relazione per il progetto del corso di
Programmazione Ad Oggetti

SPACERUNNERS

A.A. 2020/2021

Andrei Nica
Salvatore Bennici
Andrew Gagliotti

Indice

1 Analisi

Il progetto presentato consiste nella realizzazione di un gioco arcade 2D denominato **SpaceRunners**. Quest'ultimo prende proprio ispirazione dal classico *Space Invaders*, pur mantenendo elementi e dinamiche di gioco differenti.

L'obiettivo del giocatore infatti è quello di accumulare più punti possibili per battere il record più alto. Il punteggio cresce ogni volta che il giocatore abbatte una nave nemica o un nemico più grande, ma può anche diminuire se viene colpito. Il gioco dispone anche di vari modificatori di stato.

1.1 Requisiti

Analizziamo i vari requisiti del progetto, alcuni di essi già esposti sul **FORUM DEL CORSO.**

Requisiti Funzionali

- **Realizzazione dei menu' di gioco:** ce ne saranno di vario tipo e tutti a scopo informativo; sfruttare i menu' condurrà il giocatore a iniziare la sua partita.
- **Implementazione degli input da tastiera per il controllo del giocatore:** quest'ultimo potrà muoversi verso destra o sinistra, avendo anche la possibilità di sparare ai nemici e raccogliere i vari modificatori di stato.
- **Gestione degli eventi di gioco:** aggiornamento HUD di gioco, aggiornamento status di gioco e infine gestione di suoni e elementi visuali.
- **Gestione delle collisioni:** il giocatore riuscirà ad abbattere una nave nemica sparandole, subirà danni se una di esse si

schianterà contro di questo, il giocatore potrà raccogliere dei modificatori di stato.

- **Gestione di entità maggiori:** dovrà essere presente una entità più potente dei singoli nemici piccoli, la quale avrà anche un comportamento diverso e potrà comparire molteplici volte, segnando quindi un aumento progressivo della difficoltà di gioco.
- **Implementazione del sistema dei modificatori di stato:** essi riguardano completamente il comportamento del giocatore, il quale verrà più o meno avantaggiato.

Requisiti non Funzionali

- **Il gioco dovrà essere efficiente nell'uso delle risorse**, garantendo perciò una buona fluidità di gioco e anche una corretta portabilità.
- **I nemici ed i proiettili dovranno essere ottimizzati** in modo tale da evitare un sovraccarico di risorse.
- **Buona accessibilità alle meccaniche e alla comprensione del gioco:** questo al fine di favorire il giocatore nella sua esperienza; si tenga a mente però che il gioco è basato sulla linea d'onda dei vari giochi *trial and error*, ovvero impari a giocare solo provando e riprovando continuamente e apprendendo piano piano le meccaniche che lo caratterizzano.
- **Un giocatore dovrebbe poter monitorare i suoi punteggi:** la competizione è sempre un fattore importante nei videogiochi arcade.

1.2 Analisi del dominio

SpaceRunners è un gioco in cui un giocatore può affrontare la sua avventura spaziale manovrando la nave all'interno di uno spazio di gioco.
Lo spazio di gioco sarà composto da un'area di medie dimensioni in cui vi sono attive numerose entità, quali: la nave del giocatore stesso, le navicelle nemiche, modificatori di stato, proiettili sparati dal giocatore, l'entità di grandi dimensioni che rappresentà il salto di difficoltà ed infine anche elementi visuali a scopo informativo (HUD).

I modificatori di stato potranno essere di vario tipo e con varie applicazioni, l'HUD dovrà riportare le statistiche attuali del giocatore e le entità nemiche dovranno essere in grado di impedire al giocatore di proseguire nella sua avventura.
All'interno del gioco sono quindi coinvolte numerose entità e componenti, le quali dovranno essere continuamente monitorate e controllate affinchè l'esperienza di gioco risulti fruibile e funzionale al suo scopo.

Risulta intuibile la sfida principale: ovvero di realizzare e monitorare la corretta interazione tra queste. Questo anche perchè la gestione delle entità non controllate dal giocatore dovrà essere completamente randomizzata e automatizzata, garantendo così una esperienza di gioco sempre varia e imprevedibile ad ogni partita.

Vediamo quali sono le varie entità in gioco con annesso anche uno schema UML generalizzato, seguendo quello che è il design del pattern MVC (Figure 1):

- **Entità:** sarà importante accumunare quante più caratteritiche possibili riguardanti le entità, infatti ciascuna avrà una sua direzione, sarà o meno controllabile dal giocatore e sarà o meno attiva nel campo di gioco. Le entità, infine, sono sempre in movimento.
- **Player:** il giocatore vero e proprio potrà scegliere come muoversi

e potrà anche attaccare i nemici, raccogliere bonus e accumulare punti.

- **Enemy:** entità non controllate dal giocatore, dovranno muoversi a schermo autonomamente.
- **Status:** entità non controllate dal giocatore e affette da generazione randomica e movimento a schermo.
- **Bullet:** un proiettile viene sparato dal giocatore, il quale dovrà essere sottoposto a dei limiti temporali per evitare che il giocatore abusi di cheats.
- **Gestore di gioco:** dovrà essere in grado di mantenere in vita le entità per un certo tempo e dovrà essere in grado di aggiornare le loro posizioni in base a come si muovono oppure terminarle se avvengono collisio.
- **HUD:** dovrà essere in grado di rappresentare a schermo i dati di alcune entità che gli verranno indicate.

Segue rappresentazione grafica del dominio.

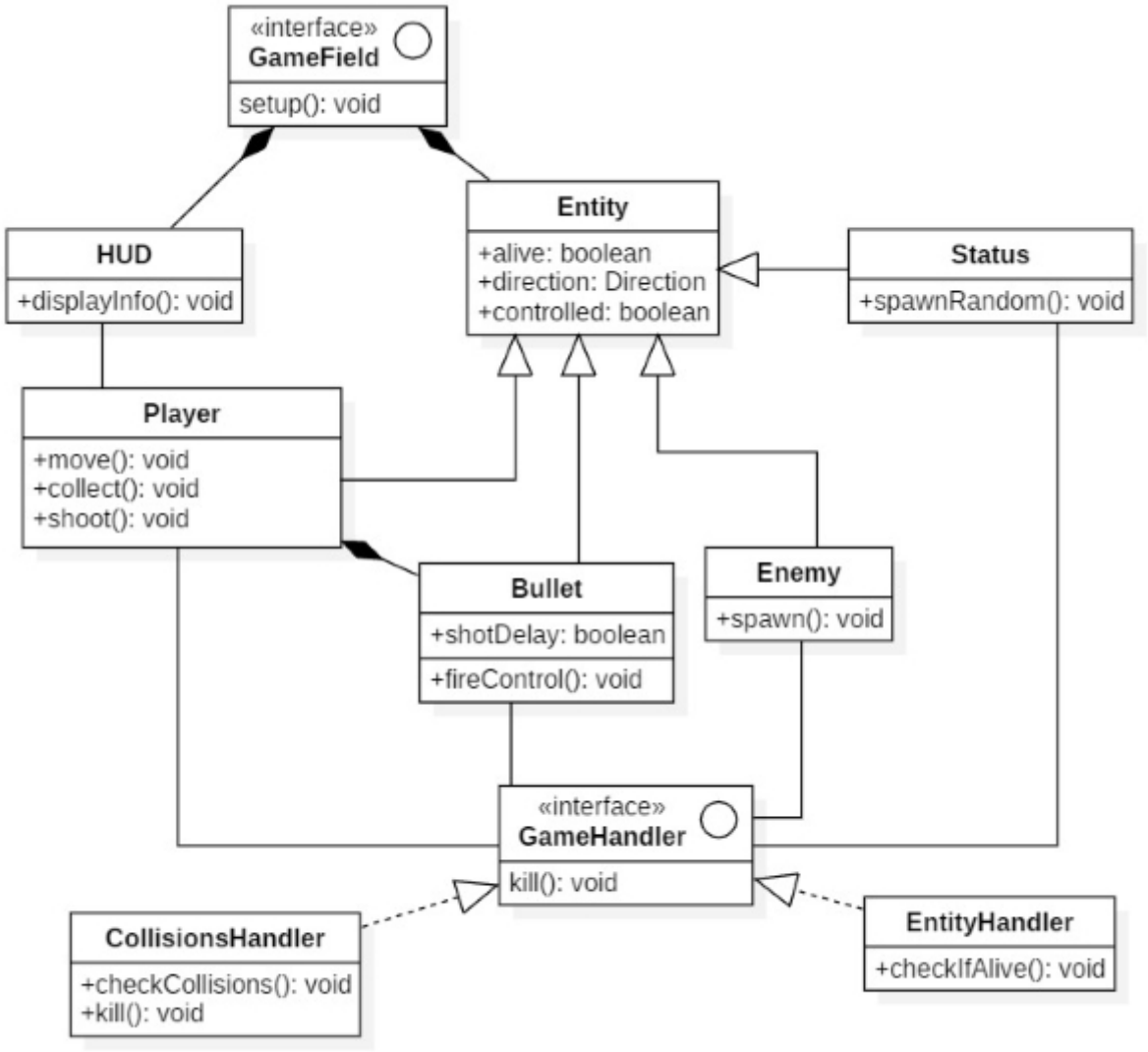

Figure 1: **MVC pattern applicato all'analisi della applicazione:** sono ben evidenziate le entità in gioco e come si relazionano fra loro. Schema realizzato da **Andrew Gagliotti**

2 Design

2.1 Design architetturale

A livello macroscopico il progetto si propone di seguire quelle che sono le linee guida del pattern architetturale MVC, creando così 3 componenti che siano in comunicazione fra loro in maniera costante. La divisione del progetto in 3 ambiti permette sostanzialmente di far sì che la componente visiva si occupi della rappresentazione grafica, che la componente di modellazione modellizzi le tecniche risolutive dei problemi preposti e che il controllore si occupi di far comunicare le due parti sopracitate, garantendo così una ottima realizzazione logico-funzionale del programma.
In particolare, il controller sarà diviso in varie componenti, ciascuna delle quali dovrà occuparsi di:

- **Controllare il funzionamento logico dei processi:** l'applicazione prima di asservire ad uno scopo deve soprattutto esistere, servirà quindi una logica dei processi e una corretta suddivisione delle risorse.

- **Controllare il corretto funzionamento delle varie schermate di gioco:** un gioco è composto da numerose schermate, le quali si dovranno alternare senza crearsi problemi a vicenda e senza danneggiare l'integrità stessa della applicazione.

- **Controllare una corretta prosecuzione degli eventi di gioco:** sarà importante impostare una logica di gioco, secondo la quale verranno scelte le regole di funzionamento dell'applicativo e quando questo deve terminare.

- **Controllare il funzionamento delle varie entità:** le varie entità esistono in contemporanea in tutto l'ambiente di gioco, sarà quindi opportuno monitorarle attentamente e conferire loro una logica di funzionamento ottimale, creandole ed eliminandole continuamente secondo un certo processo logico.

- **Controllare gli input di gioco:** un gioco non può essere tale se non si sfruttano dei comandi per poter interagire con esso, sarà quindi necessario stabilire a priori dei comandi e poi far sì che per ciascuno di essi ci sia una corrispettiva risposta all'interno dell'applicativo.

Ciascuna delle componenti che devono essere monitorate avrà una o più corrispettive parti di modellazione e, se necessarie, delle componenti che si occupino della loro rappresentazione visiva.

