Relazione per
"Programmazione ad Oggetti"
Alone In The Space

Daniele Martignani
Giuseppe Pintus
Cristian Daniel Stratu
Atanas Todorov Atanasov

A.A 2021/2022

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software, mira a creare un videogioco retrò, action game, arcade, ispirato ad astroflux, l'obbiettivò principale del gioco è di sconfiggere il maggior numero possibile dei nemici, con difficoltà della partica crescente. Il software richiesto dovrà essere facilmente espandibile a futuri aggiornamenti ed inoltre presentare un design semplice ed intuibile.

Requisiti funzionali

- Gestione nave spaziale da input
- Classifica risultati con i rispettivi utenti
- Implementazione algoritmi per le navi nemiche
- Menu principale, opzioni
- Creazione interfaccia grafica
- Menu controlli

Requisiti non funzionali

- Possibilità di mettere pausa
- Gestione inventario e potenziamento arma
- Wiki dei nemici
- Gestione armi secondarie

- Barra della vita di ogni nave
- Effetti sonori
- Animazioni

1.2 Analisi e modello del dominio

Il videogioco è composto principalmente da entità che dovranno interagire fra di loro allo scopo di eliminarsi a vicenda. Le entità principali saranno i nemici, il player, e i proiettili.

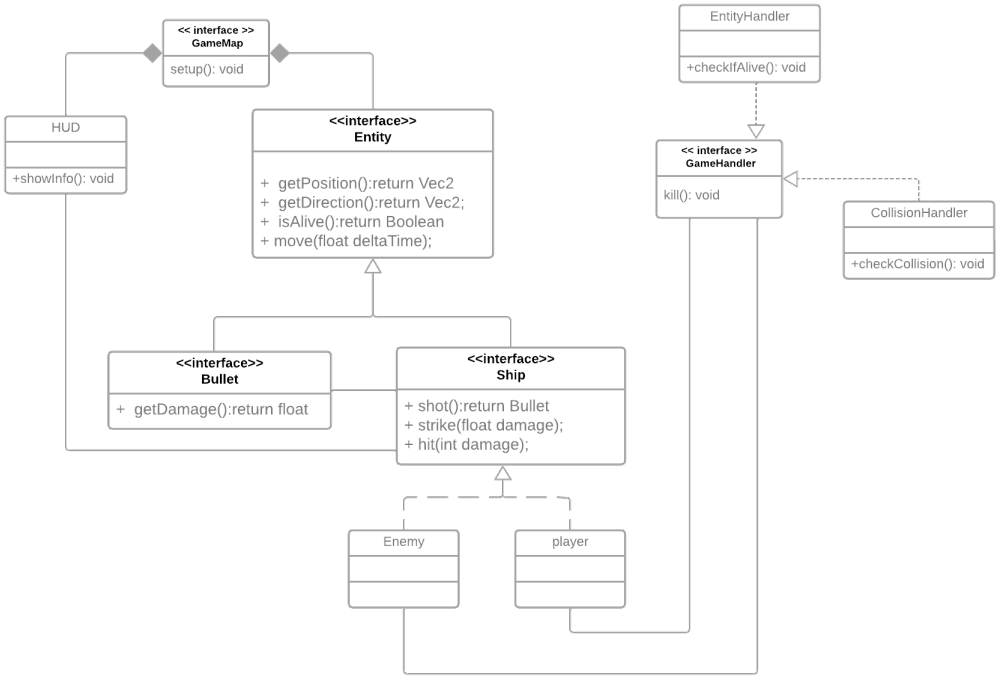

Figura 1.1: Schema UML Analisi di dominio

Capitolo 2

Design

2.1 Architettura

Il progetto si propone di seguire le linee guida del pattern MVC, creando 3 componenti che siano in comunicazione fra di loro in maniera costante. La divisione in 3 componenti permette alla componente visiva di occuparsi della rappresentazione grafica, alla componente di modellazione di realizzare le tecniche di risoluzione dei problemi preposti e al controllore di far comunicare le due parti tra di loro, garantendo così il corretto funzionamento del programma. In particolare il controller sarà diviso in componenti che dovranno occuparsi di:

- **Controllare il corretto funzionamento delle varie schermate di gioco**: il gioco è composto da varie schermate, le quali si dovranno alternare senza creare problemi tra di esse e senza danneggiare l'applicazione.
- **Controllare la corretta successione degli eventi di gioco**: è importante avere una logica di gioco, che segua certe regole di funzionamento del gioco e gestisca quando il gioco dovrà terminare.
- **Controllare il funzionamento delle varie entità**: le varie entità esistono contemporaneamente in tutto il campo di gioco, perciò è opportuno monitorarle e dare loro una logica di funzionamento, gestendo la loro creazione ed eliminazione continuamente seguendo un processo logico.
- **Controllare gli input di gioco**: è necessario stabilire a priori i comandi e la rispettiva risposta all'interno del gioco.

