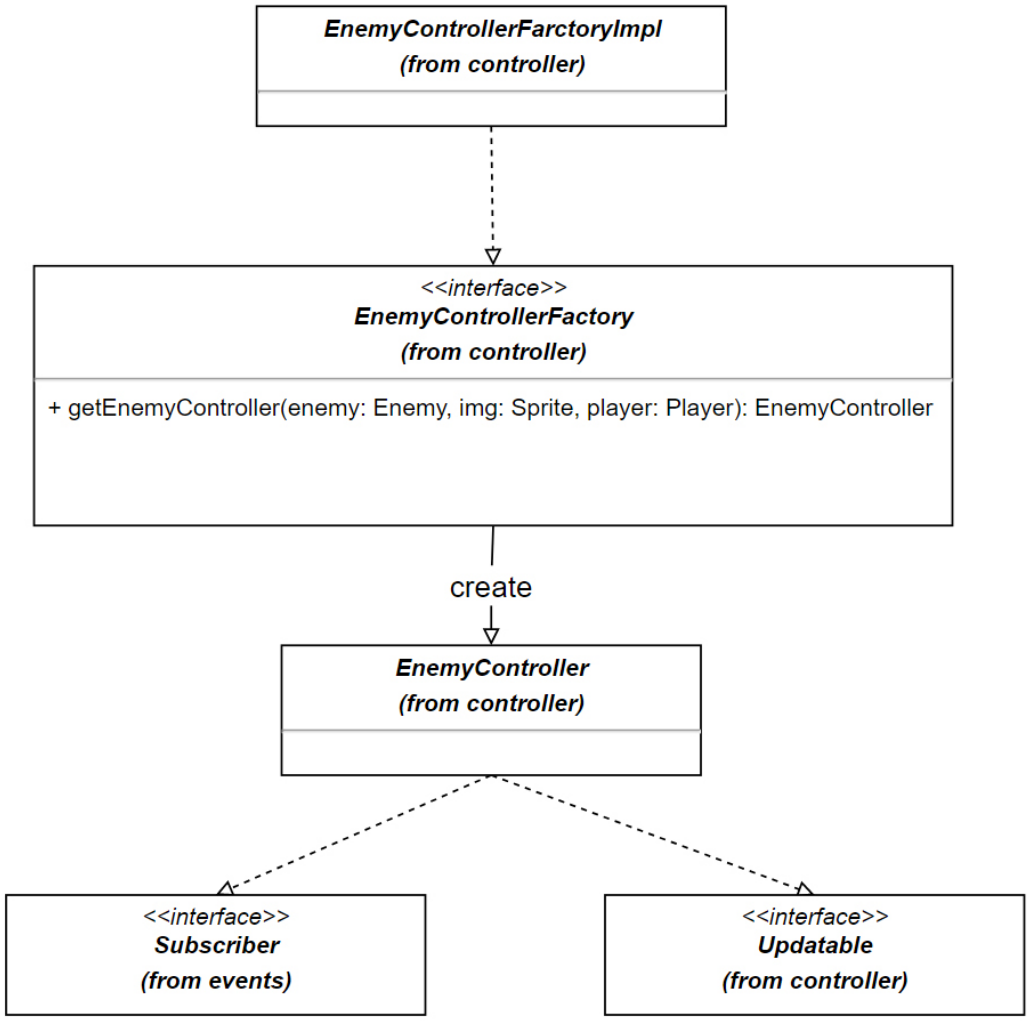


Figura 2.25: UML della factory di *EnemyController*

Pattern Proxy

L'altra mansione che mi è stata assegnata, si basa sulla creazione e caricamento di risorse grafiche/sonore. Per poter accedere al file system, sia per le immagini che per i file audio, ho deciso di utilizzare il pattern **proxy**. Questo pattern permette di fornire un'interfaccia che nasconde e protegge la vera classe che compie le operazioni: come ad esempio la gestione del caricamento in memoria di un oggetto, un file o altre risorse.
Tramite *ProxyImage* è possibile richiedere il "reale" caricamento dei file, che avviene grazie ad una classe innestata al suo interno, permettendo un accesso "lazy o on demand" (su richiesta) delle risorse.

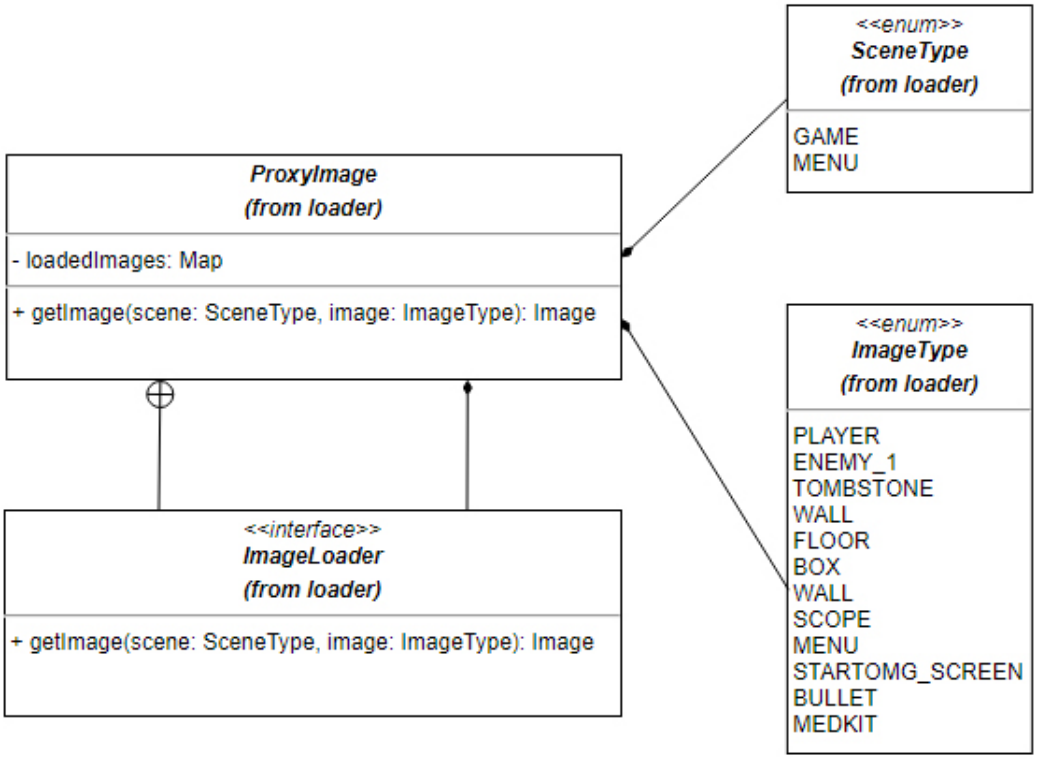


Figura 2.26: UML proxy image

Grazie ad un'interfaccia in grado di contenere informazioni con una associazione *chiave* a *valore* (la ***Map***), come mostrato in ***Figura 2.26***, il *ProxyImage* è in grado di mantenere per ogni istanza una copia dell'immagine, evitando di sprecare risorse andando a ricaricare oggetti già in memoria. Utilizzando la *Map* è quindi possibile, ogni volta che viene fatta una richiesta di istanziamento di un nuovo oggetto, evitare di caricare nuovamente la risorsa in caso questa sia già presente all'interno della mappa.
Anche per quanto riguarda le risorse audio come mostrato in ***Figura 2.27*** ho utilizzato un approccio simile, ma per questo tipo di file ho preferito proteggere e quindi "non esporre" gli oggetti che il *ProxyAudio* crea.
Per fare questo ho implementato un controller che permette la riproduzione di clip audio, file mp3 e di aggiornare le impostazioni che l'utente potrebbe cambiare. Facendo ciò il controller è in grado di impostare il volume di riproduzione e la presenza di audio all'interno del gioco (in caso questo venga disabilito dall'utente), occupandosi personalmente della logica necessaria alla riproduzione delle risorse. In questo modo si possono riprodurre "on demand" (su richiesta) file audio senza preoccuparsi delle loro impostazioni.