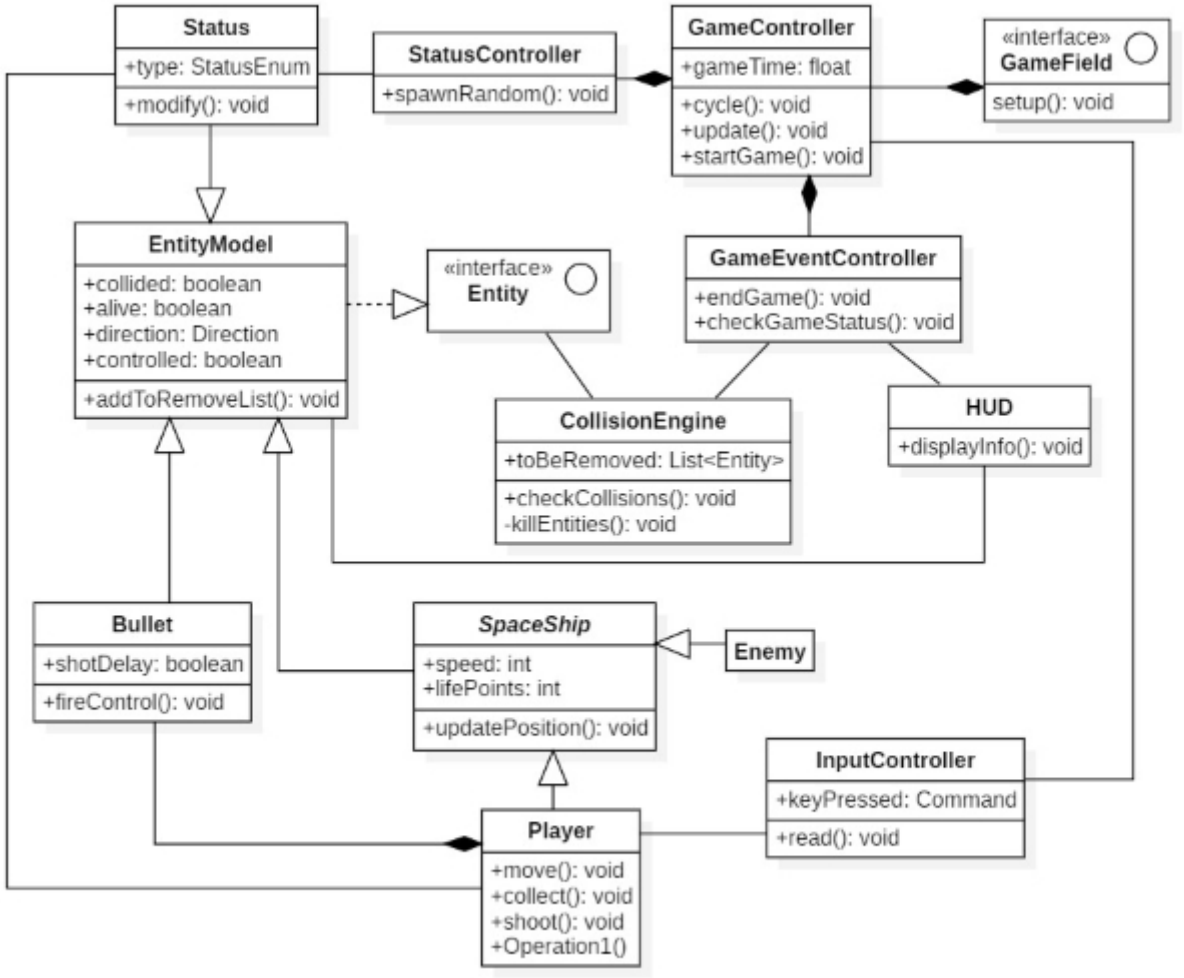

Figure 2: **MVC pattern applicato all'architettura della applicazione:** sono ben evidenziate le componenti e come si relazionano fra loro.

2.2 Design di dettaglio

2.2.1 Andrew Gagliotti

In questa sotto-sezione mi dedicherò a rappresentare quelle che sono state le mie scelte a livello di design di dettaglio nella mia sotto-parte di applicativo. Come obiettivi preposti, i miei compiti erano quelli di creare la logica dei proietili, la logica stessa di funzionamento del gioco, di implementare una interfaccia informativa real time degli eventi di gioco (HUD) e infine di trattare e realizzare la logica dietro alle collisioni fra le varie entità.

A livello architetturale ho deciso di seguire semplice il pattern architetturale MVC, senza applicare alcun tipo speciale di pattern più specifico, per quanto riguarda la realizzazione della componente relativa all'interfaccia di gioco, mentre mi sono abbandonato ad un semplice template method per la realizzazione della parte dedicata alle collisioni.
Il motivo di queste scelte è dovuto al fatto che l'HUD è una componente molto vasta, per quanto riguarda il funzionamento del gioco e per poter essere funzionale deve attingere dati e informazioni da varie componenti del programma, per esempio i modificatori di stato e le collisioni fra le entità; mentre le collisioni si basano semplicemente sulle entità attive che sono presenti nel campo di gioco.

La mia parte di progetto ha giocato un ruolo importante in quanto sono stato io a dover creare il vero game design di dettaglio, e in questo la comunicazione con tutti i membri è stata fondamentale per la mia riuscita.

L'HUD:
Nel pensare e realizzare l'HUD mi sono dedicato a ragionare su quello che era il concetto dietro di essa: l'interfaccia grafica deve informare l'utente, deve farlo in modo minimale, mirato e preciso, ed ecco perchè ho adottato un design il più possibile minimale che seguisse le linee guida architetturali del pattern MVC. Questa scelta si basa sul fatto che ho voluto attenermi al pattern generale del progetto e quindi ricrearne un piccolo microcosmo funzionale e a sé stante.
La sua portabilità e riusabilità è notevole e basta che un progetto **JavaFX** dichiari una istanza del controller per poterne quindi usufruirne gratuitamente, con piccoli accorgimenti sui nodi e sulle variabili totali: questa parte di progetto, infatti si ricollega al progetto generale solo con la classe di view **GameField** e con la classe controller **GameController**.
L'HUD è modellata su 3 sotto-classi che vengono inizializzate dalla controparte di view:

- **Bonus HUD:** Rappresenta a schermo quelle che sono le icone dei bonus raccolti e lo fa attingendo dal **GameField** i vari *tipi* di bonus (ricevendoli quindi come input). Una volta che un bonus viene raccolto, verrà mostrato a schermo (in basso a destra) fin tanto che è necessario che lo sia. Come ulteriore controllo e supporto per la fase di test, ho deciso anche di tenere traccia di quali sono i bonus attivi e quali no, con una struttura dati che è esterna da quella di instanziazione dei bonus. A livello pratico i bonus sono modellati tramite un array di **ImageView**. Segue Diagramma UML della sotto-parte di model appena descritta:

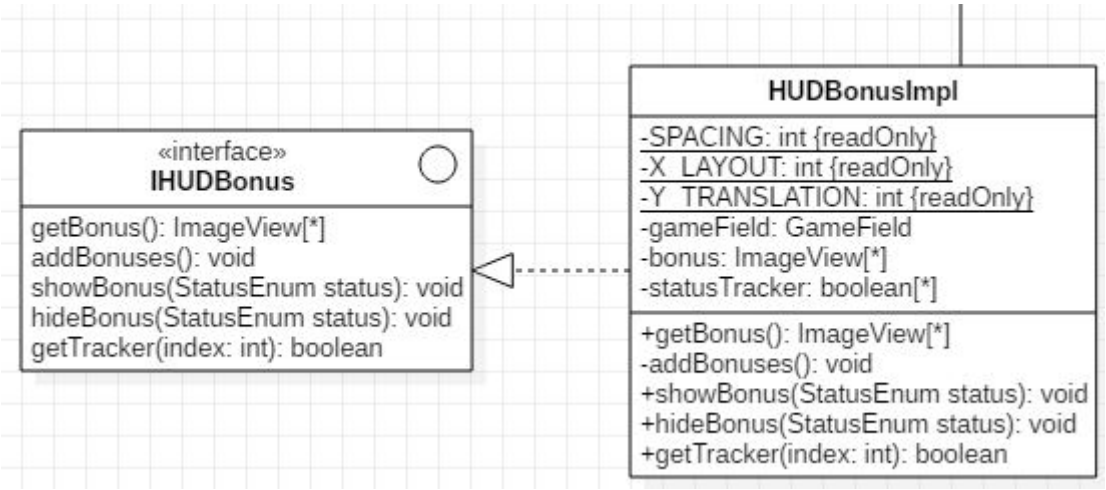

Figure 3: **Bonus HUD, design di dettaglio:** sono ben evidenziati i campi e i metodi utilizzati.

- **Lives HUD:** similmente a quella che è stata la realizzazione della componente visiva dei bonus, ho deciso di sfruttare anche qui un array di **ImageView** per rappresentare le vite (in alto a sinistra), le quali vanno anche a determinare quella che è la condizione di progressione di gioco: se un giocatore perde tutte le sue vite allora il gioco termina e si notifica la classe **GameEventController** di dover terminare il *game cycle* e di lanciare la GUI di fine gioco. Segue Diagramma UML:

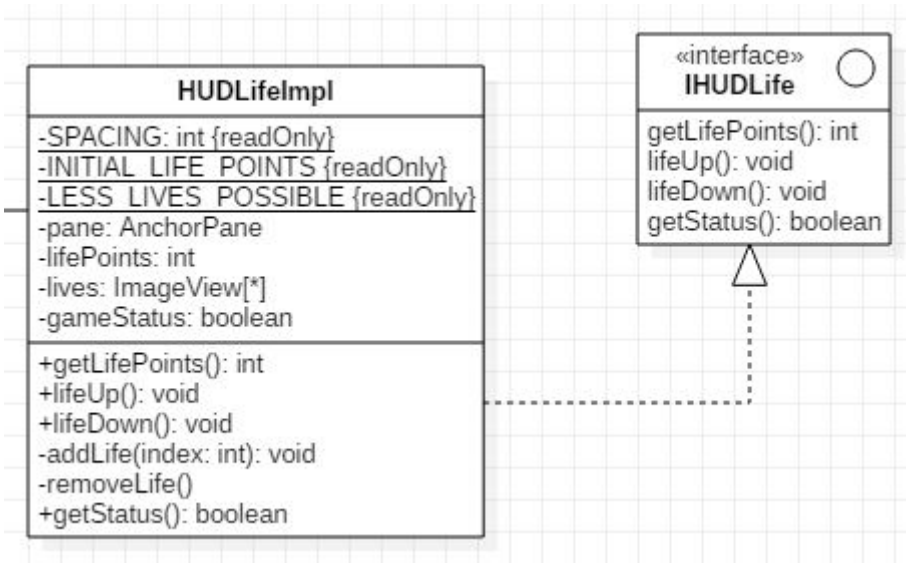

Figure 4: **Lives HUD, design di dettaglio:** sono ben evidenziati i campi e i metodi utilizzati.

- **Points HUD:** A livello grafico, il punteggio è rappresentato da una **Label** che si posiziona in alto a destra nella schermata di gioco. Al suo interno è contenuta una stringa di testo che viene aggiornata man mano che il player guadagna o perde punti. Essendo una **Label**, **JavaFX** l'ha considerata come un vero e proprio nodo dell'applicativo generale e quindi in questo punto del progetto ho avuto la necessità di dichiarare all'interno del **GameEventController** come questo oggetto si dovesse relazionare al interno di tutto il sistema di nodi totale.
  Il sistema dei punti è realizzato in modo tale che i punti non possano mai scendere in negativo. Segue Diagramma UML:

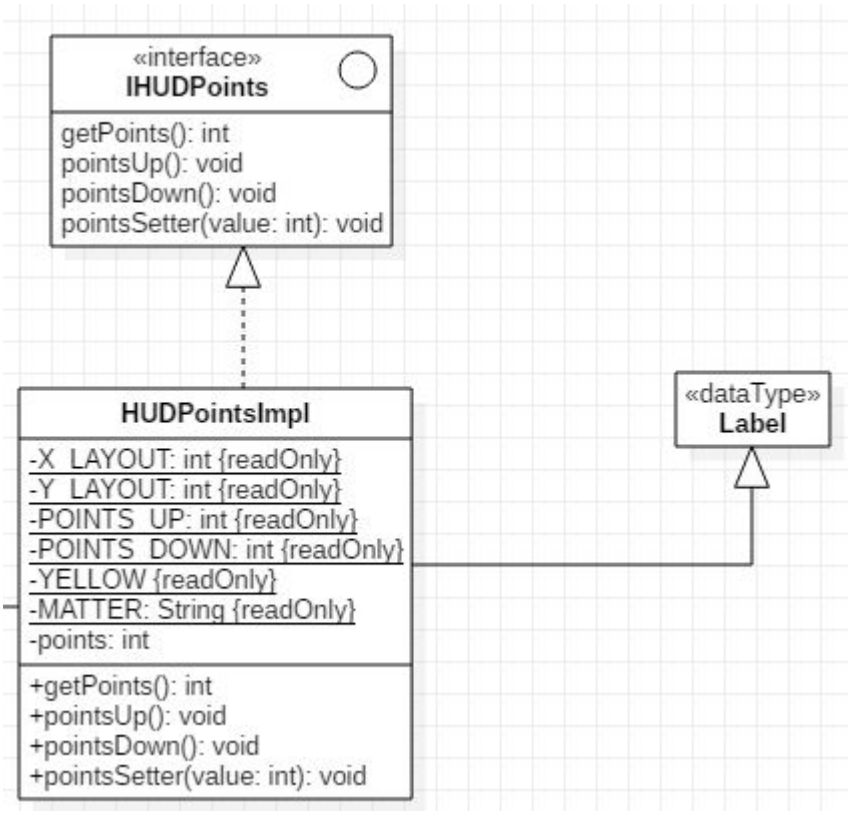

Figure 5: **Points HUD, design di dettaglio:** sono ben evidenziati i campi e i metodi utilizzati.