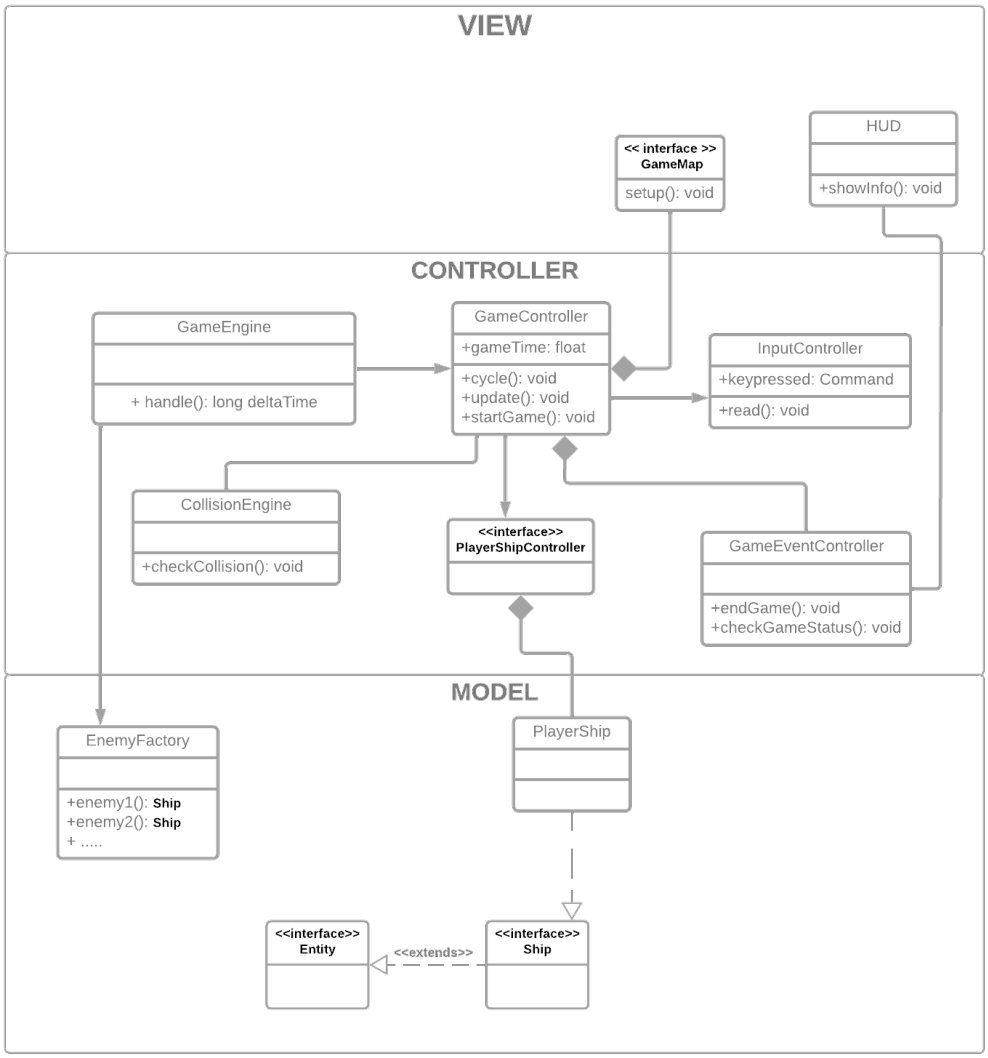

Figura 2.1: Schema UML - Architettura MVC

2.2 Design dettagliato

2.2.1 Daniele Martignani

HUD

Nel creare l'HUD ho ragionato su quello che era il suo scopo: l'interfaccia grafica deve comunicare all'utente, in modo preciso e minimale, ed ecco perchè ho scelto un design minimale che rispettasse le linee del pattern MVC. La sua portabilità è considerevole, infatti basta che un progetto JavaFX dichiari un'istanza del controller per poterne usufruire(con piccole modifiche sui nodi e sulle variabili): questa parte infatti si collega al progetto solo con la classe GameMap(view) e con la classe GameController(controller). L'HUD è organizzato in 3 sotto-classi che vengono inizializzate dalla view:

- **PowerUp HUD**: Rappresenta a schermo l'icona del power-up utilizzabile giocatore e lo fa ottenendo dal GameMap il powerUp. Quando il powerUp sarà disponibile, verrà mostrato a schermo fino a quando non verrà utilizzato. Come ulteriore controllo per la fase di test, ho implementato il controllo se il powerUp è attivo o no, con una struttura dati esterna. Il powerUp è modellato con un ImageView. Diagramma UML della parte appena descritta:

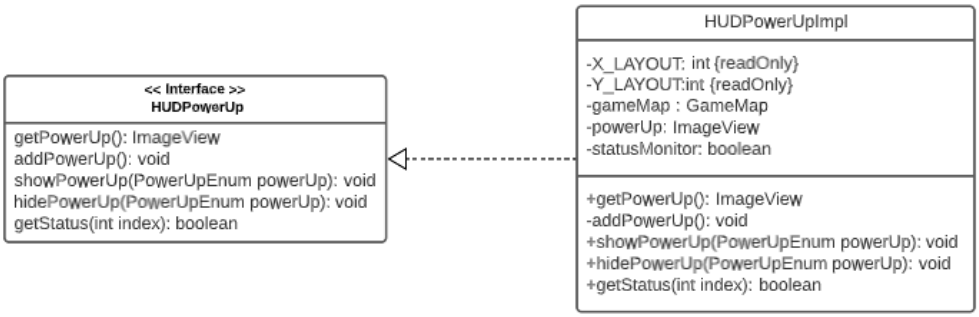

Figura 2.2: Schema UML - HUDPowerUp

- **Life HUD**: A livello grafico, la vita è rappresentata da una **Label** che si posiziona in alto a destra nella schermata di gioco. Ogni volta che la vita del giocatore viene modificata la stringa di testo contenuta nella **Label** viene modificata di conseguenza. JavaFX considera la **Label** come un vero e proprio nodo, perciò ho dovuto dichiarare all'interno del EventController come si dovesse comportare con tutto il sistema di nodi. Se un giocatore perde tutti i punti vita allora il gioco termina e si avvisa la classe GameEventController di dover terminare il *game cycle* e di lanciare la GUI di di fine gioco. Diagramma UML della parte appena descritta:

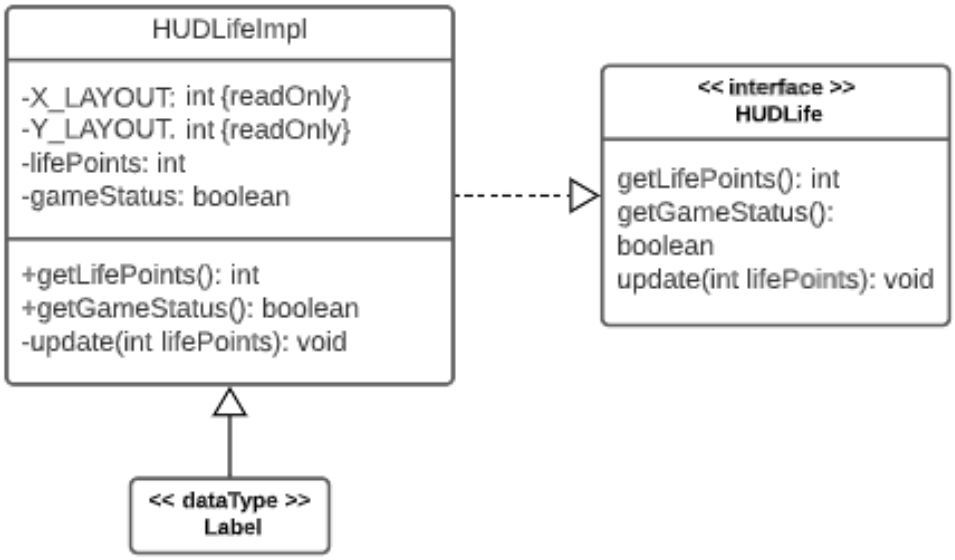

Figura 2.3: Schema UML - HUDLife

- **Points HUD**: similmente a quella che è stata la realizzazione della componente visiva della vita, ho deciso di utilizzare anche qui una **Label**. Ogni volta che il giocatore guadagna punti la stringa di testo all'interno della **Label** viene aggiornata. Diagramma UML della parte appena descritta:

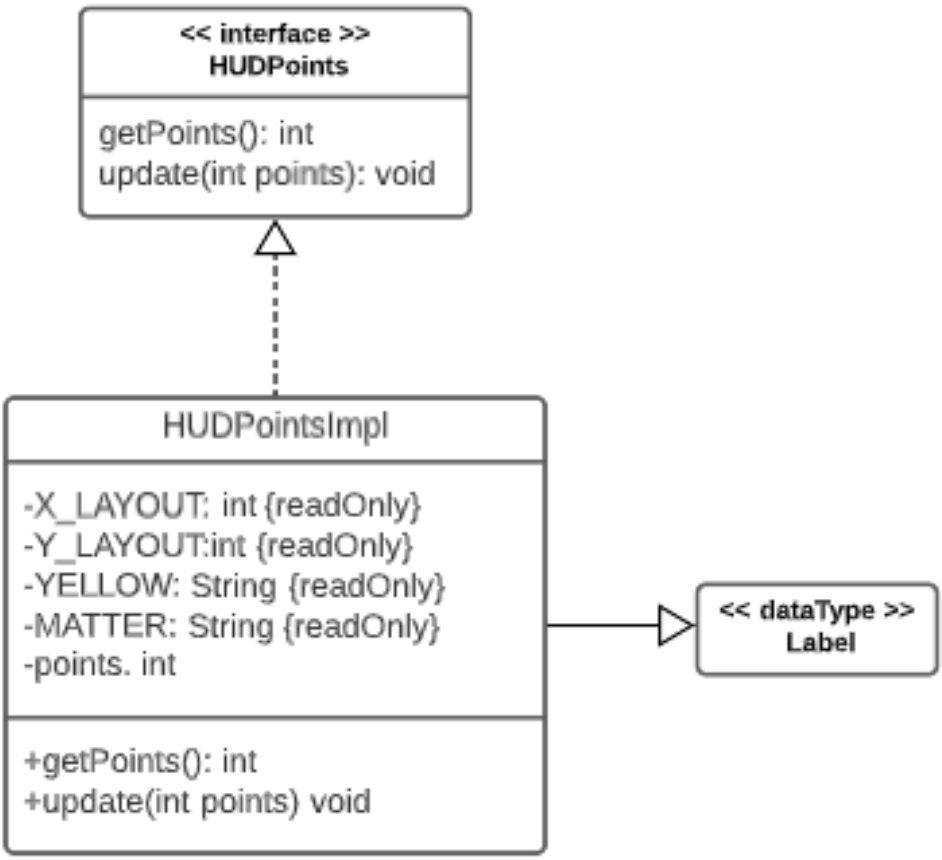

Figura 2.4: Schema UML - HUDPoints

Eventi di gioco

Come già detto, l'HUD segue il pattern MVC, perciò a una parte di model corrisponde una parte di view che si occupa di mostrare e istanziare le componenti a schermo e una controparte di controller che si occupa di gestire i vari elementi. La parte di view:

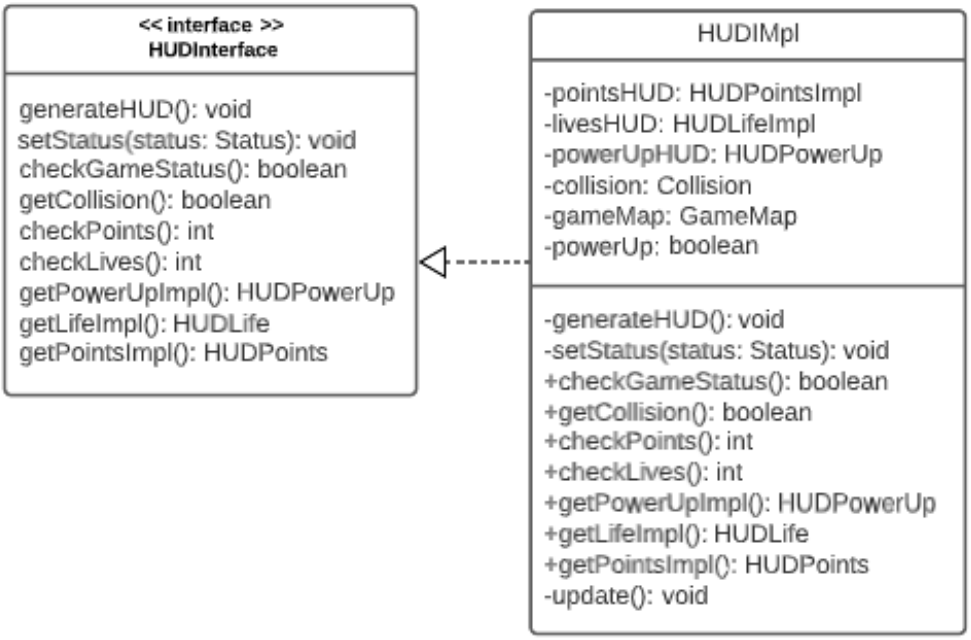

Figura 2.5: Design di dettaglio della view dell'HUD.

Siccome la parte delle collisioni incidiono sulla parte di HUD è stato creato un controllore, il quale mette in comunicazione le componenti e garantisce il corretto funzionamento del gioco. Il **GameEventController** si occupa di decidere come e quando terminare il gioco basandosi sulla condizione delle vite del giocatore. Schema UML della parte appena descritta:

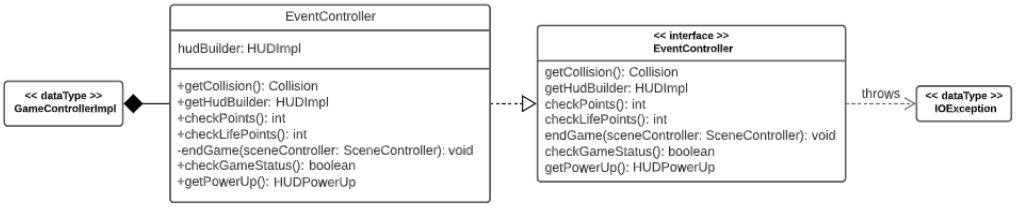

Figura 2.6: Design di dettaglio della componente controller.

Menù di gioco

Per la creazione dei menù di gioco, ho utilizzato il tool SceneBuilder. Ho deciso di creare diversi sotto menù, perciò necessitando di una gestione dello scambio tra le varie finestre di gioco, ne ho definito la logica nell'interfaccia SceneController.

Realizzazione e gestione delle finestre

Per fare in modo che il contenuto venga sostituito senza dover creare una nuova finestra ogni volta, ho sfruttato JavaFX, sostituendo la *Scene* contenuta nello Stage principale dell'applicativo. Ho creato le rispettive *Scene* a partire da file FXML e successivamente ho affiancato i relativi controller. Ho realizzato 5 menù:

- **Main**: È il menu principale, visualizzato all'avvio. Il giocatore può scegliere se navigare negli altri menù o iniziare la partita.
- **Nickname**: Finestra che precede l'avvio del gioco, richiede l'inserimento del nickname da parte dell'utente.
- **Scores**: Menù che mostra la classifica dei migliori giocatori.
- **Controls**: Menù che permette di modificare i comandi da utilizzare in gioco.
- **EndGame**: Schermata di fine gioco, mostra le informazioni relative al gioco e da la possibilità di tornare al Menu principale.