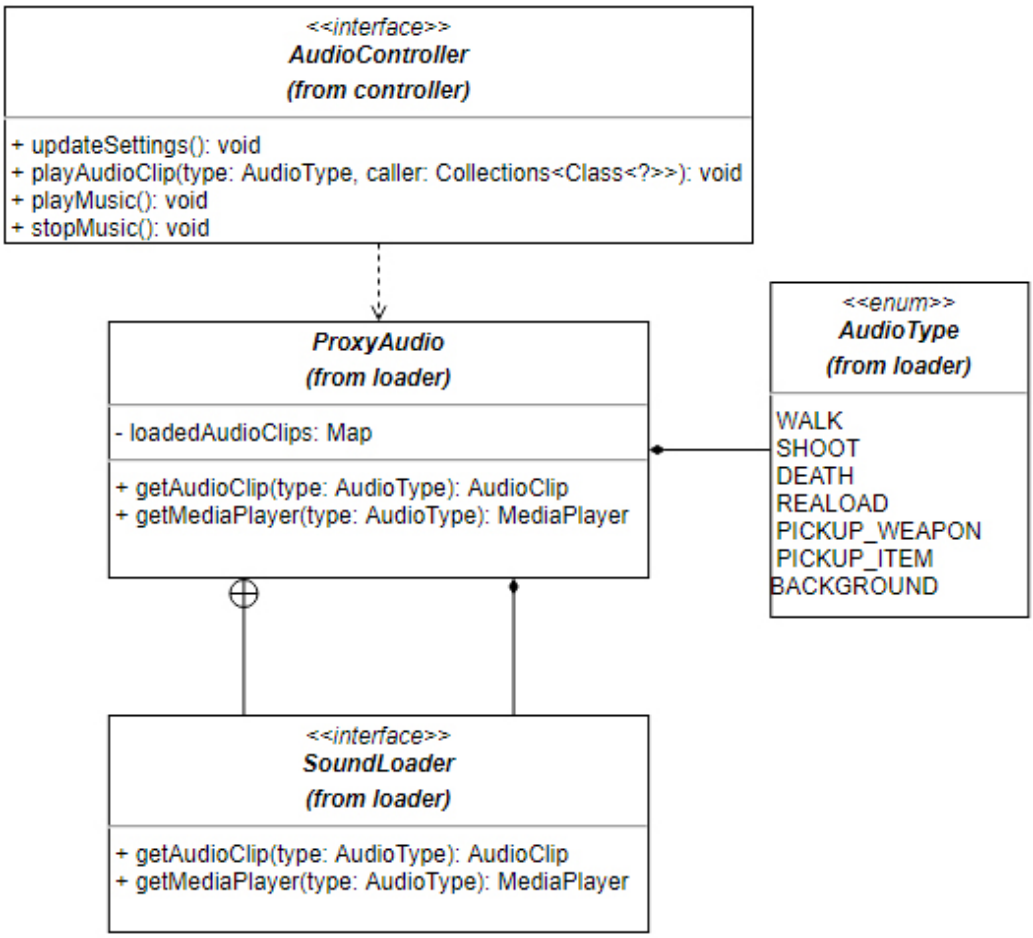


Figura 2.27: UML proxy audio

Audio Event Controller

Per poter associare un suono ad ogni azione che il giocatore o i nemici compiono, si è utilizzato l'EventBus di Guava, che tramite un pattern *Publisher/Subscriber* permette una comunicazione asincrona tra oggetti. Così facendo ogni volta che un *attore* compie una determinata azione, ne pubblica il relativo evento, che viene intercettato dall'*AudioEventController*. Tramite l'implementazione di *playAudioClip* in *AudioController*, come mostrato in ***figura 2.28***, mi è permesso conoscere l'oggetto che ha prodotto l'evento. Tale informazione è necessaria per poter aggiustare dinamicamente il volume di riproduzione di ogni *AudioClip*. Questa decisione è stata presa per portare più enfasi alle azioni che il giocatore compie e per ridurre l'inquinamento acustico prodotto dai nemici.

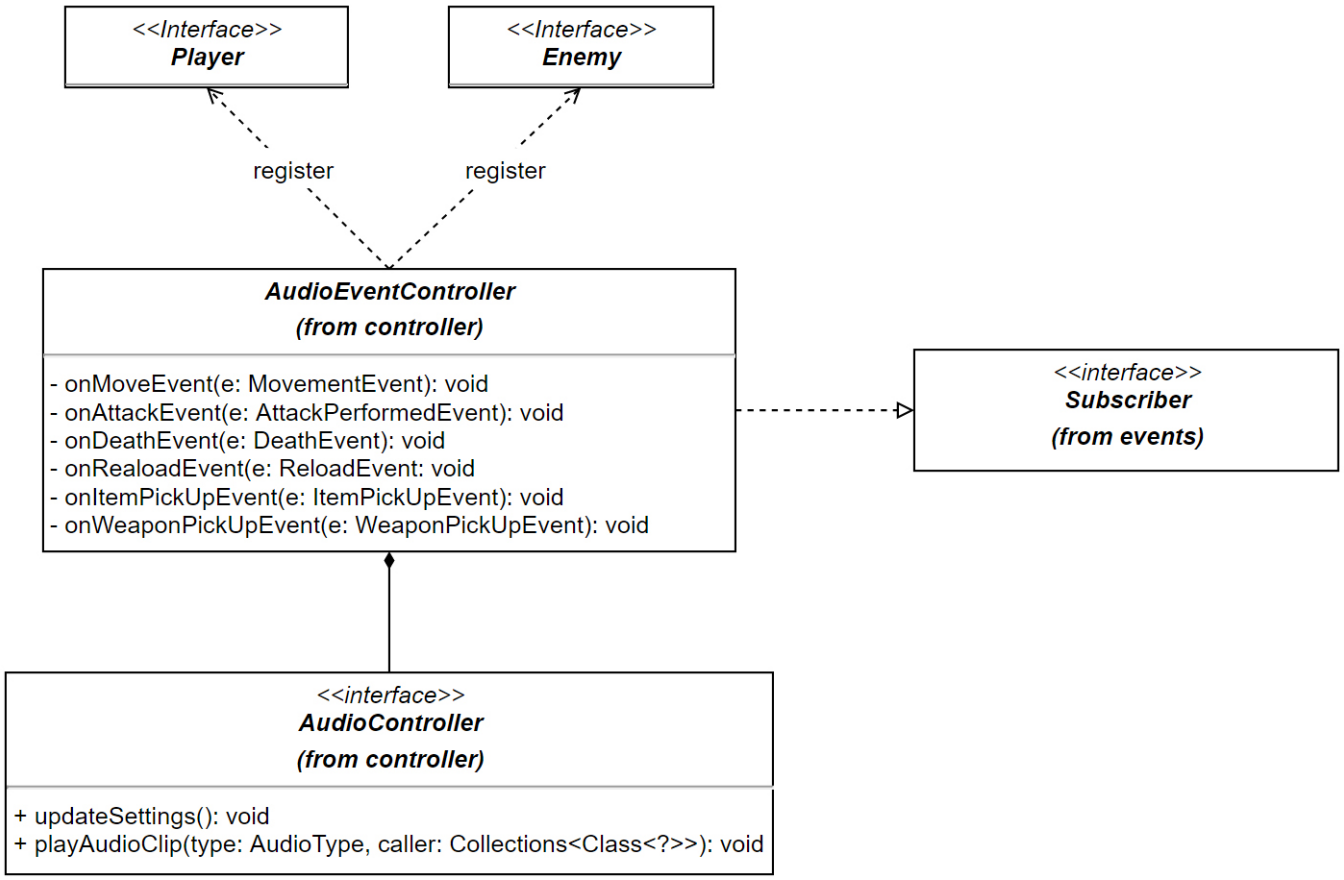


Figura 2.28: UML audio event controller

2.2.4 Andrea Zammarchi

Il mio ruolo all'interno del gruppo prevede l'intera realizzazione della GUI, oltre all'implementazione dei vari tipi di items, armi e proiettili.