Le collisioni:
Per la realizzazione delle collisioni, mi sono abbandonato ad una semplice implementazione di un Template Method con funzione di supporto alla HUD: quando avviene una collisione è importante che la HUD si aggiorni correttamente e questo è un vincolo sempre presente e costante all'interno di un gioco. Quindi è importante che, successivamente ad una collisione, l'HUD si aggiorni ma è anche importante che ci sia una logica corretta di realizzazione delle collisioni in sè; in sostanza, le collisioni si dividono in due parti:

- **Template Method di supporto:** come detto, si occupa di coordinare la parte di view della HUD con l'update delle varie entità da parte del controllore stesso delle collisioni.
- **Interfaccia classica di implementazione:** da questa è possibile dedurre quali sono le collisioni da monitorare. Per questa parte, ho deciso di sfruttare il concetto di **Bounds** che le immagini delle entità generano all'interno del gioco, cosicché si possa monitorare in modo semplice e diretto le collisioni fra entità sfruttando l'immagine stessa che viene caricata dal file system.
  Le collisioni fra le entità possono avvenire nei seguenti modi: collisioni fra il giocatore e le entità a lui esterne, collisioni fra proiettili e entità nemiche, collisioni delle entità di gioco con il bordo visivo di gioco.
  Segue il diagramma UML complessivo.

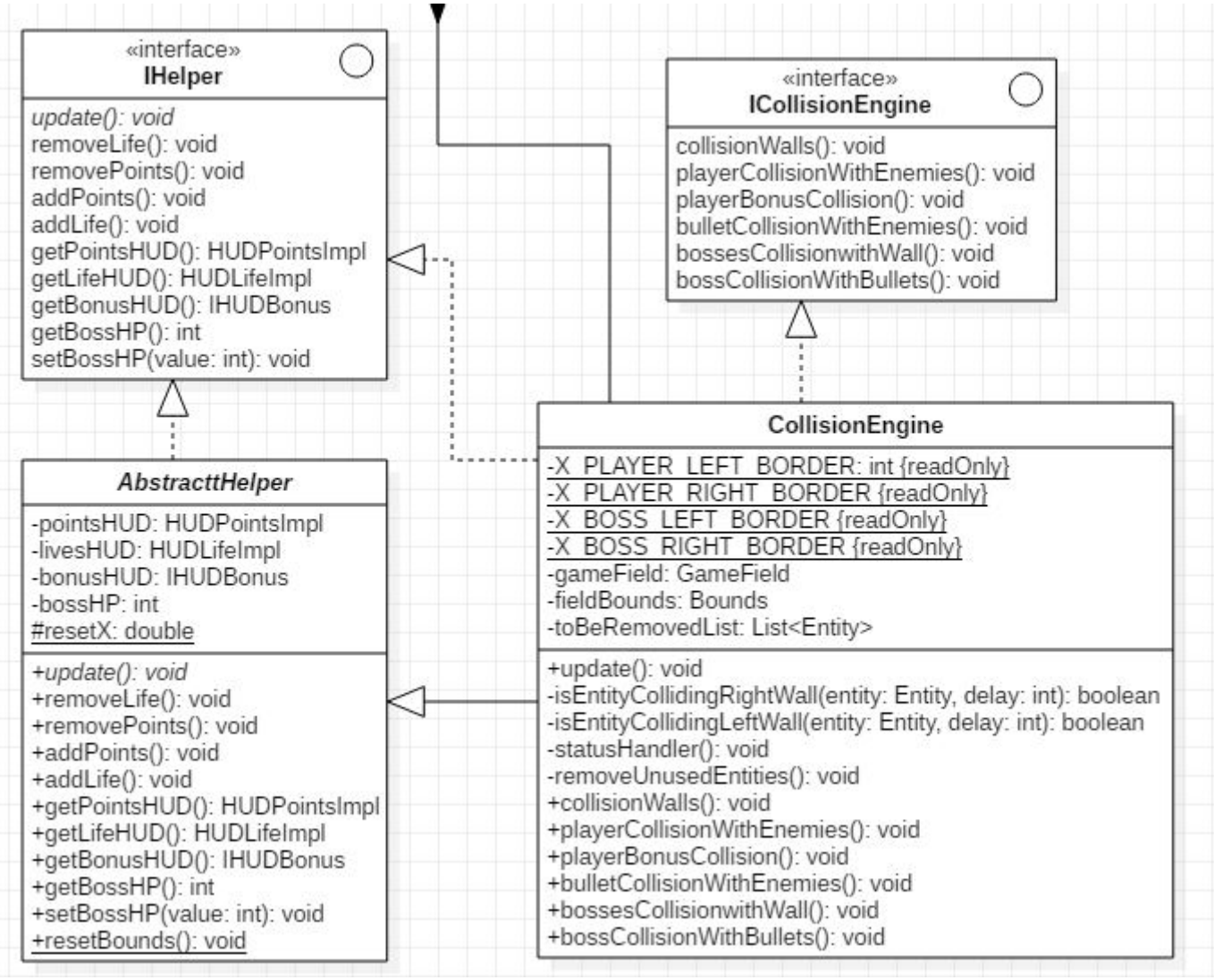

Figure 6: **Design di dettaglio del Collision engine:** sono ben evidenziati i campi e i metodi utilizzati.

La gestione degli eventi di gioco:
In tutti i diagrammi presentati precedentemente è sempre rimasto in piena vista un collegamento relazionale di tipo *contains-a* che non conduceva a nulla. Per chiarire questa notazione che potrebbe sembrare apparentemente erronea, partiamo dal considerare le componenti di model della HUD. Come già detto, l'HUD segue il pattern MVC senza scendere in specifiche implementazioni di pattern aggiuntivi e, come logico che sia, a una parte di model corrispondono una parte di view che si occupa di visualizzare e istanziare le varie componenti a schermo e una controparte di controller che si occupa di coordinare i vari elementi.
La parte di view, quindi, contiene le 3 modellazioni della HUD:

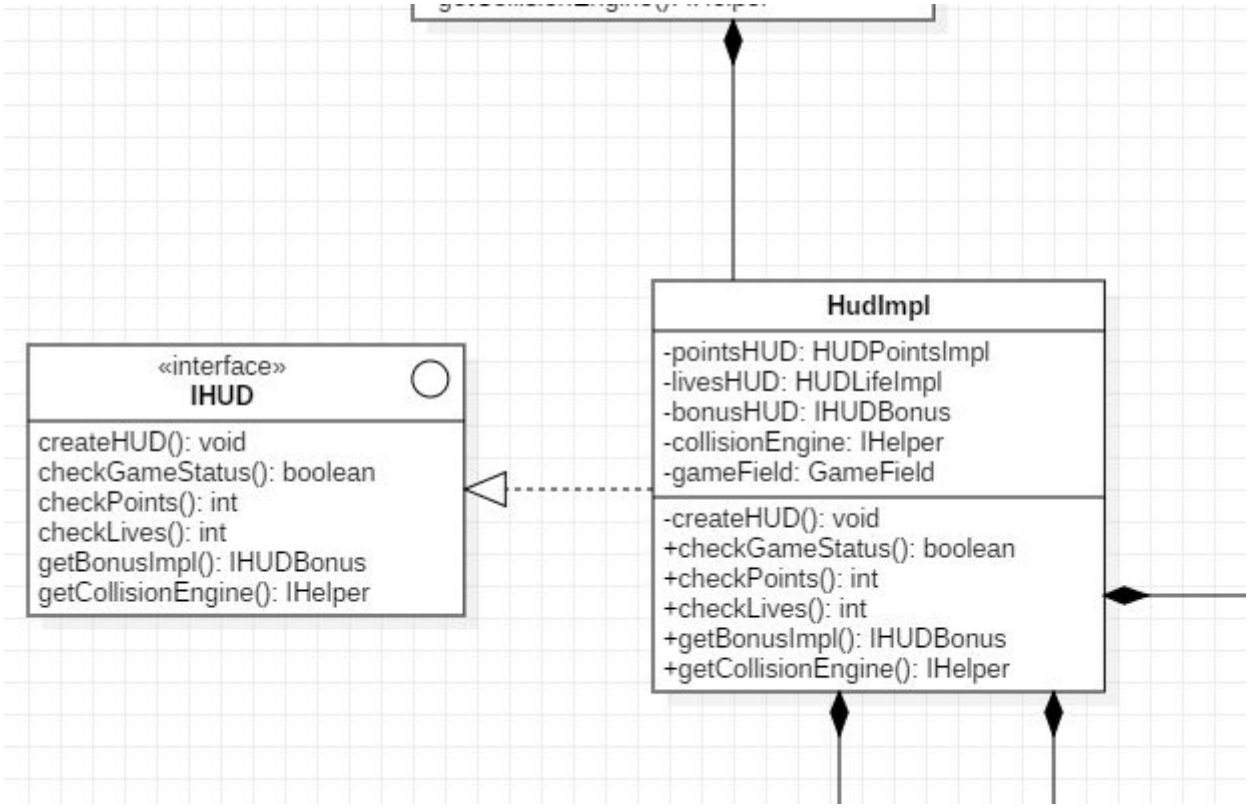

Figure 7: **Design di dettaglio della componente di view della HUD:** sono ben evidenziati i campi e i metodi utilizzati.

Non solo, siccome la parte delle collisioni si riflette sulla HUD

è quindi necessario che ci sia un controllore comune, il quale si occupa solamente di mettere in comunicazione entrambe le componenti e di garantire quindi un corretto funzionamento pratico del gioco. Come funzionalità specifica, il **GameEventController** si occupa di determinare come e quando il gioco deve terminare basandosi sulla condizione che il giocatore non abbia più vite. Segue schema UML del controller:

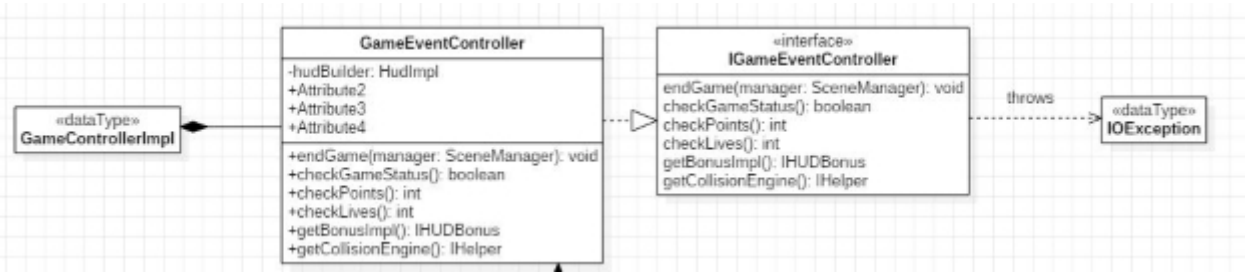

Figure 8: **Design di dettaglio della componente controller:** sono ben evidenziati i campi e i metodi utilizzati.