Controllers dei Menù

Dopo aver creato le classi di controllo per ciascuna *Scene*, ho deciso di creare una classe astratta, *BasicFXMLController*. Questa classe verrà estesa da tutti i controller, questa scelta è dovuto principalmente alla presenza di funzionalità e campi comuni.

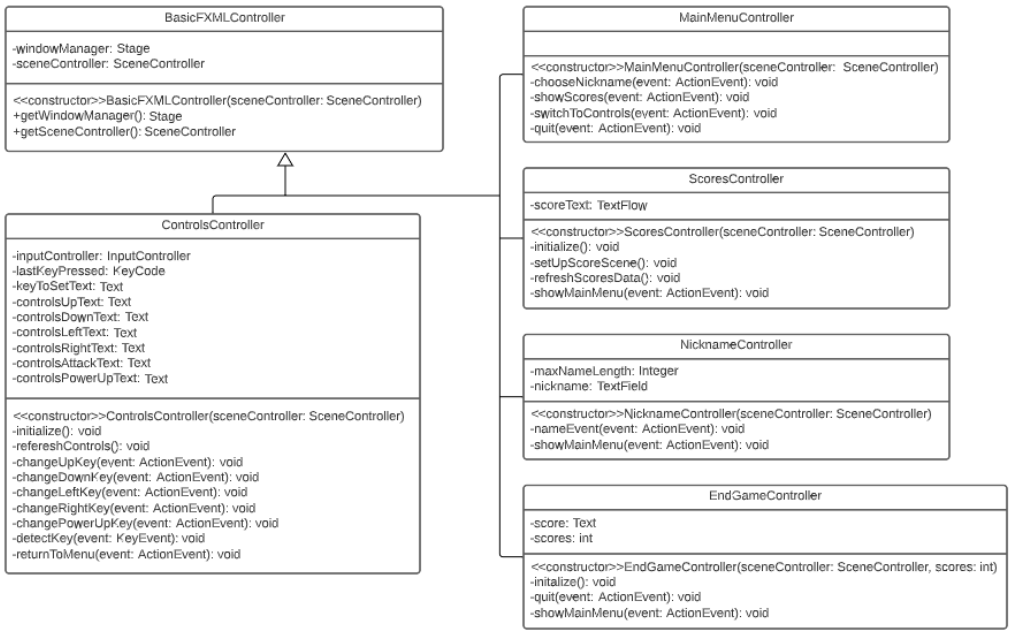

Figura 2.7: Gerarchia controller delle finestre di gioco.

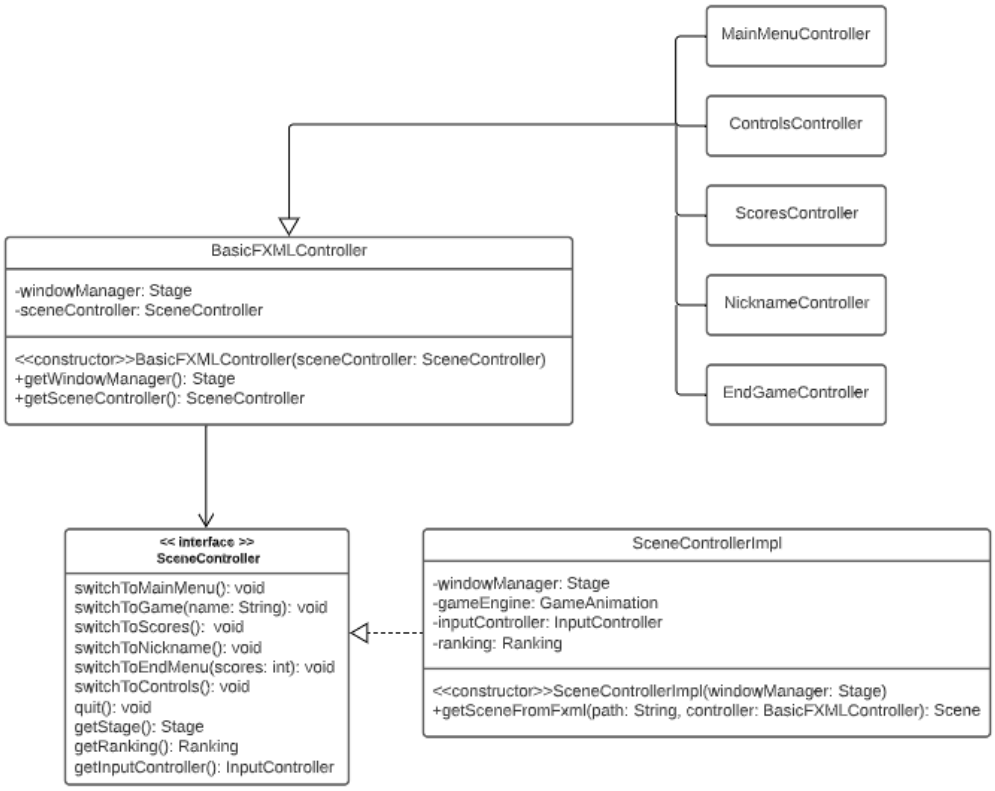

Figura 2.8: Architettura di gestione delle finestre di gioco.