GUI

Per la realizzazione delle varie interfacce grafiche è stata sfruttata la libreria *JavaFX*.
Innanzitutto ho realizzato un oggetto di supporto, denominato `SceneSwapper`, con lo scopo centralizzare (e quindi facilitare) lo scambio di scene. La potenza di questo componente sta nella sua flessibilità, ovvero il poter essere impiegabile anche in progetti non appartenenti al mondo videoludico.
Il suo impiego all'interno del progetto mi ha permesso di risparmiare tempo e linee di codice ogniqualvolta fosse richiesto uno "swap".
In particolare lo `SceneSwapper`, per creare una scena, necessita di tre argomenti:

- Un file fxml (view design);
- Il controller di tale file;
- Lo stage nel quale verrà caricata la scena.

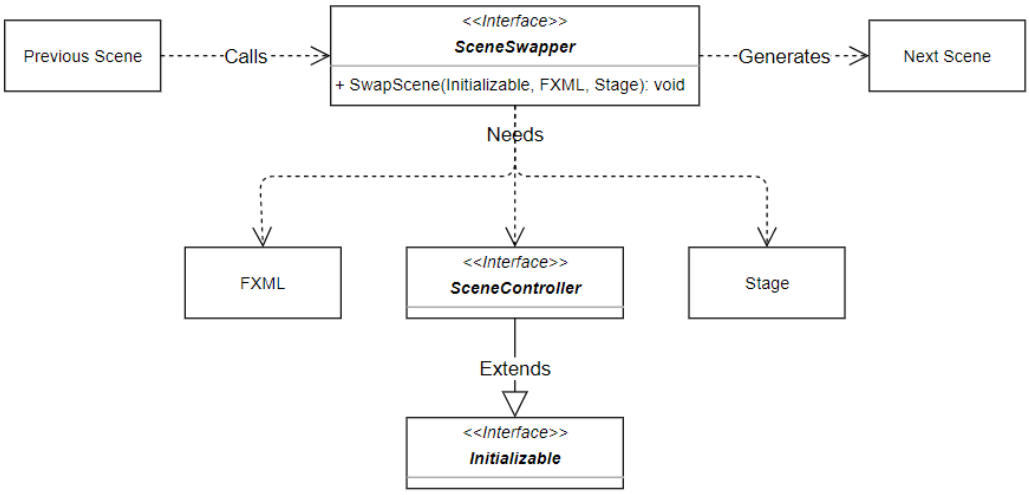


Figura 2.29: UML funzionamento SceneSwapper

In questo modo, la gestione della GUI si adatta perfettamente all'implementazione del pattern principale dell'applicazione, ovvero MVC.

WorldView

La `WorldView` è l'unica scena che non viene generata dallo `SceneBuilder`, bensì viene creata da zero dal comando del `WorldController`. Ho fatto questa scelta perché è un tipo di scena complessa, difficile da implementare tramite file FXML.

Ho deciso che alla base di tale scena ci fosse uno *StackPane* in quanto è un particolare tipo di *Pane* che favorisce la sovrapposizione di più *Containers*. Questo pannello, nel nostro caso, è formato da due "layer": il BottomLayer e il TopLayer.
Come mostrato in **Figura 2.30**, il BottomLayer è costituito da un *GridPane* contenente il vero e proprio mondo di gioco, mentre il TopLayer è costituito da un *BorderPane* contenente l'HUD.

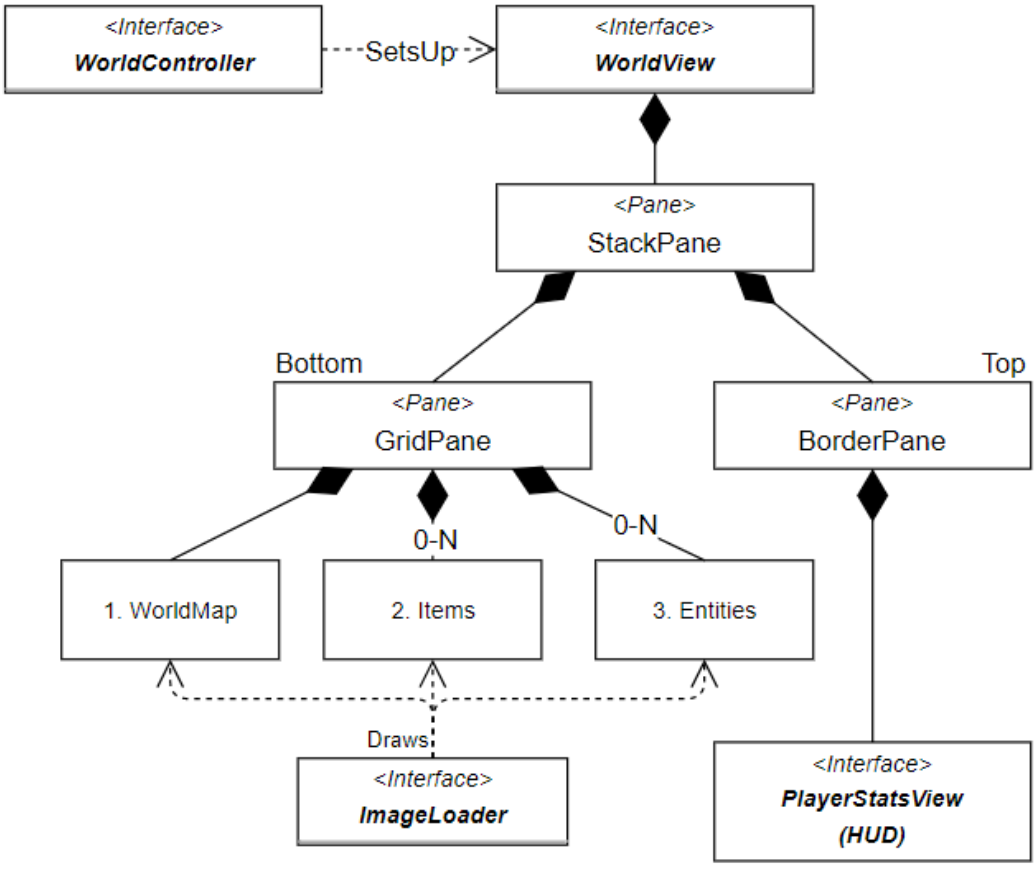


Figura 2.30: UML struttura WorldView

Questo tipo di struttura permette all'HUD di sovrapporsi al mondo di gioco senza intaccarne minimamente il suo funzionamento. In questo modo ho potuto lavorare sull'overlay senza preoccuparmi di manomettere ad esempio la camera, implementata da Bryan Corradino.

Items

Esistono tre categorie di `Items`:

- `Weapons`: verranno trattate nella prossima sezione;
- `ItemsType`: oggetti sparsi per la mappa, con i quali il `Player` può interagire;
- `CollectibleType`: oggetti collezionabili, non presenti sulla mappa, ma contenuti nell'inventario del `Player`;

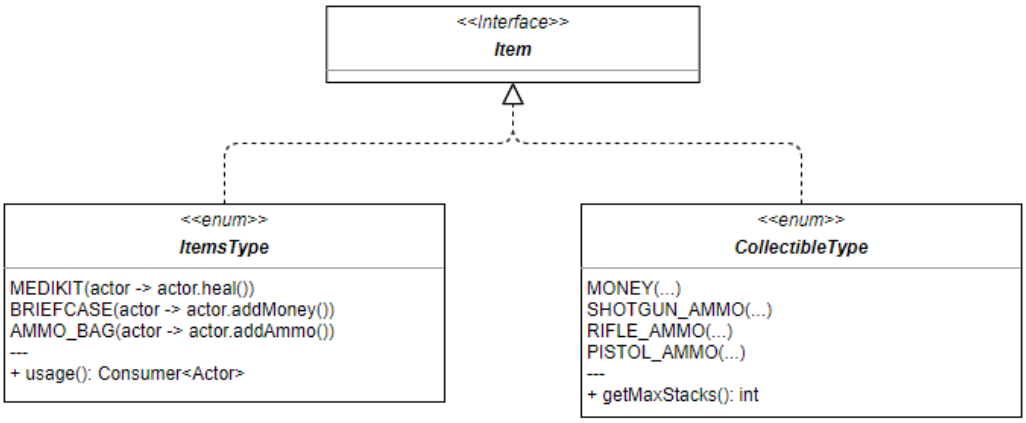


Figura 2.31: Items

Gli `Items` collezionabili vengono accumulati nel momento in cui il giocatore si posiziona sopra un `ItemType` e preme il comando per interagire con esso. Ad esempio, se il giocatore raccoglie la valigetta di denaro (`BRIEFCASE` → `ItemsType`), viene richiamato il suo metodo *usage()* che aggiungerà all'inventario del `Player` una certa somma di denaro (`MONEY` → `CollectibleType`).