I menù di gioco:
Avevo inizialmente definito il comportamento della schermata di fine gioco ma con la esclusione dal progetto del collega **Andrea Arlotti**, io e **Salvatore Bennici** ci siamo affidati a *SceneBuilder* per colmare l'inadempimento del primo membro.
Inzialmente, io definii il comportamento dell'**EndGameGUI** che poi il collega ha deciso di inglobare e adattare nella sua parte di progettazione. A livello generale comunque il **GameEventController** si occupa di chiamare la fine del gioco se le vite scendono a 0, richiamando un opportuno controllore e lanciando la schermata di fine gioco.
Nel fare ciò, il *game timer* si deve arrestare e la finestra di gioco deve essere nascosta (altrimenti il *container* di gioco si *rompe*, distruggendo il sistema di funzionamento delle collisioni, o peggio).
Come supporto al collega **Salvatore Bennici**, ho deciso di curare personalmente l'aspetto grafico e di stile, aiutandolo anche a risolvere i vari bug e problemi incontrati in fase di progettazione.

Il level design:
Nel testare e realizzare le collisioni, mi sono dedicato a definire quelli che erano i comportamenti che dovessero avere le entità di gioco, in particolare:

- **Boss:** ho personalmente sistemato certi problemi che aveva l'entità nemica maggiore, la quale tendeva a generare bug dopo la sua prima generazione.
  Nel risolvere ciò ho definito il suo comportamento sfruttando il movimento randomico *sinistra-destra* definito originariamente dal collega **Andrei Nica** e conferendogli una maggiore logica fissa e più forte: il boss si muove orizzontalmente molto velocemente e se impatta con un bordo scende di 10 pixel.
  Nonostante la grande velocità di movimento del boss, è facilmente eliminabile e sconfiggerlo conferisce punti aggiuntivi. Il boss, di default, ha 10 punti ferita, ma per complessità aggiuntiva esso non avrà una barra degli HP visibile a schermo, ma conferirà feedback sonoro se colpito.

- **Logiche della HUD:** dovendo ragionare sulle componenti visive di gioco, ogni nemico ucciso conferisce punti ma è grande la punizione se si viene colpiti, in quanto si perdono sia vite che punti.
  Le vite possono essere sia guadagnate che perse ma hanno un *cap* molto basso, in particolare 4.
  Il design grafico è minimale per garantire una maggiore visione sul campo di gioco e con il sistema dei nodi di **JavaFX** ho potuto posizionare l'HUD visivamente più sopraelevata rispetto alle altre entità.

- **Logiche delle collisioni:** Le hitbox sono state minimizzate e migliorate rispetto alla versione originale per rendere ancora più complesso il gioco. La precisione è quasi del tutto massima e poco permissiva.
  Diventano quindi fondamentali i riflessi del giocatore.

- **Cura delle immagini:** Ho scelto appositamente immagini appetibili e molto minimali per rendere migliore la resa grafica complessiva.

- **Logiche dei suoni:** Ho scelto appositamente suoni divertenti e molto *easter egg* per far divertire il giocatore e immergerlo nel mood di gioco. A questa ultima parte, allego il diagramma UML della classe sulla logica dei suoni realizzata con il collega **Salvatore Bennici**:

Figure 9: **Design di dettaglio del SoundManager.**

Enum di supporto per la cura del codice:
Per me è stato vitale che il codice fosse ordinato, completo e rispettoso delle regole e dei principi di programmazione Java. A supporto della eliminazione dei magic number e di elementi ridondanti, mi sono affidato a delle enum da me create, le cui componenti vanno a riempire tutti i punti del codice. Seguono i rispettivi diagrammi UML:

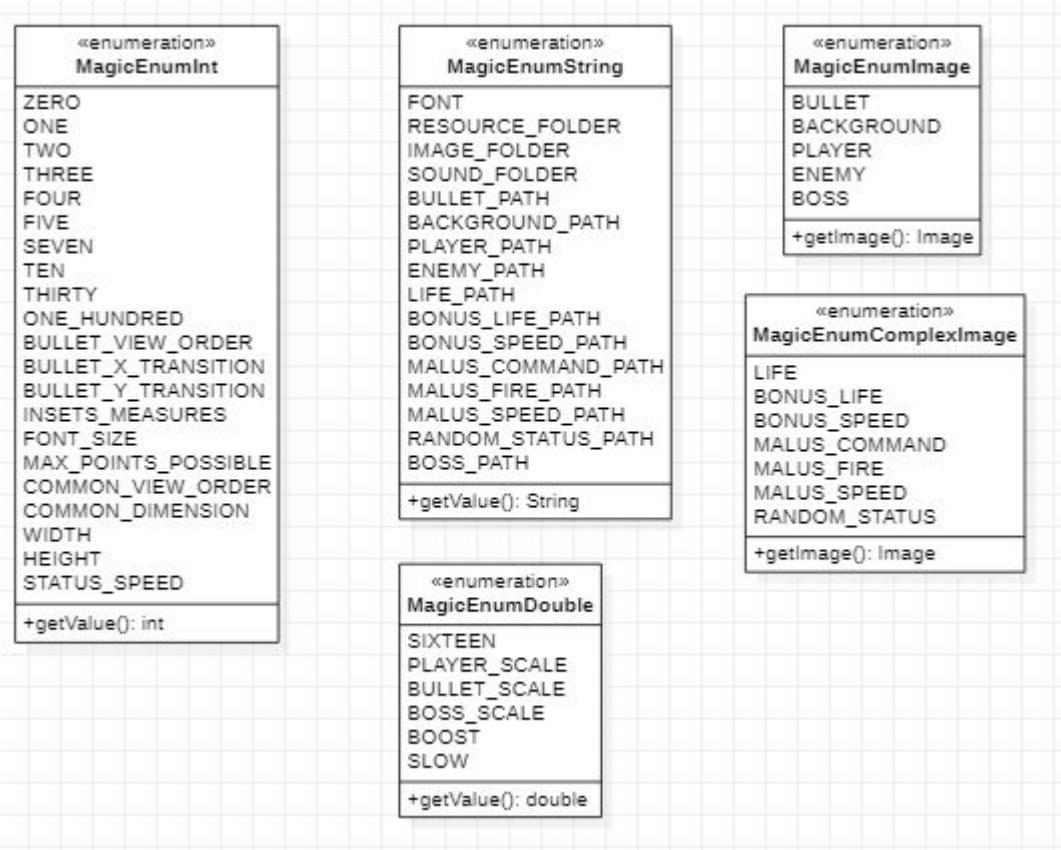

Figure 10: **Design di dettaglio delle Enumerazioni:** sono usate in quasi tutti i sorgenti del codice e migliorano la leggibilità dello stesso.

2.2.2 Salvatore Bennici

Menù di gioco

Per la realizzazione dei menù di gioco, abbiamo usufruito del tool *SceneBuilder*. In particolare, come già citato precedentemente, il collega **Andrew Gagliotti** si è occupato della realizzazione grafica, mentre personalmente ho avuto il compito di realizzarne la gestione complessiva.
In seguito all'analisi di progetto si è voluta realizzare la presenza di diversi sotto menù, dunque necessitando un meccanismo di scambio tra le varie finestre di gioco, ne ho definito la logica nell'intefaccia *SceneManager*.

Creazione e gestione delle finestre
Concettualmente, si vuole sempre sostituire il contenuto di una finestra, senza crearne di nuove. Per realizzare tale compito, sfruttando JavaFX, possiamo semplicemete sostituire la *Scene* contenuta nello *Stage* principale dell'applicativo. Avendo utilizzato *SceneBuilder*, ho avuto la necessita di creare le rispettive *Scene* a partire da file FXML (come si può notare nell'implementazione di *SceneManager* stesso) e succesivamente affiancarne il relativo controller. Abbiamo realizzato un totale di 5 menù:

- **Start**: È il Menu principale, visualizzato all'avvio dell'apllicativo. Il giocatore può scegliere di navigare verso altri sotto menù, o semplicemente iniziare una nuova partita.
- **Nickname**: Finestra precedente all'avvio del gioco, si richiede l'inserimento di un nickname da parte del giocatore.
- **Scores**: Accesibile dal Menu principale, mostra la classifica locale dei migliori giocatori.
- **Controls**: Accesibile dal Menu principale, permette di personalizzare le associazioni dei comandi da utilizzare in gioco.

- **EndGame**: Visualizzata una volta terminata la partita, mostra informazioni relative al gioco e permette di ritornare al Menu principale.

Controllers dei Menù

Una volta create le singole classi di controllo per ciascuna *Scene*, e sapendo che è opportuno creare una sola associazione(Scene-Controller), ho preferito adottare una semplice gerarchia a partire da una classe astratta, *BasicFXMLController*. Questa scelta è dovuta principalmente alla presenza di funzionalità e campi comuni.

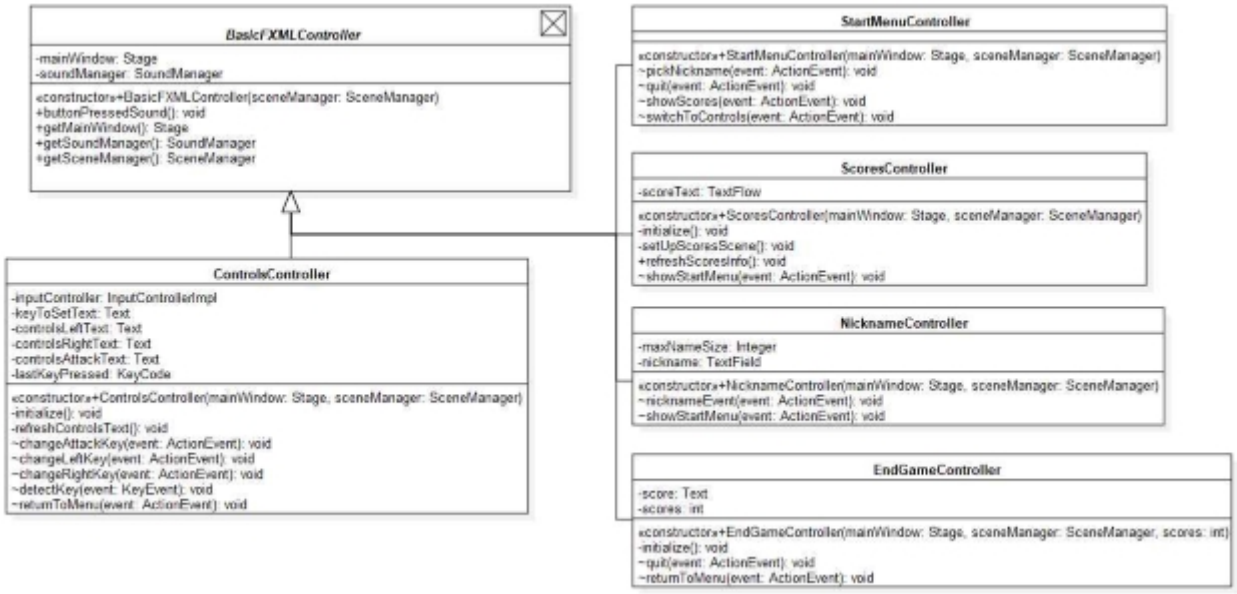

Figure 11: **Gerarchia controller delle finestre di gioco**

Ogni controller avendo accesso a *SceneManager* riesce automaticamente a far cambiare, quando necessario, il contenuto visualizzato nella finestra.