Gestione degli Input

InputController tiene traccia dei tasti premuti dall'utente per poi elaborarli. Ad ogni tasto viene associata una singola azione, tuttavia possono verificarsi combinazioni di input che annullino il comportamento aspettato. È possibile associare tasti personalizzati alle varie azioni eseguibili dal player, tramite il menù Controls.

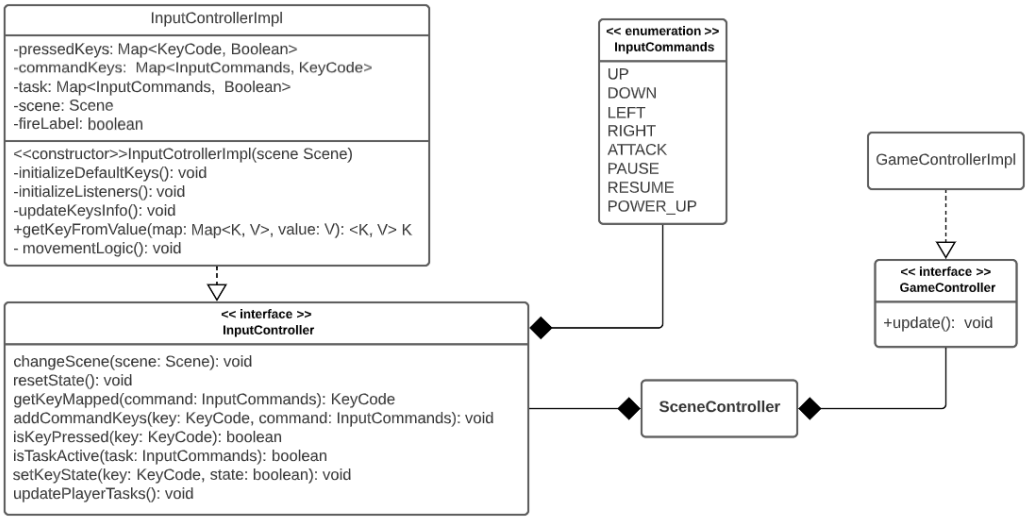

Figura 2.9: Gestione degli input.

FrameManager

Ho creato questa classe per il render di ogni frame di gioco. Prende le posizioni di ogni oggetto del gioco e in base al deltaTime e alla direzione aggiorna la posizione dell'oggetto.

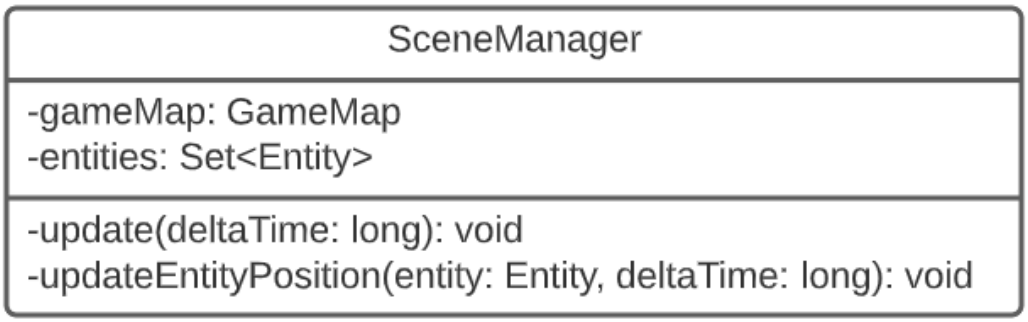

Figura 2.10: Scene Manager.

GameMap

Ho creato una classe che contiene tutte le informazioni riguardanti la logica del gioco, come il giocatore, i nemici attivi, la posizione e la velocità di tutte le entità.