Si è scelta questa specifica struttura perché è facilmente estensibile. Infatti, per quel che è stata la mia esperienza, una volta creato lo scheletro degli `Items`, è stata immediata la creazione sia di oggetti semplici (come il

MEDIKIT che non interagisce con alcun CollectibleType), sia di ogetti più avanzati (come la AMMOBAG che interagisce con più tipi di munizioni, ovvero oggetti collezionabili). Ciò la rende ideale per futuri scenari dove si avrà la necessità di aggiungere ulteriori tipi di Items.

Armi e proiettili

Come precedentemente accennato, nella mappa si possono trovare sparse anche varie armi. Se ci si posiziona sopra una di queste, si avrà la possibilità di scambiare l'arma attualmente equipaggiata con quella sottostante.
Ho deciso che le Weapons fossero un'estensione di Item, visto che anch'esse hanno una funzione (*usage()*), ovvero sparare. Inoltre, per la creazione della demo, è stata fissata a 1 la quantità massima di armi possedute, ma in realtà, essendo Items, possono essere collezionate. In questo modo, nell'ipotesi di una futura implementazione di una nuova modalità di gameplay, il Player potrà tenere nell'inventario diverse armi e selezionare quella desiderata in base alla situazione di gioco.
Come già detto, esistono diverse tipologie di armi, elencate nella enum WeaponsType. Ogni tipologia di arma, ovviamente, ha valori diversi per ogni attributo, come il danno, la cadenza di fuoco, il rumore, le dimensioni del caricatore, ecc... Per incentivare ulteriormente lo scambio di armi a seconda della situazione, si è pensato anche di creare diversi tipi di strategie di attacco. In particolare una Weapon (come PISTOL e RIFLE) può sparare un singolo proiettile diritto oppure può sparare diversi proiettili contemporaneamente con una certa ampiezza (come SHOTGUN).

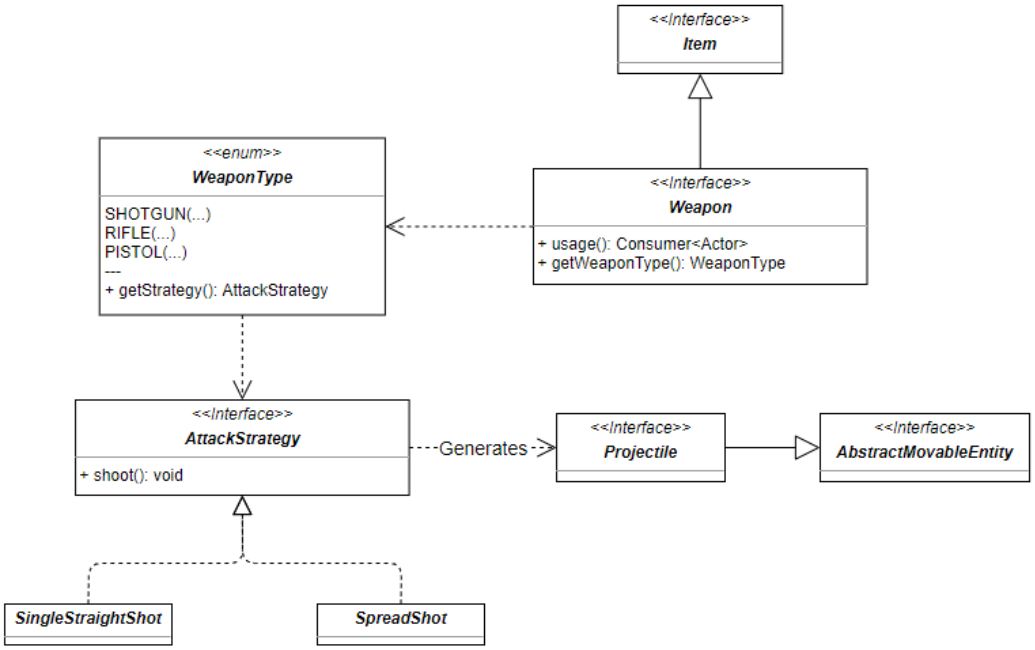


Figura 2.32: UML model di armi e proiettili

Come per gli `Items`, il design mostrato in **Figura 2.32** rende immediata l'aggiunta ad esempio di nuovi tipi di armi e/o di strategie d'attacco.
L'obiettivo fissatomi inizialmente era quello di rendere il più variegato possibile il gameplay. La creazione di uno scheletro facilmente scalabile sia per gli items che per le armi, mi ha permesso di raggiungere appieno questo traguardo. In questo modo l'utente può divertirsi nell'elaborare strategie diverse a seconda della situazione di gioco in cui si trova.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il progetto è stato configurato in modo tale da supportare l'esecuzione di test JUnit tramite Gradle.
Qui di seguito sono elencati i nomi delle classi di test realizzate per verificare il corretto funzionamento delle varie componenti dell'applicazione.

- `ActorModelTest`: verifica il funzionamento dei metodi comuni a tutte le implementazioni dell'interfaccia `Actor`;
- `CameraTest`: verifica che la telecamera segua correttamente i movimenti di uno `Sprite`;
- `EnemyModelTest` verifica il corretto funzionamento dei sistemi di movimento, visione e attacco dei nemici.
- `InputListenerTest`: verifica la corretta ricezione degli input relativi a mouse e tastiera [1];
- `InterpreterPlayerTest`: verifica che gli input per il giocatore ricevuti dalla View vengano interpretati in modo corretto dal Controller;
- `LoaderTest` verifica il corretto caricamento di risorse grafiche e sonore all'interno dell'applicazione.
- `MapGeneratorTest`: Questo test e' composto di due parti. La prima, genera delle immagini (.png), e le memorizza nella cartella $.HotLineCesena \longrightarrow$

[1] Può capitare che la finestra creata dalla classe di test non riesca a passare subito in primo piano, comportando il fallimento di alcuni test di cattura degli input. Se ciò dovesse accadere, provare a terminare tutte le istanze della JVM e a riavviare Eclipse.

File $\longrightarrow$ *GeneratedMap*, questo serve per vedere visivamente se il generatore produce delle mappe giocabili e divertenti. La seconda verifica il corretto utilizzo dei metodi nel *WorldGeneratorBuilder*, cioe' verifica possibili criticita' dei parametri e se i metodi vengono chiamati in un ordine accettabile (non posso chiamare *WorldGeneratorBuilder::generateEnemy* senza prima aver chiamato *WorldGeneratorBuilder::generateRooms*).

- `UtilitiesTest` verifica il corretto utilizzo delle funzioni di della classe *Utilities*.

- `ViewsTest` verifica il corretto layout delle scene.

3.2 Metodologia di lavoro

Il gruppo, valutando attentamente il tempo richiesto per l'implementazione di ciascuna funzionalità e operando una leggera ridistribuzione delle mansioni rispetto a quanto riportato nella proposta di progetto, si è assicurato di assegnare a ciascun membro un carico di lavoro equo.
Per meglio organizzare lo sviluppo dell'applicazione, si è fatto largo utilizzo del DVCS **Git**: in particolare, si è proceduto aggiungendo gradualmente nuove funzionalità nel branch `develop`; poi, dopo fasi di test intermedie atte a verificare il corretto funzionamento dell'applicazione nel suo complesso, le modifiche sono state integrate nel branch `main`.