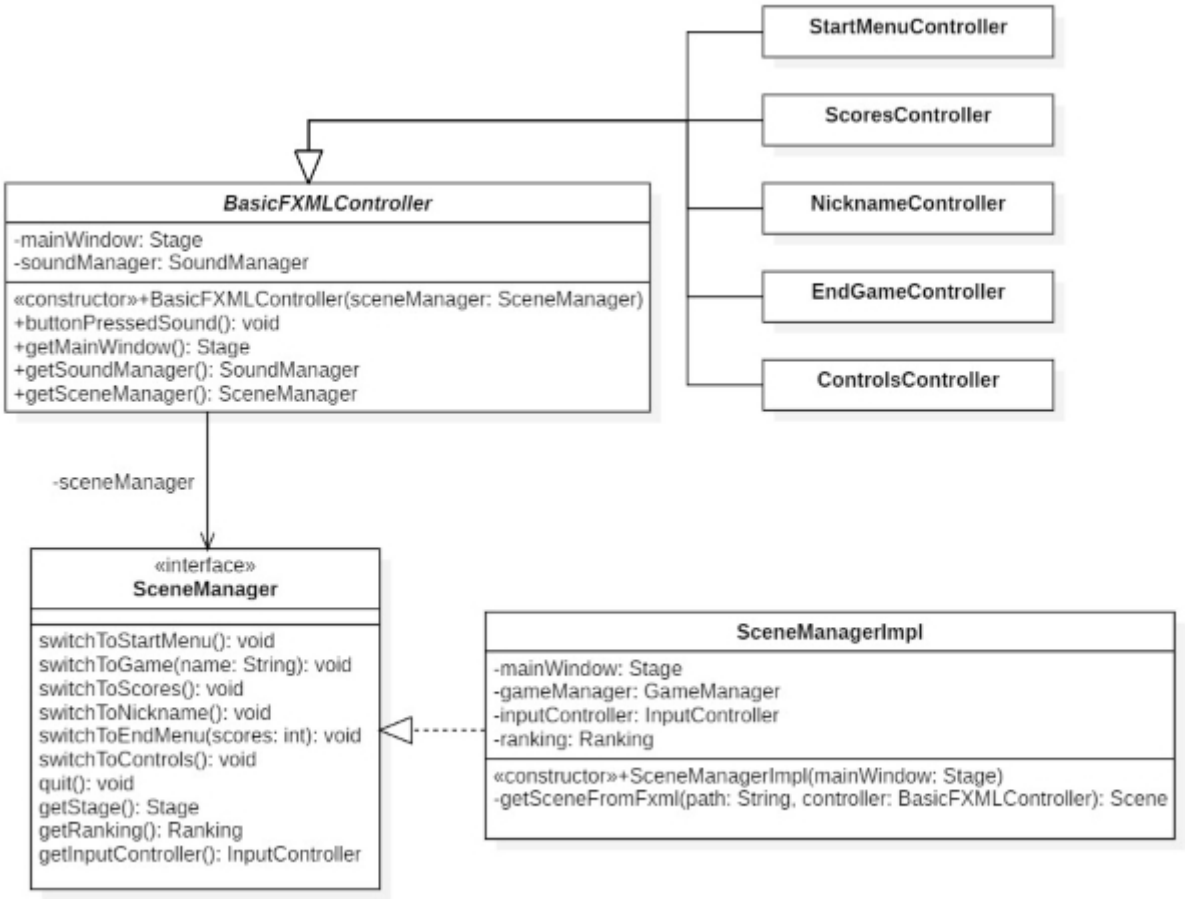

Figure 12: **Architettura di gestione delle finestre di gioco.**

Ranking System

Durante la fase di analisi dell'applicativo, si decise di implementare una classifica locale dei punteggi, dove verrano visualizzati i nomi dei migliori giocatori con i relativi punteggi associati. L'utente dunque, una volta terminata la sessione di gioco, potrà confrontare il proprio score con quelli presenti in classifica e magari ritrovarsi fra questi. Banalmente, si collezionano i nomi dei giocatori con i loro rispettivi punteggi all'interno di una Map. Questa viene succesivamente elaborata per essere salvata su file locale. É possibile aggiungere una nuova associazione, ma non eliminarne di già presenti. Si tiene inoltre

presente, che verrà soltanto salvato il punteggio più alto per ciascun giocatore.

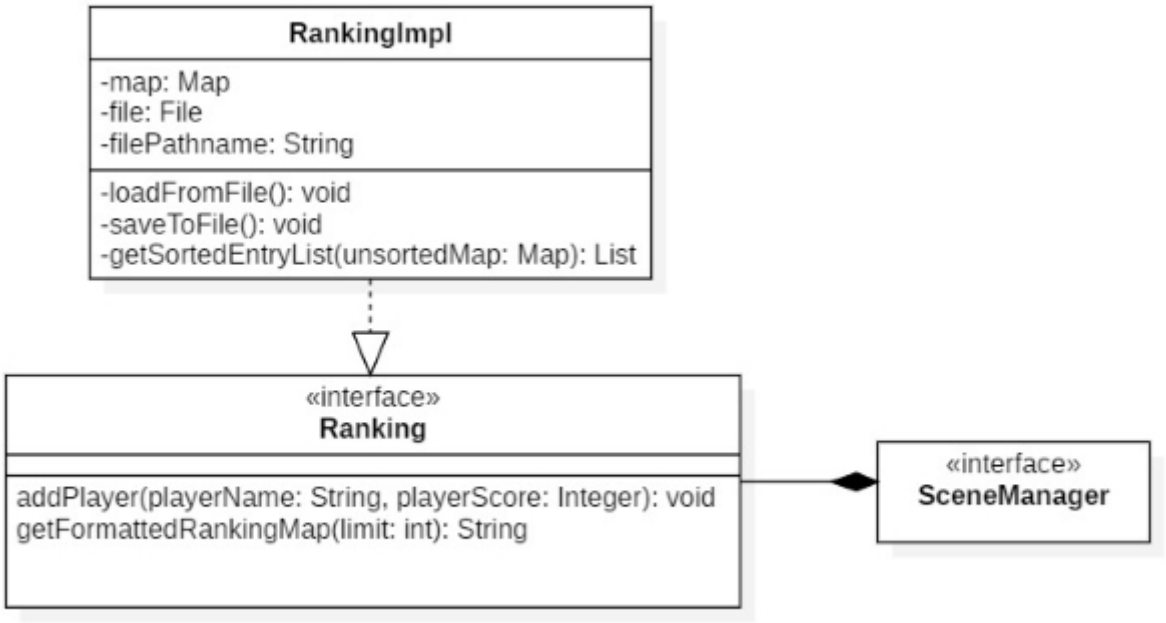

Figure 13: **Semplice struttura del Ranking System**

Status

Volendo aggiungere un po' di funzionalità al gioco, si è deciso di implementare degli effetti applicabili al giocatore. Ho rappresentato questa idea tramite gli *Status*. Definiamo quest'ultimo come l'astrazione di un'entità che, allo scatenarsi di determinate condizioni (collisione), applica al player un alteratore. Spesso, tali effetti sono solo temporanei, e la loro durata varia a seconda dell'implementazione. Vi sono due tipi di *Status*: *Bonus* e *Malus*. Banalmente, i primi favoriscono il giocatore mentre i secondi lo ostacolano. L'enum *StatusEnum* contiene tutti i vari tipi realizzati. In particolare:

- **BonusLife**: Aggiunge un punto vita al giocatore. Questo bonus

ha effetto se il giocatore non ha raggiunto il numero massimo di punti vita possibili(4).

- **BonusSpeed**: Incrementa temporaneamente la velocità di movimento del giocatore.
- **MalusCommand**: Inverte temporaneamente le direzioni di movimento del giocatore.
- **MalusFire**: Disabilita temporaneamente il fuoco del giocatore.
- **MalusSpeed**: Decrementa temporaneamente la velocità di movimento del giocatore.

Volendo aggiungere un po' di difficoltà, ho deciso di rendere non solo la loro generazione totalmente casuale, ma anche di fare in modo che il giocatore non abbia la possibilità di distinguerli. Infatti in gioco, possiedono la stessa rappresentazione grafica. Potranno essere visualizzati solo una volta applicati, tramite la HUD. Per facilitare la creazione di tali entità, ho optato per adottare il pattern *Factory*. Questa scelta è dovuta principalmente per delegare la responsabilità di creazione.

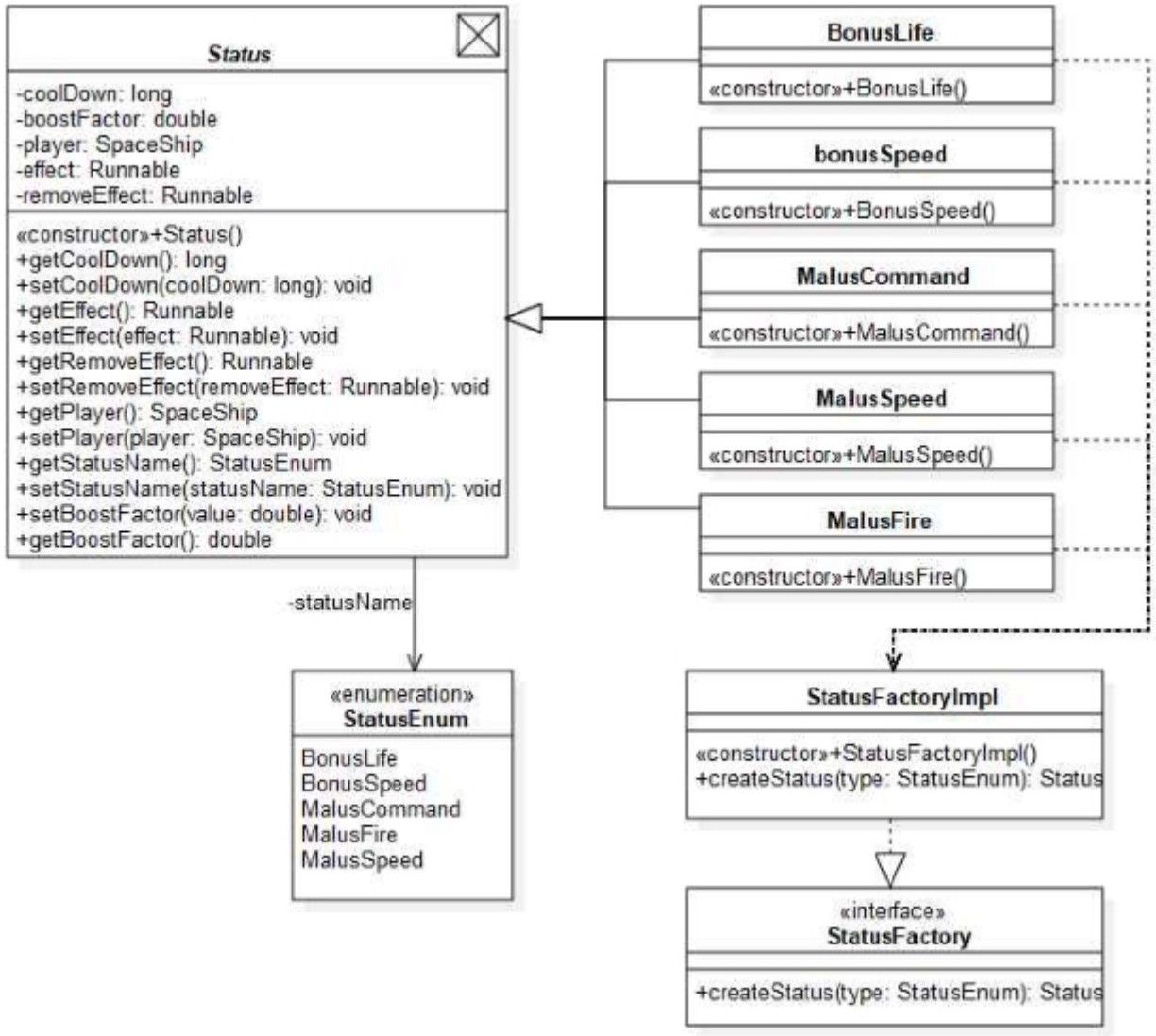

Figure 14: **Pattern Factory per la creazione degli Status**

Trattata la modelizzazione, quando concerne la parte di controllo è affidata alla classe *StatusController*. Sapendo che gli *Status* possiedono un proprio tempo di durata (cooldown), ho preferito utilizzare un Thread apposito per la loro gestione. In particolare, utilizzando *ScheduledExecutorService*.