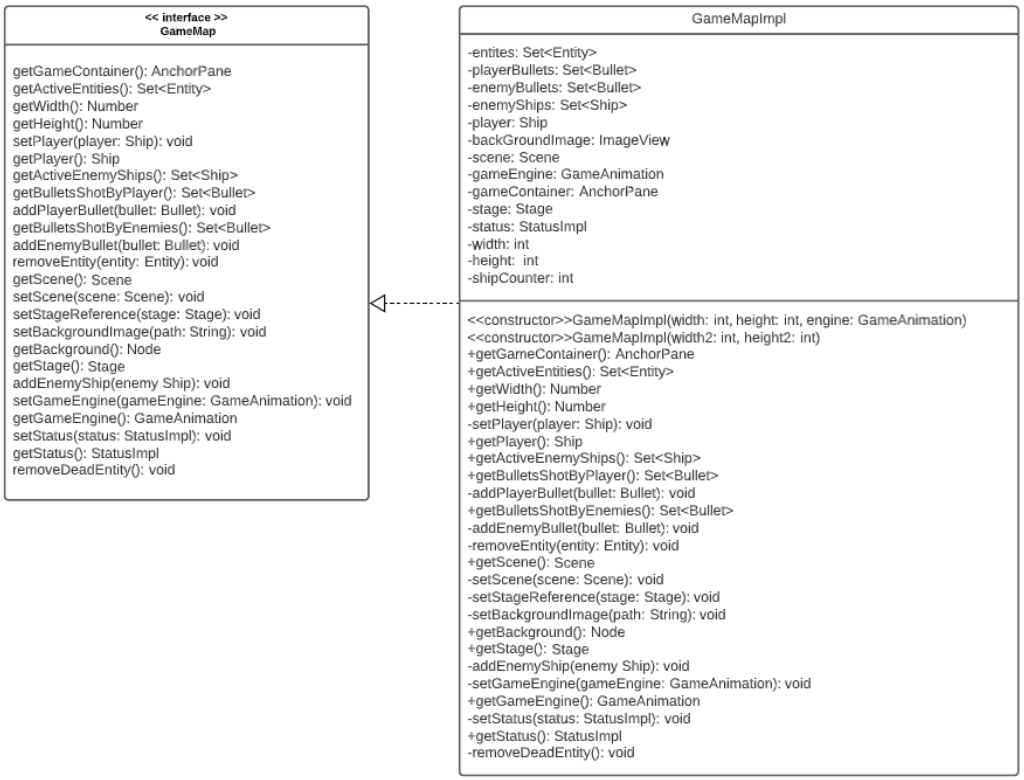

Figura 2.11: Game Map.

2.2.2 Cristian-Daniel Stratu

La mia parte di progettazione svolta riguarda:

- Player
- PowerUp

Player

Un elemento principale d'ogni gioco, il giocatore si basa sul movimento in 2D di una spaceship, normalmente fatta con controlli a tank (Left and Right gira, Up and Down muove). Il movimento dello giocatore viene realizzata in un modo più semplice, muovendosi in base al tic update dentro **GameEngine** con una direction sempre in fronte al giocatore.

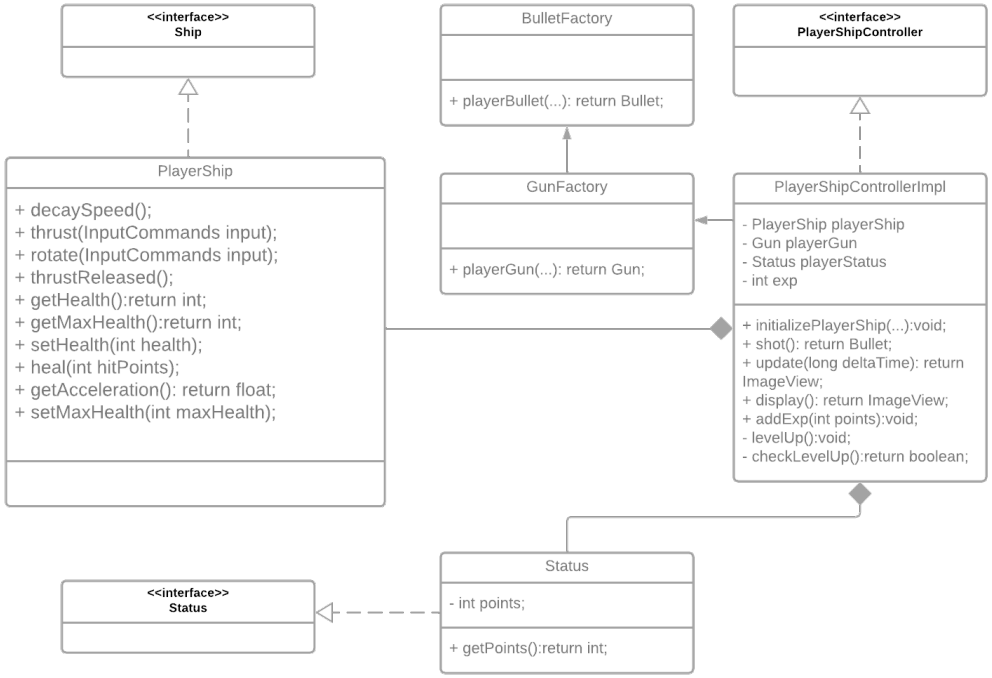

Figura 2.12: Schema UML PlayerShip e PlayerShipController

Il Player si muove in base al input dello giocatore dato tramite **PlayerShipController**. Mentre il Player uccide dei nemici, lui prenderà punti per il score ed esperienza che aumenta il livello dello giocatore.

Problema In che modo si muove il Player?

Soluzione In base alla accelerazione e il tic update che e' il nostro tempo si trova la nuova velocità (in modo che sia minore di quella massima) che aggiorna la posizione in base all'angolo in cui e' girata la nave. La rotazione viene aggiornata automaticamente con l'aggiornamento della posizione e dello sprite dentro **PlayerShipController**. Questo gestisce anche la posizione dello sprite sulla **GameMap**, mettendolo a pari con il **PlayerShip**.

Problema Come viene il aumento di livello dello giocatore?

Soluzione Si ottengono i punti fatti che vengono poi messi come esperienza in **PlayerShipController**. Si fa una verifica in base al livello corrente e la esperienza necessaria, dopo si fa un level up che cambia vari statistiche

dello giocatore come la vita massima, la velocità con cui si può' sparare e il danneggio che il giocatore fa.