3.2.1 Bryan Corradino

- Sviluppo di classi astratte per tutta la gerarchia di `Entity` fino a `Player`; implementazione di `Player`; creazione dell'interfaccia `Status` e dell'enumerazione `ActorStatus` per la gestione degli stati degli `Actor`.
  Ho consultato Andrea Micheli e Andrea Zammarchi per l'inserimento e l'implementazione di metodi delle `Entity` comuni alle interfacce `Enemy` e `Projectile`.

- Creazione dell'interfaccia `Inventory` e della sua implementazione `NaiveInventoryImpl` per una gestione semplificata dell'inventario degli `Actor`.

- Implementazione di `Score` e dei punteggi parziali tramite l'interfaccia `PartialScore` per il calcolo del punteggio complessivo.

Ho collaborato con Andrea Zammarchi per l'aggiunta dei punteggi alla classifica nel suo `RankingController`.

- Implementazione di `Sprite` per la rappresentazione grafica delle `Entity`.
- Implementazione di `Camera` e `CameraView` per il movimento della telecamera.
- Creazione e gestione del `PlayerController` e del suo componente chiave `InputInterpreter`.
- Sviluppo e implementazione dell'interfaccia `InputListener` per agevolare la ricezione di input dalla View.
- Creazione dell'interfaccia `Command` e sue implementazioni (sia dirette, sia sotto forma di espressioni lambda), utilizzate dall'`InputInterpreter` per la conversione di input in comandi.
- Realizzazione dell'interfaccia `Event` e sue implementazioni, di `Publisher` e la sua implementazione `GameBus` (facente uso dell'`EventBus` di Google Guava), e infine dell'interfaccia marker `Subscriber`.
  Mi sono interfacciato con tutti i membri del gruppo per la gestione degli eventi all'interno del Controller e della View: in particolare, ho collaborato con Andrea Micheli per la creazione di eventi specifici riguardanti la riproduzione di effetti sonori e i cambiamenti di immagini.
- Creazione dell'interfaccia `Updatable` per una più ordinata organizzazione dei controller all'interno del `GameLoopController` di Federico Campanozzi.

3.2.2 Federico Campanozzi

- Realizzazione delle classi per l'accesso ai dati. Mi sono interfacciato molto con i miei colleghi per capire quali dati memorizzare e per semplificare il loro lavoro fornendo delle elaborazioni ad alto livello.
- Realizzazione del *WorldGeneratorBuilder* e delle classi ad esso collegate. Ho creato e gestito in maniera autonoma questa porzione di codice.
- Creazione del *MissionBuilder* e del *MissionController*. Mi sono interfacciato con Andrea Zammarchi per integrarlo correttamente nel *WorldController*

- Creazione del *GameLoopController*. Ho creato il GameLoopController poi mi sono interfacciato con Bryan Corradino per quanto riguarda l'interfaccia *Updatable* per una miglior gestione e pulizia del codice.

- Creazione del *Laucher* e del *GameLoader* che fanno da entry point all'applicazione. Mi sono accordato con Andrea Zammarchi per quanto riguarda le classi da utilizzare per il caricamento dei file FXML.

- Creazione parziale della classe *MathUtils* e totale della classe *ConverterUtils* e dei suoi test nella classe *TestUtils*.

3.2.3 Andrea Micheli

- Sviluppo ed implementazione di **Enemy**, interfacciandomi con le gerarchia di interfacce che Bryan Corradino aveva implementato per specializzare *MovableEntity*; gestione della creazione di diversi tipi di nemici tramite una enumerazione **EnemyType**.

- Sviluppo ed implementazione di **AI**, interfaccia in grado di impartire comandi al rispettivo nemico, rispettando il pattern architetturale assegnatoci.

- Utilizzo di pattern *Strategy*, tramite **MovementStrategy**, per la gestione di diverse implementazione di movimento, quali: **Idle**, **RandomMovement**, **Patrolling**.

- Implementazione di A* (algoritmo di ricerca) che permette ai nemici di tipo *Patrolling* di inseguire il giocatore.

- Sviluppo ed implementazione di **EnemyPhysicsUtils**, utilizzato per verificare che le decisioni di movimento prese da **AI** siano attuabili e per controllare che la linea di tiro di ogni nemico sia libera. Controlli eseguiti utilizzando le collezioni di muri ed ostacoli generati da Federico Campanozzi alla creazione della mappa di gioco.

- Sviluppo ed implementazione di **EnemyController** che, tramite l'interfaccia *Updatable*, è in grado ad ogni frame di attuare i comandi impartiti da **AI** al rispettivo nemico e di aggiornare la relativa immagine.

- Sviluppo di **EnemyFactory** e **EnemyControllerFactory**, interfacce che aiutano la creazione di diversi tipi di nemici e relativi controller.

- Sviluppo ed implementazione della gestione di caricamento di risorse grafiche e sonore tramite **SoundLoader** e **ImageLoader**, innestate all'interno di due classi, **ProxyAudio** e **ProxyImage**, che utilizzato il pattern *Proxy* per proteggere i "real loaders" (coloro che compiono il reale caricamento della risorsa).
  Ho collaborato con Bryan Corradino per la creazione e gestione di nuove immagini e risorse audio relative alle *Entity*. Mi sono interfacciato con Andrea Zammarchi per la creazione di immagini da aggiungere alla *View*.

- Sviluppo ed implementazione di **AudioController**, utilizzato per facilitare l'accesso alle risorse audio e per proteggere gli oggetti creati dal *ProxyAudio*.

- Sviluppo ed implementazione di **AudioEventController**, tramite il quale è stato possibile associare ad ogni evento prodotto da ogni *Entity* il relativo suono.
  Mi sono interfacciato con Bryan Corradino per la creazione di eventi ad-hoc a cui associare il relativo suono.

3.2.4 Andrea Zammarchi

- Progettazione e realizzazione dell'intera interfaccia grafica: menu, mondo di gioco e HUD. Ho collaborato con diversi colleghi a seconda della scena in oggetto.

- Sviluppo e implementazione dello `SceneSwapper` per facilitare lo scambio tra scene.

- Realizzazione della schermata introduttiva del progetto, dove ho cooperato con Federico Campanozzi per la creazione del `Launcher`.

- Realizzazione dell'`OptionsMenu` dove l'utente può cambiare varie impostazioni audio/grafiche e salvare le modifiche, sia prima della partita che durante. Ho cooperato sia con Andrea Micheli per i settaggi audio, in quanto l'`AudioController` del gioco è stato implementato da lui, che con Federico Campanozzi per il salvataggio delle preferenze dell'utente.

- Sviluppo e implementazione della `WorldView`, contenente il mondo di gioco, durante la quale ho collaborato con Andrea Micheli per poter sfruttare il suo metodo di caricamento di immagini tramite il `ProxyImage`.

- Realizzazione del `WorldController`, gestore della `WorldView`. In questa fase ho collaborato con tutti quanti i colleghi in quanto in questa classe vengono istanziate tutte le implementazioni di `Updatable`, che vengono poi aggiunte al `GameLoopController`.
- Sviluppo e implementazione dei `Projectiles`, degli `Items` e quindi anche delle `Weapons` e dei `Projectiles`. Ho cooperato molto con Bryan Corradino in questa fase, in quanto l'`Inventory` del giocatore è stato implementato da lui. Inoltre i `Projectiles` sono una estensione della classe astratta `AbstractMovableEntity`, creata sempre da Bryan Corradino.
- Realizzazione della schermata di leaderboard post-partita, dove il giocatore può registrare i suoi risultati e verificare la sua posizione in classifica. In questa parte ho collaborato con Bryan Corradino in quanto lui ha implementato il sistema di assegnamento dei punti (`Score`). Inoltre ho collaborato anche con Federico Campanozzi per gestire, tramite il DAL implementato da lui, il salvataggio degli `Score` aggiornati.