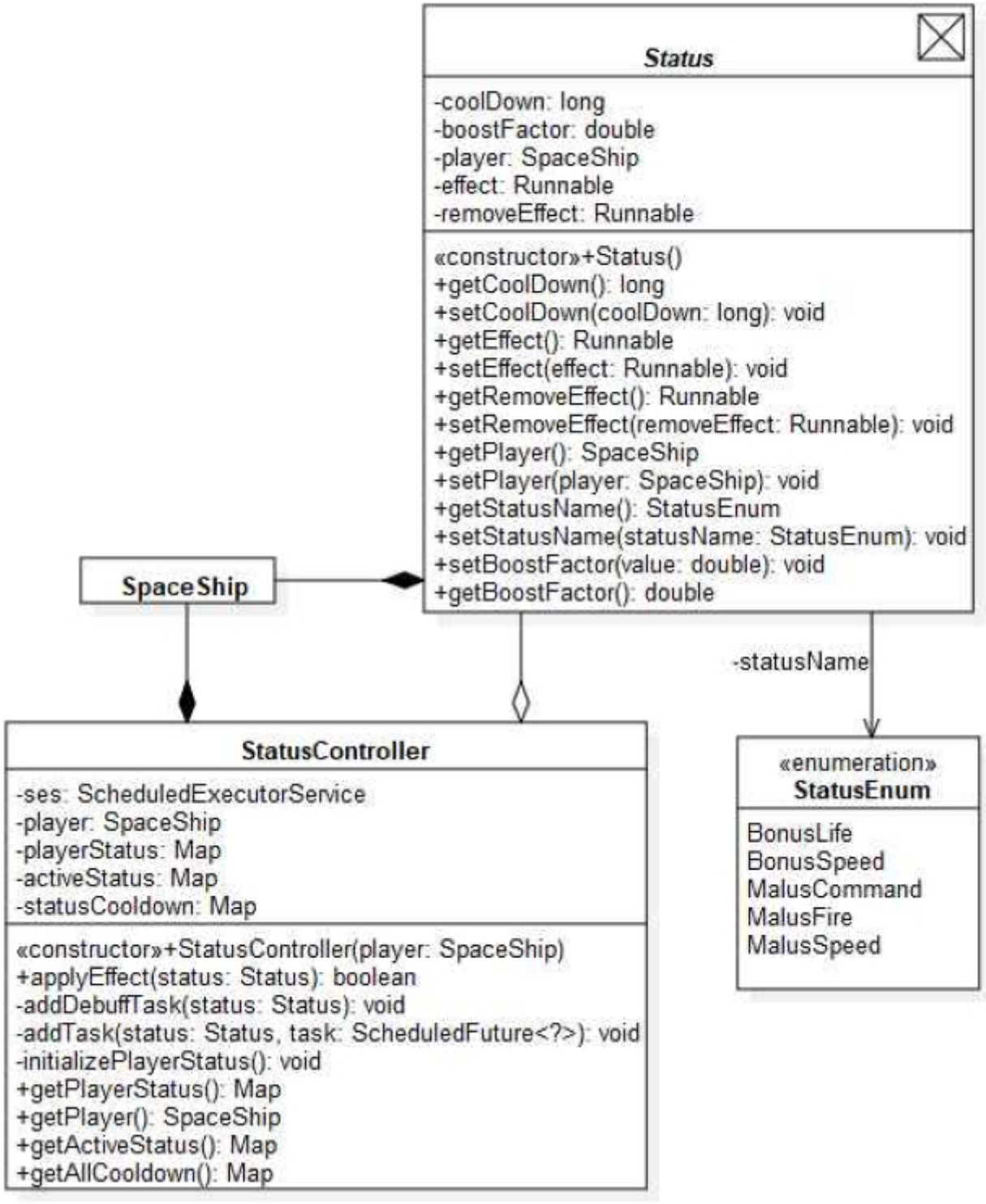

Figure 15: **Relazione StatusController**

Gestione degli Input

Volendo gestire solamente i movimenti e le azioni del giocatore, ho trovato opportuno creare una singola e semplice implementazione da *InputController* . Si tiene traccia dei tasti premuti dall'utente per poi elaborarli a seconda della finestra di gioco attuale. Seppur ad ogni tasto viene associata una singola azione, possono verificarsi combinazioni di input tali che modifichino o annulino il comportamento aspettato. Oltre a questi vi è applicato un filtro superiore nel ciclo di gioco principale dove eventi, quali ad esempio l'attivazione di determinati *Status*, possono nuovamente alterare il comportamento della nostra navicella. Ho reso possibile, come già citato precedentemente, la possibilità di associare tasti personalizzati alle varie azioni che il player potrà compiere in gioco, tramite il menu Scores. Vorrei infine tener presente che, eccezion fatta per la GUI, il gioco non prevede l'utilizzo di interazioni da mouse.

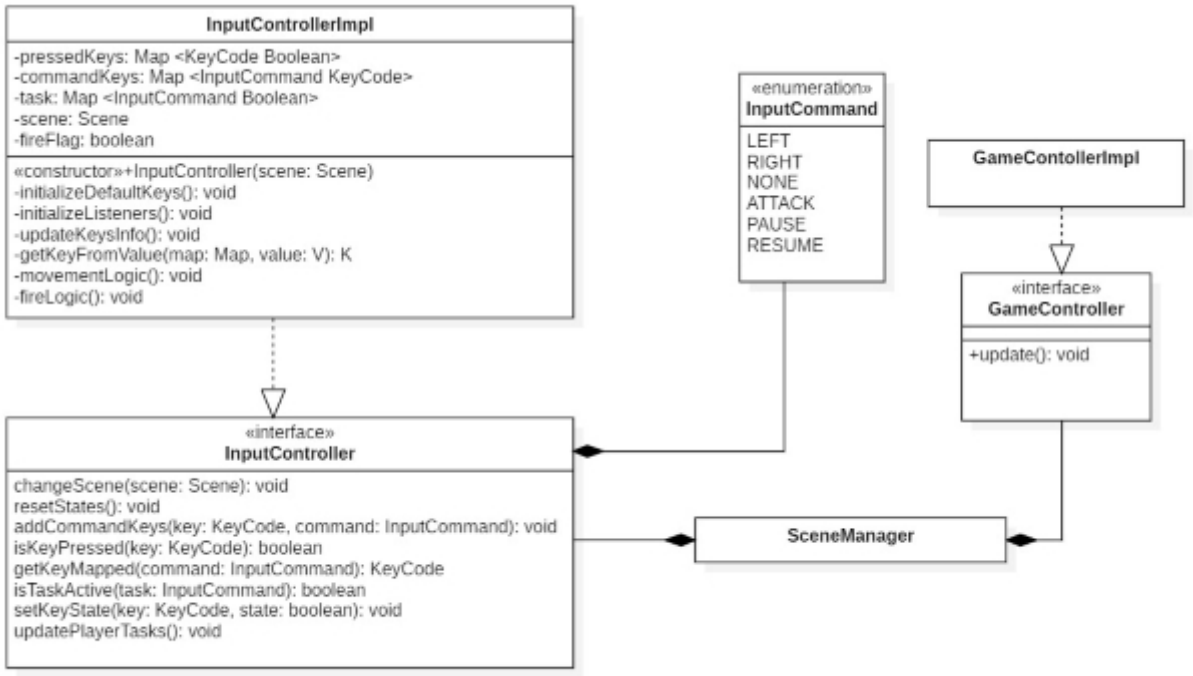

Figure 16: **Gestione input dell'applicativo**

2.2.3 Andrei Nica

GameLoop:
Ho modellato la gestione del gameloop per gestire il loop di gioco principale seguendo il modello di gestione degli input,controllo della logica e renderizzazione della grafica.
Segue diagramma UML:

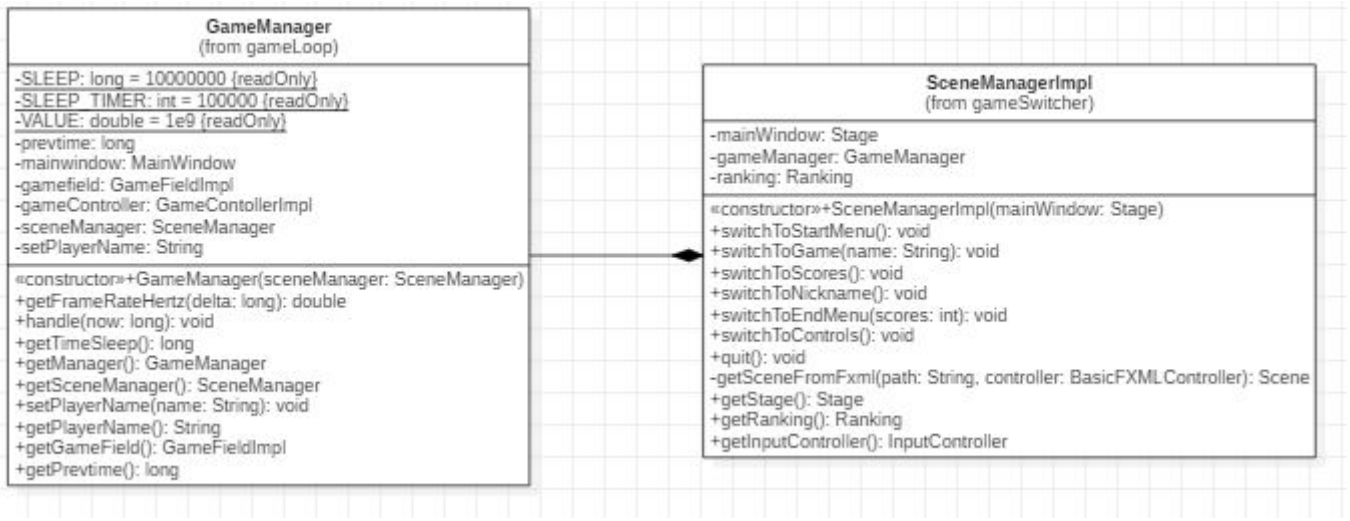

Figure 17: **Logica di gioco.**

FrameManager:
Ho creato la classe per la renderizzazione dei frame di gioco per ogni frame. Prende le posizioni di ogni oggetto del gioco, e' in base alla velocita e alla direzione aggiorna di una quantita infinitesima la posizione dell'oggetto. Oltre a fare questo, aggiorna ad ogni frame l'immagine dello sfondo per creare l'illusione del movimento.
Segue diagramma UML:

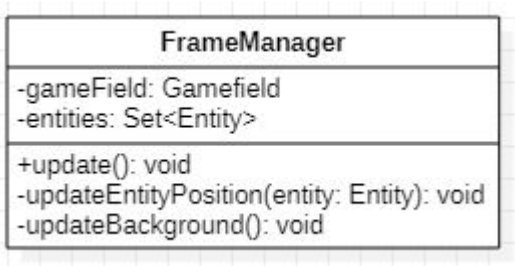

Figure 18: **Frame Manager.**

Entita Giocatore e Navi Nemiche:
Ho modellato il comportamento e il funzionamento di tutte le entita riguardanti la classe del giocatore e delle navi nemiche,che al loro interno contengono tutte le informazioni riguardanti le entità come la posizione, l'immagine e le carratteristiche riguardanti la fisica del gioco come la direzione e la velocita della nave.
Seguono diagrammi UML:

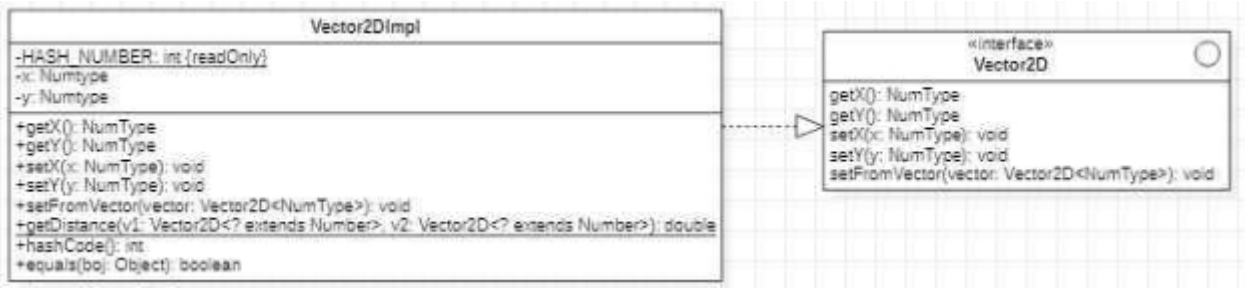

Figure 19: **Definizione base delle entità.**

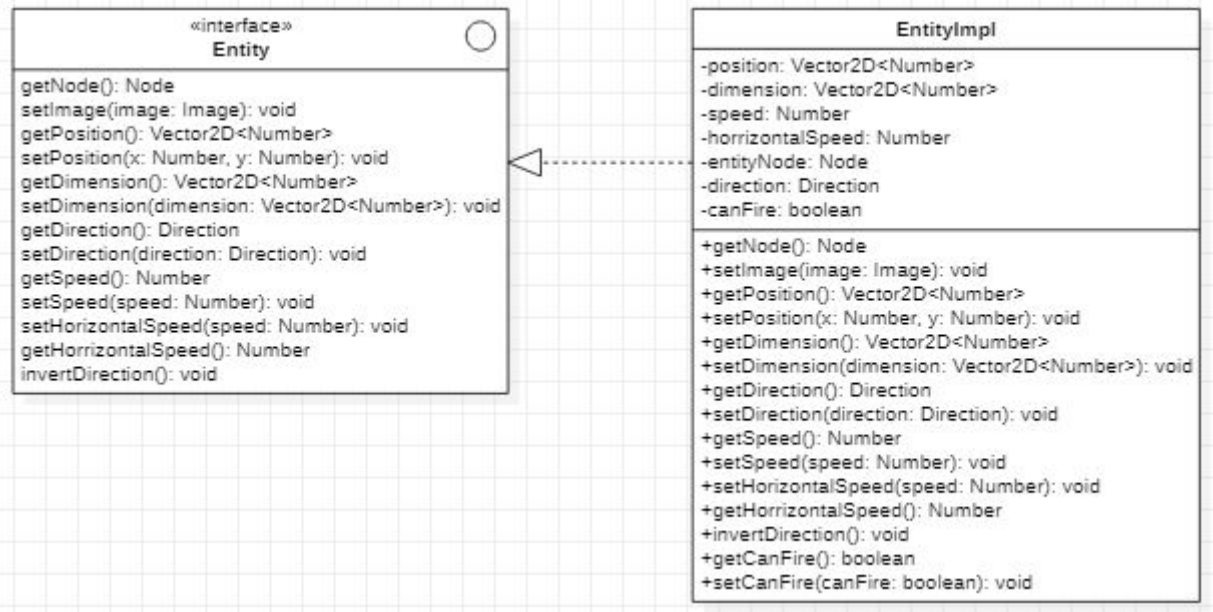

Figure 20: **Entity.**

Campo di Gioco:
Ho creato una classe che al interno contiene tutte le informazioni riguardanti la logica del gioco, come i nemici attivi, il giocatore, la posizione e la velocita di tutte le entita.
Segue diagramma UML:

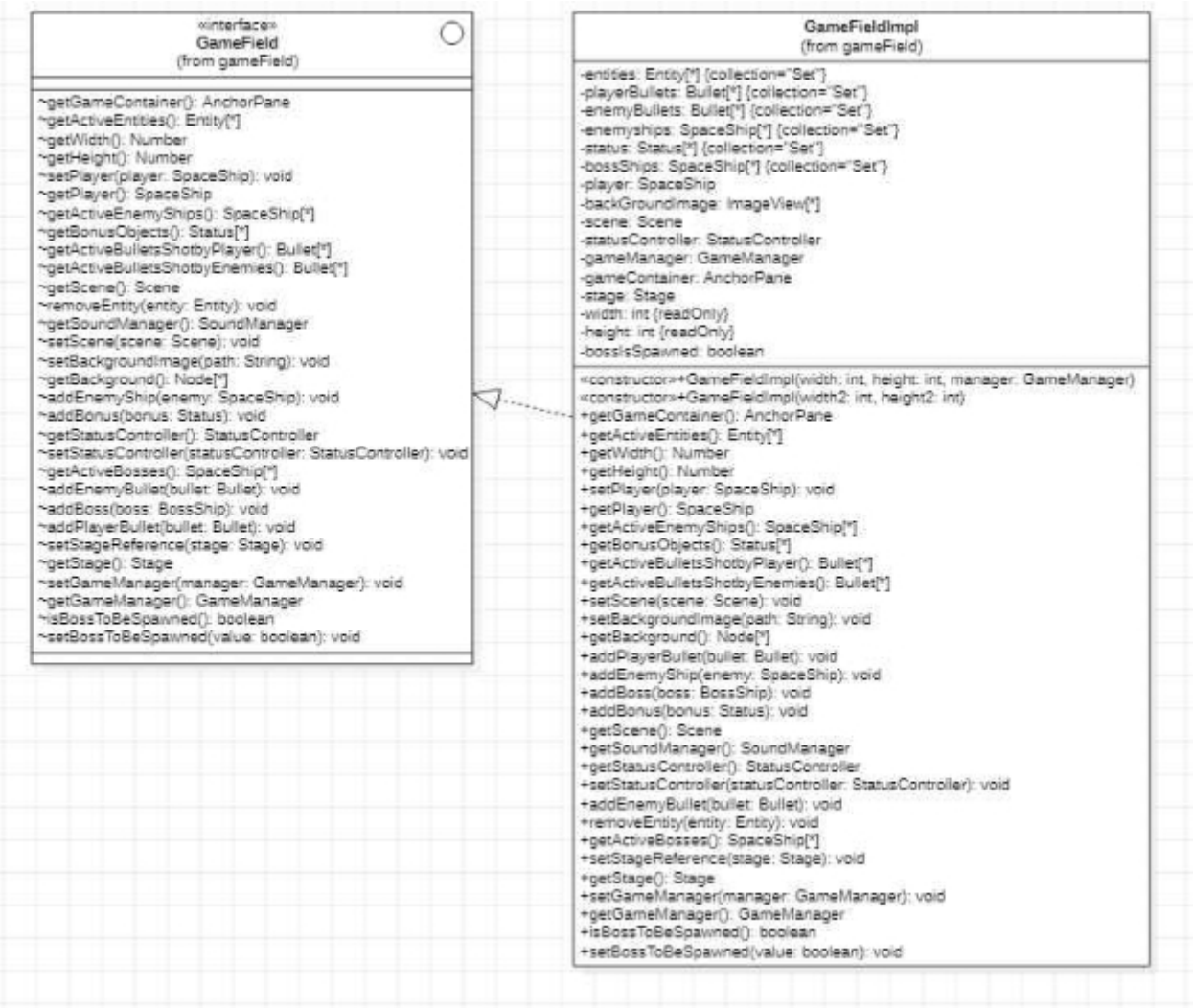

Figure 21: **Campo di gioco.**

Enemy AI:
Ho creato la classe EnemyAI per lo spawning e il movimento delle navi nemiche. Le navi vengono generate ad un intervallo casuale, in una posizione casuale.

BossAI:
Ho creato una classe separata per la gestione del Boss di fine livello, perche era piu gestibile usando un thread.
Segue diagramma UML:

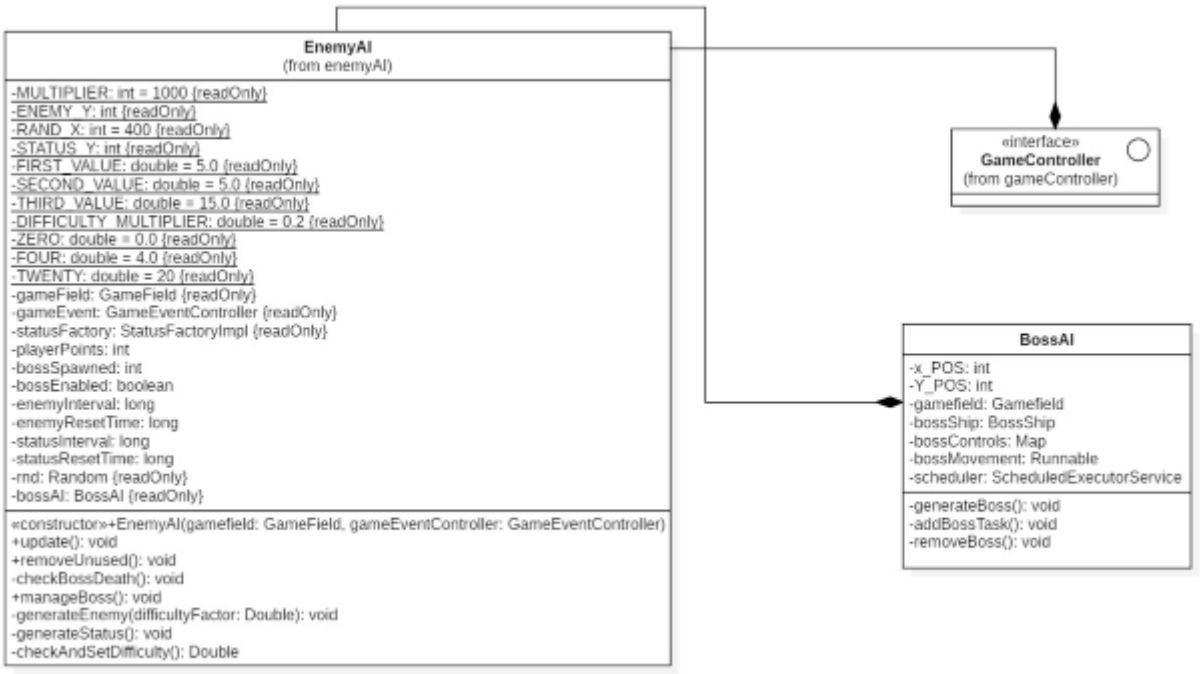

Figure 22: **Classi dei nemici**

Proiettili:
Ho Modellato la gestione dei proiettili similmente alla gestione delle navi nemiche, aventi lo stesso una direzione e una velocita.
Segue diagramma UML:

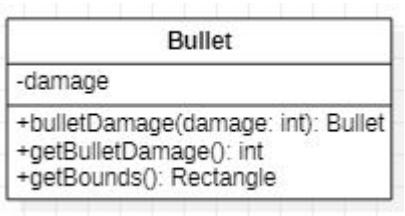

Figure 23: **Bullet.**

3 Sviluppo

3.1 Testing automatizzato

3.1.1 Andrew Gagliotti

TESTING HUD: per testare il comportamento di modellazione della HUD mi sono affidato al supporto di **JUnit** con il quale ho testato nel completo tutti i possibili comportamenti in gioco della stessa:

- **Bonus:** i bonus vengono visualizzati correttamente e per evitare Eccezioni di *input-output*, ho sfruttato il costrutto **try-catch** dei metodi *showBonus()* e *hideBonus()*.
- **Punti:** ho testato i punti in situazioni di normale incremento e decremento e anche in situazioni limite, ovvero che i punti non possono scendere sotto 0 e non possono andare oltre 999.
- **Vite:** ho deciso di testare il comportamento delle vite dimostrando sia situazioni di incremento e decremento normali sia il fatto che non possono essere più di 4 e se scendono 0 a il gameStatus viene settato a false, causando il termine del gioco.

3.1.2 Salvatore Bennici

Prima di passare ai vari dettagli implementativi, vorrei far presente che nel realizzare questa parte del progetto, abbiamo usufruito del tool **JUnit**, come del resto richiesto.

Status

Tenendo bene a mente lo scopo ed il comportamento dei vari *Status*, ho voluto testarli sotto due aspetti:

- **Corretta applicazione dell'effetto:** considerando tutti i vari casi possibili.
- **Corretta gestione del cooldown:** testando tutte le varie casistiche dove si potrebbero creare comportamenti inaspettati.

Task Player

In questo caso ho banalmente voluto testare un aspetto fondamentale:

- **Corretta manifestazione delle task:** testandole sia separatamente attivando l'input ad esse associate, sia mettendo in difficoltà la loro logica di applicazione.

3.2 Metodologia di lavoro

3.2.1 Andrew Gagliotti

Purtroppo, al momento del mio inserimento nel progetto, la task dei proiettili era stata già realizzata, assieme a una piccola parte della logica delle collisioni, dal collega **Andrei Nica**, con il quale ho avuto opportune discussioni di design che hanno visto me e lui collaborare in parte, per un piccolo periodo, sulle scelte di progettazione da adottare. Stessa cosa vale anche per l'altro collega, **Salvatore Bennici**, con il quale ho realizzato una minima parte della componente logica dei menù secondari (in particolare il menù di fine gioco, da poi lui riadattato ad una sua personale architettura) e con il quale mi sono dovuto coordinare per la realizzazione della HUD.

Per compensare quanto riguarda la parte di progetto che era stata già completata, ho deciso di prendermi personalmente il compito di *mantenere* il codice risolvendo quindi eventuali bug, risolvendo problemi di stile Java e adottando scelte di level design dell'applicativo che per me erano funzionali. Volendo fare un paragone personale: sono un amante dei videogiochi arcade e quindi mi sono divertito sia a dare una componente punitiva alle meccaniche di gioco sia a conferire in generale uno stile grafico a metà tra il retrò e il moderno.

Come primo obiettivo da perseguire mi sono dedicato a realizzare la HUD di gioco, lavorando molto sui concetti di estendibilità e portabilità. Dopo aver creato una piccola impostazione di base mi sono poi spostato a lavorare sia sulle collisioni sia sul fixare i comportamenti divergenti delle entità, accordandomi e facendo brainstorming assieme agli altri colleghi. L'ultima parte dello sviluppo da parte mia è stata un continuo fix di bug, adattamento del codice alle regole di Java e creazione di un solido level design.

Per ulteriori commenti rimando alla sezione dei commenti personali.

3.2.2 Salvatore Bennici

Prendendo in considerazione gli obiettivi inizialmente preposti, ho mantenuto la realizzazione della gestione dei bonus/malus (Status) ed i controlli da tastiera, per poi aggiungere la gestione logica delle varie finestre secondarie e lo sviluppo del Ranking System. Quando concerne la modelizzazione ed i movimenti dell'astronave principale, è stata invece realizzata dal collega **Andrei Nica**.

Avendo deciso, durante la fase di analisi, di utilizzare strumenti quali Gradle e JavaFX, ho voluto assurmermi la responsabilità di creare l'opportuno ambiente di progetto. Tuttavia, trovandomi a volte in difficoltà, ho trovato necessaria la collaborazione con i miei colleghi.

Ho inoltre collaborato all'implementazione della difficoltà di gioco, una funzionalità opzionale insieme al collega **Andrei Nica**. Seppur è stata sviluppata come procedurale e non più, come dichiarato inizialmente, su livelli.

Ricapitolando i miei ruoli in questo progetto sono stati:

1. **Bonus/Malus**: creazione e gestione di quest'ultimi. Ho anche creato le icone visibili in gioco.

2. **Input e Task Player**: gestione della maggioranza degli input dell'applicativo e della corretta manifestazione delle task del Player.

3. **Menu**: gestione di controllo relativa alle varie finestre di gioco.

4. **Ranking System**: realizzazione e gestione della classifica locale.

5. **Gradle e JavaFx**: Strutturazione di un corretto ambiente di lavoro, trovando necessaria la collaborazione con entrambi i miei colleghi.

3.2.3 Andrei Nica

I miei obiettivi erano creare un interfaccia per la grafica per i miei compagni che potessero utilizzare per gestire la logica in modo piu efficiente per fare in modo che la gestione della grafica e della logica del gioco siano gestite separatamente. Oltre a questo o modellato le entita di gioco e le loro caratteristiche e credo che sono riuscito ad adempiere ai miei obiettivi.

3.3 Note di sviluppo

3.3.1 Andrew Gagliotti

A livello implementativo ho deciso di adottare varie tecniche a seconda delle casisitiche di problemi che mi sono ritrovato ad affrontare:

- **Cicli e assenza di elementi di programmazione funzionale:** per tutta la mia parte di progetto ho cercato di discostarmi appieno da tutte le cattive abitudini di programmazione funzionale ereditate da altri corsi, cercando di pensare e programmare del codice che fosse veramente orientato agli oggetti.
  Reputo la mia parte di codice ben strutturata e adempiente a tutte quelle soluzioni implementative mostrate a lezione, ma ovviamente non sono abbastanza esperto e maturo da dire ho lavorato completamente bene: ho dovuto sfruttare una soluzione orrenda per la gestione dei bordi del container.
  Sostanzialmente ho sfruttato un campo statico e pubblico che veniva updatato quando entravano in gioco il bonus e malus di velocità, garantendo un corretto funzionamento delle collisioni laterali con il bordo di gioco. Pessima soluzione implementativa.

- **Lambda Expression:** Finché ho potuto, ho cercato di abolire iterazioni che fossero troppo C like o che comunque si adattassero a costrutti ereditati dal C o C++.
  Le varie iterazioni delle parti di model della HUD sfruttano il concetto di Lambda Expression e degli stream funzionali.

- **JavaFX come framework di lavoro:** sfruttare come framework di lavoro **JavaFX** è stata una impresa bella e buona, per commenti riguardo a questa scelta rimando alla sezione dei commenti personali.

3.3.2 Salvatore Bennici

- **Lambda expressions**: dove neccesario, per mantenere il codice il più possibile conciso e leggibile.
- **Stream**: usate anch'esse per rendere il codice conciso e leggibile.
- **Runnable**: utilizzato per gestire gli effetti degli *Status*.
- **ScheduledExecutorService**: utilizzato per alcuni elementi di programmazione parallela necessaria per la gestione dei cooldown degli *Status*.
- **JavaFx e Gradle**: per preparare il progetto, importare librerie e semplificare alcuni aspetti logici.

3.3.3 Andrei Nica

Per avere una linea guida su come scrivere il codice del progetto mi sono ispirato al gameloop pattern come modello principale, dividendo il lavoro in controllo della logica e renderizzazione dei componenti. Ho cercato di rendere le classi create piu concise e brevi possibili. In alcune situazioni ho dovuto fare la scelta difficile se rendere piu leggibile il codice o ottimizare il gioco per renderlo piu fluido. Come la scelta se usare dei thread o meno per la gestione delle navi nemiche. O la gestione della fluidita dei frame, se usare i metodi per le animazioni di JavaFX o crearli manualmente dove o preferito la seconda opzione.Se avessi avuto piu tempo a disposizione avrei cambiato le gestione della direzione, usando al posto di 4 enum, un vettore bidimensionale per gestire la direzione sia verticale che orrizontale.

4 Commenti personali

4.1 Autovalutazione

4.1.1 Andrew Gagliotti

Come ribadito precedentemente nei vari punti della relazione, vado molto fiero del mio *microcosmo* di progetto. Durante la fase di design mi sono sforzato molto per pensare e progettare qualcosa che fosse leggero, compatto, coinciso e riusabile e il risultato mi sembra molto buono. Nonostante non abbia sfruttato molto i *pattern*, reputo comunque funzionali e ben progettate tutte le mie porzioni di codice: reputo sia un concetto di maturità di pensiero ad oggetti riuscire a progettare un software complesso sfruttando molti pattern, maturità che io ho iniziato a coltivare da poco tempo. Sono comunque molto fiero e orgoglioso di ciò che ho fatto e di ciò che sono le mie potenzialità, tant'é che non nascondo la mia volontà nel continuare un percorso del genere e di continuare a lavorare con Java il più possibile, il quale sicuramente non è il linguaggio OO più potente ma sicuramente quello più pratico, dalle grosse potenzialità e semplice a livello applicativo. Ho trovato molto piacevole ideare, progettare e mettere in pratica tutte le mie idee e di collaborare con i miei colleghi, in particolare con **Salvatore Bennici**.
Purtroppo però con il resto del gruppo non è stata una esperienza tanto piacevole a causa della scarsa comunicazione e scarsa organizzazione: molte volte é stato necessario insistere per organizzarsi e alla fine sono anche passato per persona spiacevole. Questo non mi esclude dalle mie colpe, anche io all'inizio ho procastinato come tutti, ma dalla mia parte, mi sono dovuto riprendere da un brutto colpo: (e mi trattengo dal narrare tutto) a causa di una discussione personale con una persona e un suo successivo comportamento malevolo, io ho perso il mio precedente gruppo di lavoro, il quale ha deciso di seguire il gregge e di escludermi. Tutto sommato la mia parte l'ho fatta e mi è piaciuto molto lavorarci, visto che è anche sulla linea d'onda dei miei interessi.

4.1.2 Salvatore Bennici

Ipotizzando che l'obiettivo principale di questo progetto sia quello di farci collaborare in team, non nascondo le difficoltà incontrate a riguardo. Essendo stato il mio primo vero e proprio progetto di gruppo, non sono riuscito a gestire le situazioni sgradevoli che vi sono venute a creare, in particolare la mancanza di comunicazione e la pessima organizzazione. A lungo andare questo non ha fatto altro che distruggere l'atmosfera armoniosa che si era creata inizialmente, tuttavia riconosco che non ho altro che da imparare da questa esperienza. Personalmente cercherò di essere generalmente più affidabile ed anche più diretto circa l'esposizione delle mie idee ed opinioni (mettendole sempre in discussione quando necessario, ovviamente).
Tornando al progetto in sè, la mia principale disavventura è stata la strutturazione efficace del codice. I pochi pattern utilizzati, la non di certo eccellente ottimizzazione e la discutibile architettura del progetto non fanno altro che evidenziare questo problema. Sono perfettamente consapevole di non aver maturato una competenza a riguardo, ma sono pronto a mettermi in discussione per raggiungerla. Mi ritengo piuttosto soddisfatto del mio utilizzo di costrutti avanzati di Java e della realizzazione di una buona leggibilità del codice. Come già detto in precedenza, seppur tortuosa, è stata un'ottima esperienza che auguro non venga negata ai futuri studenti.

4.1.3 Andrei Nica

La mia parte del lavoro e' stata principalmente fornire un interfaccia di collegamento fra la gestione manuale della grafica e il controllo astratto dei componenti del gioco. Avevo gia un po di esperienza in questo campo avendo gia lavorato con OpenGl, e le API di javafx sono state molto di aiuto.Ho cercato di usare la programmazione funzionale dove mi era possibile. Come difetto non ho usato molto i pattern perche mi sembrava che appesantissero inutilmente il gioco.

A Guida utente

Ottenuto il file Jar, eseguirlo da console tramite il comando:
java pathfile.jar -jar
Una volta runnato il gioco ci si troverà davanti al menu principale, da qui basterà scegliere **PLAY** per cominciare a giocare, oppure vedere la *LeaderBoard* o modificare i comandi di gioco di base. Allego schermata del menu principale:

B Esercitazioni di laboratorio

Ecco qui di seguito i link al forum di OOP e GitHub dove sono contenuti i vari esercizi di tutti gli studenti:

B.0.1 Andrew Gagliotti

- **ESERCITAZIONE 04:** https://virtuale.unibo.it/mod/forum/discuss.php?d=62685#p105381
- **ESERCITAZIONE 05:** https://virtuale.unibo.it/mod/forum/discuss.php?d=62684#p105382
- **ESERCITAZIONE 06:** https://virtuale.unibo.it/mod/forum/discuss.php?d=62579#p105384
- **ESERCITAZIONE 07:** https://virtuale.unibo.it/mod/forum/discuss.php?d=62582#p108851
- **ESERCITAZIONE 08:** https://github.com/AndrewGagliottiUnibo/OOP_Laboratorio08
- **ESERCITAZIONE 09:** https://github.com/AndrewGagliottiUnibo/OOP_Laboratorio09
- **ESERCITAZIONE 10:** Non fatta.
- **ESERCITAZIONE Csharp:** Non fatta.