3.3 Note di sviluppo

3.3.1 Bryan Corradino

- Uso di Optional
- Espressioni lambda e Stream
- Uso della Reflection
- Progettazione con generici bounded
- Librerie esterne: JavaFX, Google Guava, Apache Commons
- Librerie per il testing automatizzato: JUnit, TestFX per JavaFX, Hamcrest per la realizzazione di test più espressivi
- Utilizzo del build system Gradle

Le implementazioni dei metodi `MathUtils::lerp` e `MathUtils::blend` sono state prese da `https://gamedev.stackexchange.com/a/152466`.
L'implementazione del metodo `MathUtils::isCollision` è una versione riadattata e semplificata del metodo `BoundingBox::intersects` di JavaFX.
L'implementazione della classe `InputListenerFX` è fortemente ispirata a

quella della classe `InputHandlerImpl` del progetto `https://bitbucket.org/danysk/oop19-de-bonis-gianluca-de-crescenzo-andrea-pagliazzi-lorenzo/` ed entrambe risolvono gli stessi problemi: in JavaFX, mediante la sola aggiunta di `EventHandler` ad un nodo, la pressione prolungata di un tasto qualsiasi viene rilevata con un ritardo di circa 1 secondo; ad un `KeyEvent` o `MouseEvent` è associato un singolo `KeyCode` o `MouseButton`, rendendo impossibile rilevare la pressione di combinazioni di tasti (necessario per normalizzare il vettore direzione nel caso in cui il giocatore si muova in diagonale).

3.3.2 Federico Campanozzi

- Espressioni lambda
- Stream
- Programmazione Funzionale
- Uso di Optional
- Librerie esterne: jackson per la serializzazione e deserializzazione di file json
- Librerie per il testing automatizzato: JUnit
- Utilizzo del build system Gradle

3.3.3 Andrea Micheli

- Lambda expressions
- Stream
- Optional
- JavaFX
- Google Guava
- Gradle
- JUnit

3.3.4 Andrea Zammarchi

- **JavaFX**: motore grafico dell'applicazione. Sfruttati ovviamente file FXML per il design, oltre ai CSS per lo styling. Ho puntato sulla flessibilità grafica, in particolare ogni menu è "resizable" e adatta il suo layout in base alle dimensioni della finestra.
- **Lambda expressions**: ho sfruttato parecchio le lambda expressions, in particolare nella `WorldView` per analizzare il contenuto della mappa generata dal DAL. Le ho utilizzate inoltre nelle `Weapons` nell'azione dello sparo, che può essere richiamata da qualsiasi `Actor`.
- **Optional**: seppur poco, l'ho utilizzato nell'`OptionsMenu` in quanto può essere aperto sia quando è presente la finestra del mondo di gioco, che non.
- **Librerie per il testing**: JUnit, TestFX per JavaFX.
- **Gradle**: è stata sicuramente una scelta azzeccata l'utilizzo di Gradle nel nostro progetto. Ciò ci ha permesso di importare JavaFX e effettuare debug con molta rapidità.

L'idea dello strumento `SceneSwapper` si è ispirata alla classe `SceneSwapper` del progetto Zombieversity.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Bryan Corradino

Nel complesso, mi ritengo adeguatamente soddisfatto di ciò che sono riuscito a creare col poco tempo a mia disposizione. Tuttavia, sento che alcune parti del progetto si sarebbero potute gestire in modi migliori: nello specifico, la progettazione delle entità per mezzo di una catena di estensioni risulta essere particolarmente limitante, specialmente nel caso in cui si vogliano modificare metodi e funzionalità già esistenti e utilizzati da molteplici sottoclassi. In tal senso, il rischio di introdurre bug o di alterare involontariamente il comportamento delle sottoclassi è altissimo. Mi sarebbe piaciuto, di fatto, esplorare la possibilità di integrare il pattern architetturale **Entity-Component-System** all'interno di MVC, o quantomeno nella sola parte di Model: sarà un esperimento per un altro progetto.
Per il resto, credo di aver riconosciuto correttamente le situazioni in cui applicare i pattern giusti e di aver prodotto del codice relativamente pulito, sebbene la mia sensibilità sia negativamente influenzata dalla mia scarsa esperienza da programmatore.
C'è stata buona sintonia all'interno del gruppo e la collaborazione con i vari membri è risultata piacevole, nonostante non siano mancati problemi organizzativi o di sovrapposizione di ruoli. Inizio solo ora a comprendere il perché dell'esistenza della metodologia di sviluppo *agile*. In ogni caso, se in futuro si decidesse di espandere questo progetto, sarei ben felice di continuare a lavorarci.

4.1.2 Federico Campanozzi

Mi e' piaciuta molto questa parte di esame di programmazione ad oggetti perche' mi ha fatto capire molte cose importanti di come si lavora in un team di sviluppo e mi ha fatto crescere molto come programmatore object-oriented in generale. Sono molto soddisfatto del risultato complessivo e di come abbiamo cooperato, non conoscendoci abbiamo avuto qualche difficolta' aggiuntiva ma tutti quanti abbiamo dato il massimo e ci siamo aiutati a vicenda per rispettare gli alti canoni del corso. Purtroppo per questioni lavorative non ero sempre presente pero' penso che questo abbia portato un valore aggiunto al progetto. Per questo motivo ringrazio i miei colleghi e la loro disponibilita'.
Avrei voluto dedicare molto piu' tempo all'implementazione di logiche piu' avanzate sul generatore. Ad esempio mi sarebbe piaciuto creare una logica differente per ogni tipologia di oggetto oppure cercare metodi piu' intelligenti, basati su machine learning e deep-learning, per una distribuzione piu' organizzata delle stanze.

4.1.3 Andrea Micheli

Lavorare a questo progetto mi ha aiutato a consolidare quanto appreso durante il corso, e di sicuro è stata un'ottima esperienza formativa. Mi ritengo soddisfatto del progetto finale e delle sezioni da me svolte. Penso di essere stato un buon collega durante la progettazione e realizzazione di questo progetto, perchè sono sempre stato aperto ad un altro punto di vista e mi sono sempre reso disponibile. E' la prima volta in cui mi è stato assegnato un framework di sviluppo ben definito: inizialmente ho fatto molta fatica per cercare di suddividere correttamente tutti gli aspetti di progettazione assegnatomi. Poi, una volta presa dimestichezza con il pattern architetturale, mi sono appassionato, tanto che non escludo, in futuro, di aggiungere al progetto nuove funzionalità. Trovo che attività di questo tipo possano aiutare in futuro nell'approccio a situazioni che si potranno presentare nel mondo del lavoro, sia per le relazioni con i propri colleghi, che per affrontare nuovi strumenti.

4.1.4 Andrea Zammarchi

Posso confermare, senza esitazione, che questo progetto mi ha dato tanto. Mi ha permesso di crescere come programmatore. Sono entrato in contatto (ovviamente intendo contatto remoto, in tempi di Covid è necessario specificarlo...) con persone fantastiche, sicuramente più esperte di me in fatto

di programmazione, che non hanno esitato ad aiutarmi quando lo necessitavo. Penso di essere riuscito a dare un contributo comunque importante alla realizzazione dell'applicazione. Fatta questa premessa, sono pienamente consapevole che alcuni aspetti di progettazione della mia parte possono essere migliorati, come ad esempio la maggiore applicazione di pattern.
Le possibili direzioni future di questo progetto sono numerose e già predisposte, come l'aggiunta di vari livelli di difficoltà, la creazione di items che conferiscano malus (trappole) anziché bonus, una gestione avanzata dell'inventario del giocatore (dove potrà trasportare più armi), ecc...
Ringrazio i miei colleghi per aver, come me, dedicato tante forze a questo progetto a cui tengo molto.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Bryan Corradino

Realizzare questo progetto ha comportato, almeno nel mio caso, una mole davvero enorme di studio individuale. Molto del tempo che avrei potuto dedicare all'approfondimento di altri corsi è stato invece impiegato per ampliare le mie conoscenze di programmazione ad oggetti, per destreggiarmi tra molti dei bizzarri tecnicismi di Java e per imparare le basi del build system **Gradle**, al quale a lezione sono stati dedicati solo pochi accenni. Nonostante queste criticità, posso dire di aver acquisito una visione più chiara e completa su tutti (o quasi) i concetti affrontati durante il corso.

4.2.2 Federico Campanozzi

Ho trovato questo progetto abbastanza impegnativo nonostante avessi gia' lavorato come junior-coder in una azienda esperta nel settore videoludico, in particolare di racing-games stile Formula1. La parte piu' complessa da me sviluppata e' sicuramente il generatore, infatti mi ci sono volute piu' ore del previsto pero', pero' non mi sono pesate e nonostante questo ho completato tutte le parte che mi erano state assegnate in fase di Analisi.

4.2.3 Andrea Micheli

Personalmente ho trovato il progetto molto più impegnativo di quanto pensassi: siccome si tratta di un lavoro di gruppo, con scadenze da rispettare, ho "lasciato perdere" corsi del secondo semestre e ho sempre lavorato con

la preoccupazione di rallentare i colleghi (per fortuna non è successo), dal momento che ho seri problemi di salute. Quindi, pur essendo i progetti di gruppo molto formativi in vista di un futuro mondo del lavoro, avrei apprezzato la possibilità di realizzare progetti singolarmente o con team più piccoli.

4.2.4 Andrea Zammarchi

Ho riscontrato alcune difficoltà all'inizio del progetto: ero un programmatore con poca esperienza, non conoscevo nessuno dei miei colleghi e l'aver contratto il Covid per due settimane non ha sicuramente aiutato. Sono però fiero di essere riuscito a superare questi scogli, soprattutto grazie al mio team.

Appendice A

Guida utente

Al lancio dell'applicazione, superata la schermata iniziale, verrà presentato il *Menu principale* (come mostrato in ***Figura A.1***).

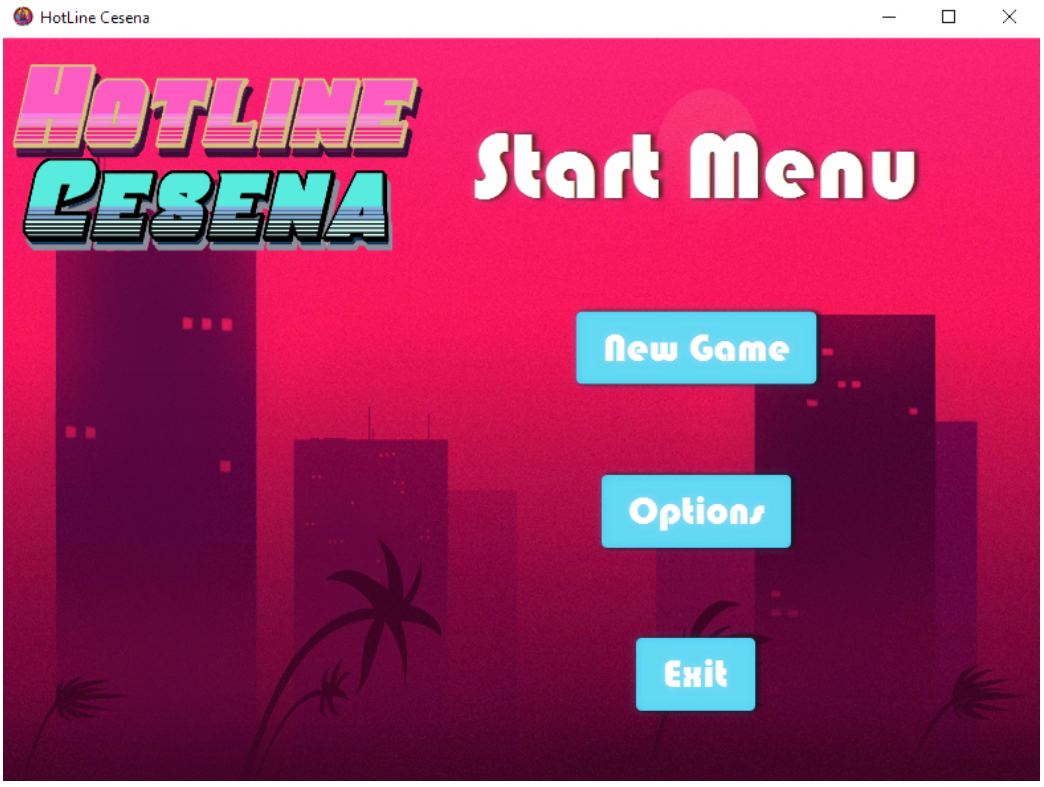


Figura A.1: Main menu

- **New Game** : genera una nuova partita
- **Options** : mostra il menu delle opzioni (come mostrato in ***Figura A.2***)
- **Exit** : termina l'applicazione

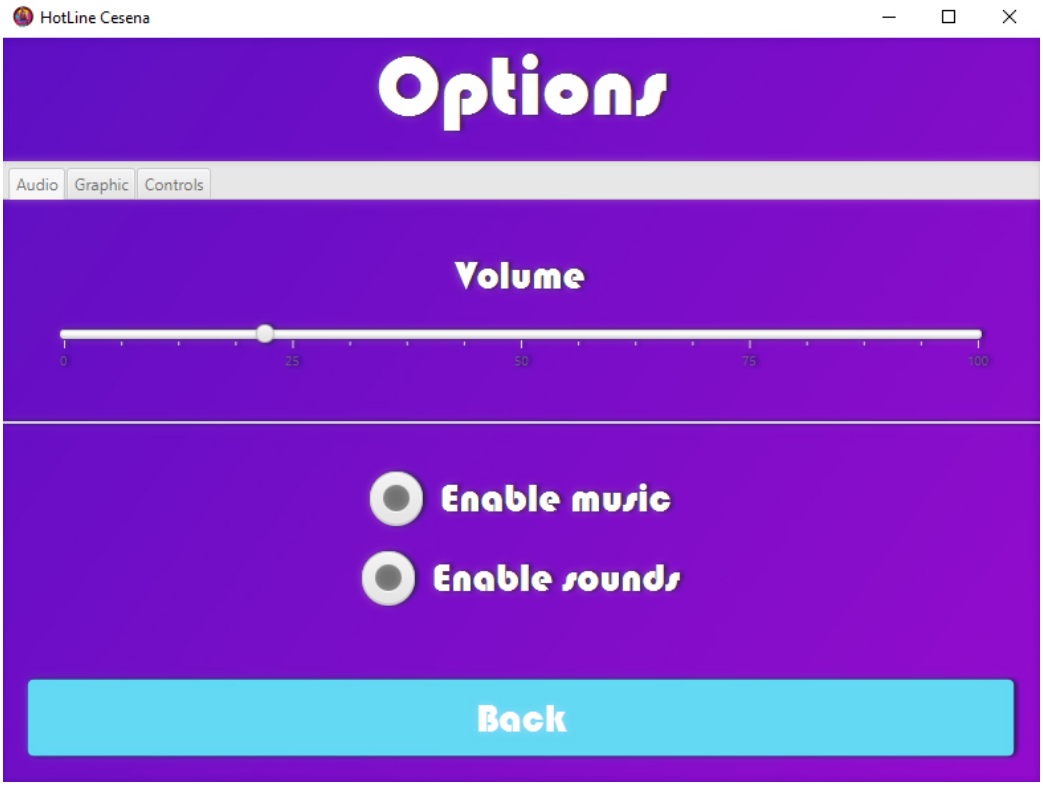


Figura A.2: Option menu

Alla pressione del bottone *New Game*, verrà caricata una nuova scena contenente il mondo di gioco (simile a come mostrato in ***Figura A.3***).

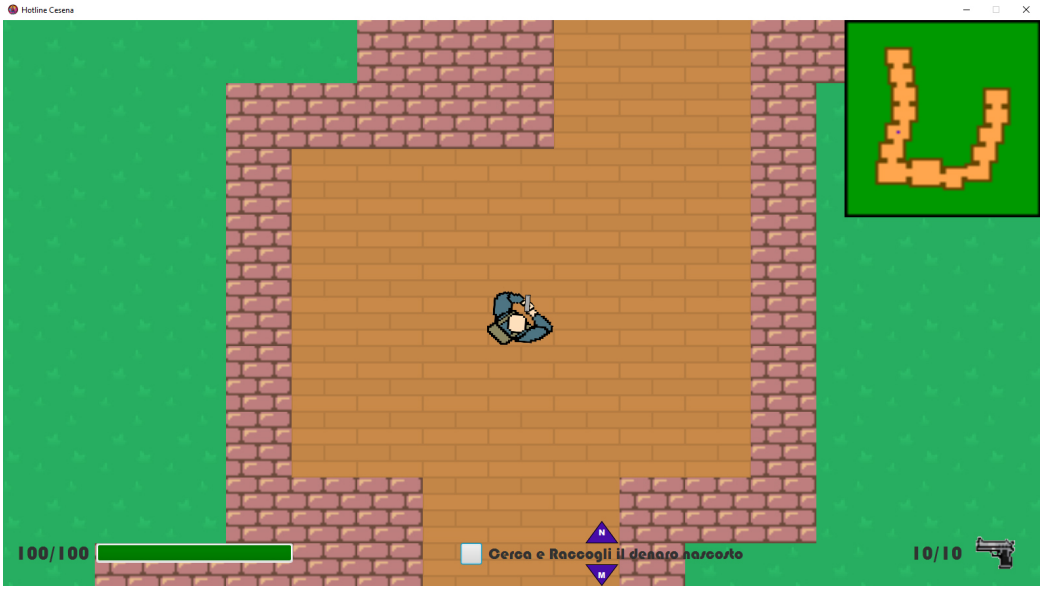


Figura A.3: Game scene

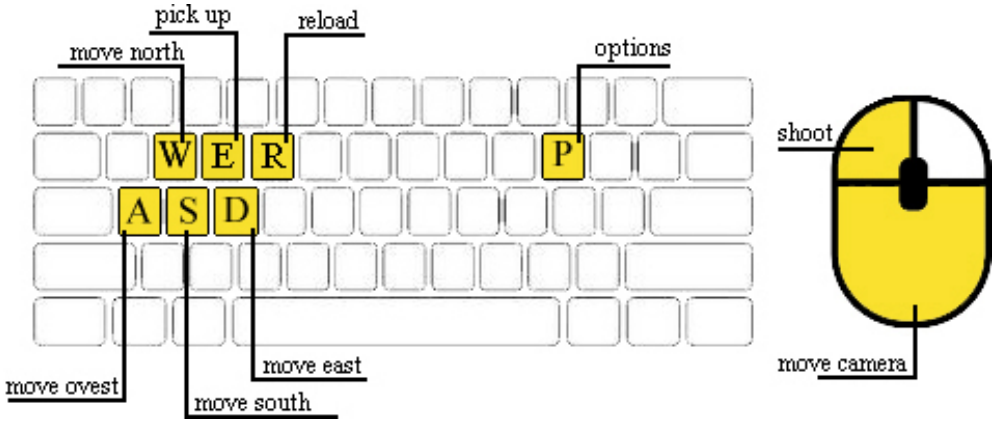


Figura A.4: keybindings

Terminata la partita verrà mostrata la leaderboard locale dei punteggi passati e verrà data la possibilità di salvare la sessione appena conclusa(come mostrato in ***Figura A.5***).

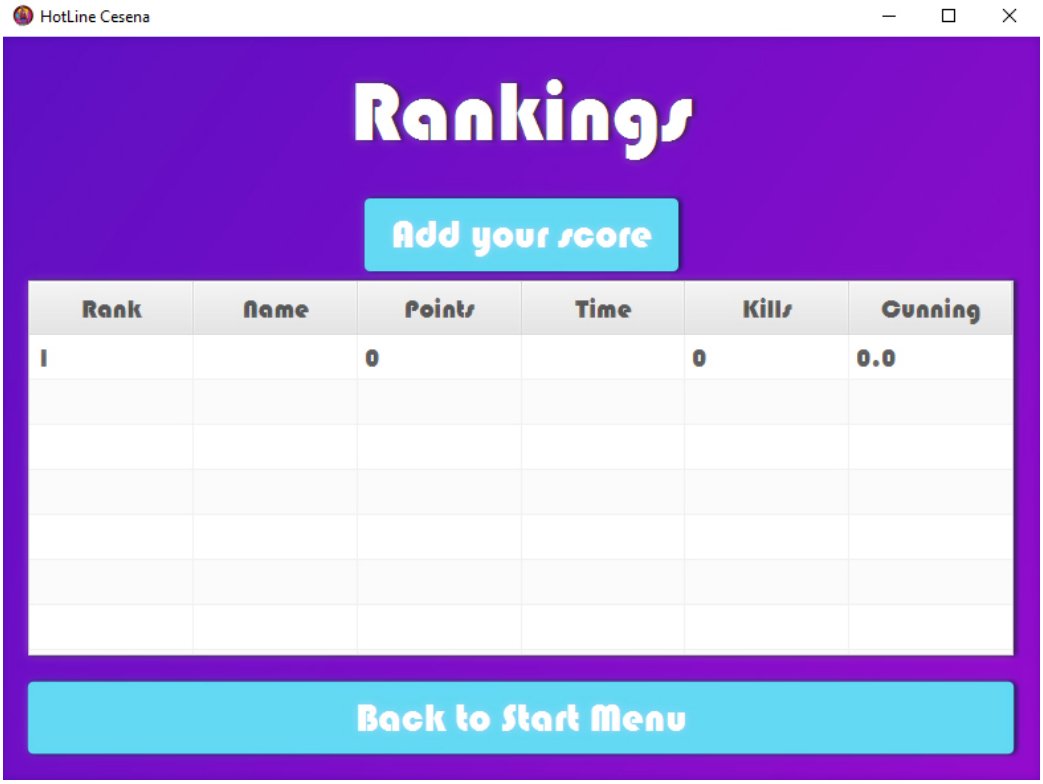


Figura A.5: Rankings menu

Cliccando su *Back to Start Menu* si tornerà al *Menu principale* per poter iniziare una nuova partita.

Appendice B

Esercitazioni di laboratorio

B.1 Bryan Corradino

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101497
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100890
- Laboratorio 07: https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p101076
- Laboratorio 08: https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102869
- Laboratorio 09: https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103916
- Laboratorio 10: https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106670
- Laboratorio 11: https://virtuale.unibo.it/mod/forum/discuss.php?d=66463#p106514

B.2 Federico Campanozzi

- Laboratorio 05: https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p101184
- Laboratorio 06: https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p100994

- Laboratorio 07: `https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p100987`
- Laboratorio 08: `https://virtuale.unibo.it/mod/forum/discuss.php?d=63865#p102757`
- Laboratorio 09: `https://virtuale.unibo.it/mod/forum/discuss.php?d=64639#p103948`
- Laboratorio 10: `https://virtuale.unibo.it/mod/forum/discuss.php?d=66753#p106708`
- Laboratorio 11: `https://virtuale.unibo.it/mod/forum/discuss.php?d=66463#p106424`

B.3 Andrea Zammarchi

Per motivi vari non ho effettuato le consegne su virtuale, ma comunque le esercitazioni sono sul mio profilo GitHub:

- **Lab01** : https://github.com/andreazammarchi3/OOP-Lab01
- **Lab02** : https://github.com/andreazammarchi3/OOP-Lab02
- **Lab03** : https://github.com/andreazammarchi3/OOP-Lab03
- **Lab04** : https://github.com/andreazammarchi3/OOP-Lab04
- **Lab05** : https://github.com/andreazammarchi3/OOP-Lab05
- **Lab06** : https://github.com/andreazammarchi3/OOP-Lab06
- **Lab07** : https://github.com/andreazammarchi3/OOP-Lab07
- **Lab08** : https://github.com/andreazammarchi3/OOP-Lab08
- **Lab09** : https://github.com/andreazammarchi3/OOP-Lab09
- **Lab10** : https://github.com/andreazammarchi3/OOP-Lab10