Problema Come viene limitato il fire rate dello Player?

Soluzione Una volta chiamato il metodo per fare sparare un proiettile, il Controller disabilita l'abilita di sparare in più finché un certo tempo passa tramite tic update. Il metodo update tiene traccia se il giocatore ha sparato e, se vero, quanto tempo ha passato dall'ultimo tic update. Questo e' abbastanza utile per la possibilità d'espansione su più tipi di armi. In questa sezione si possono approfondire alcuni elementi del design con maggior dettaglio.

PowerUp

Il potenziamento verrà dato una volta che un certo numero di punteggio viene raggiunto dal giocatore. Questo farà comparire un'icona. Da quel momento in poi lo può usare premendo il bottone per l'attivazione dello PowerUp che aumenta il danno e il fire rate del PlayerGun.

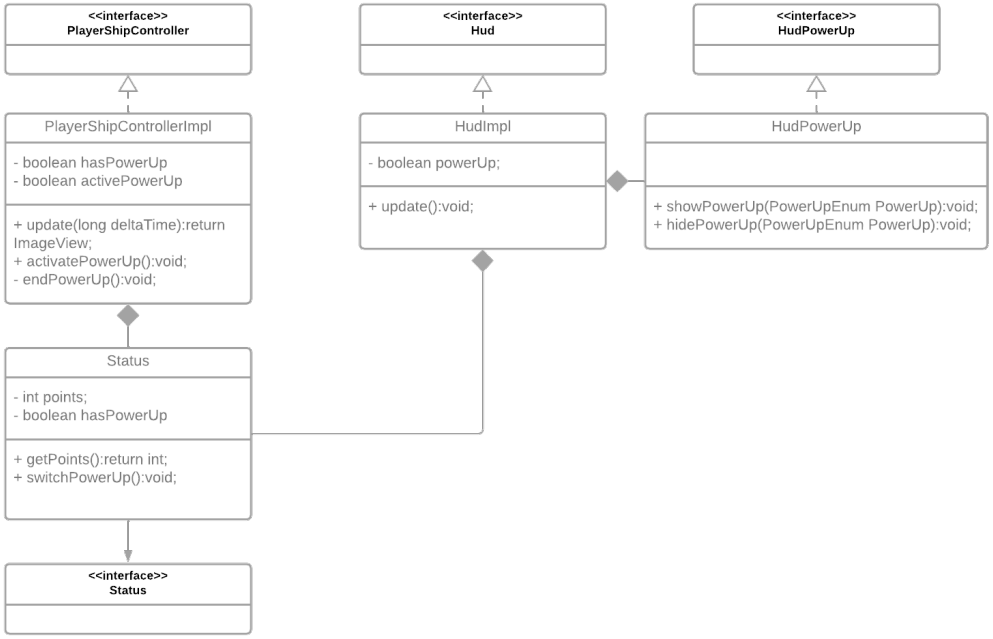

Figura 2.13: Schema UML PlayerShipController e PowerUp

Problema Come si sincronizza il PowerUp?

Soluzione Si tiene presente un valore per il PowerUp in **Status** che viene aggiornato ogni volta che viene ottenuto il punteggio necessario per ricevere il Power Up almeno che l'abbia già. Si può' ottenere il PowerUp mentre l'ultimo PowerUp e' ancora attivo. Il **PlayerShipController** aggiorna in quale modalità' si trova il giocatore aggiornando poi Status. L'HUD guarda dentro il Status in quale stato si trova il giocatore e manda i dati al **HudPowerUp**.

Problema Come si mantiene il PowerUp Mode con il sistemo di Level Up?

Soluzione Semplicemente il Controller tiene aggiornato un **PlayerGun** suo, sostituendo il **Gun** dentro giocatore quando necessario (o per PowerUp, o per potenziamento per livello).

2.2.3 Atanasov Atanas Todorov

Asteroidi

Per la modellizzazione degli asteroidi ho realizzato un'interfaccia Asteroid contenente i metodi principali per la gestione dei vari attributi come: posizione, danno e esplosione(animazione). Gli asteroidi si distruggono se entrano in contatto con la nave del giocatore. Ho suddiviso poi gli asteroidi in due tipi:

- BasicAsteroid
- StrongerAsteroid

BasicAsteroid

Questo tipo di asteroide ha un attributo health che rappresenta la vita dell'asteroide. Ogni volta che un bullet del giocatore oppure dei nemici va a colpire il BasicAsteroid, health viene decrementato a seconda del danno inflitto dal bullet. L'asteroide si distruggerà in seguito all'azzeramento del attributo health.

StrongerAsteroid

Questo tipo di asteroide è immune ai bullet del giocatore e dei nemici, di conseguenza non dispone di un attributo health.

Animazioni

Per le animazioni ho creato un oggetto Explosion la cui unica funzione è di restituire l'immagine dell'animazione associata all'oggetto. L'animazione in se poi viene trattata da JavaFX utilizzando FadeTransition che appartiene al package Animation. Attraverso FadeTransition è possibile manipolare l'opacità di un'immagine.

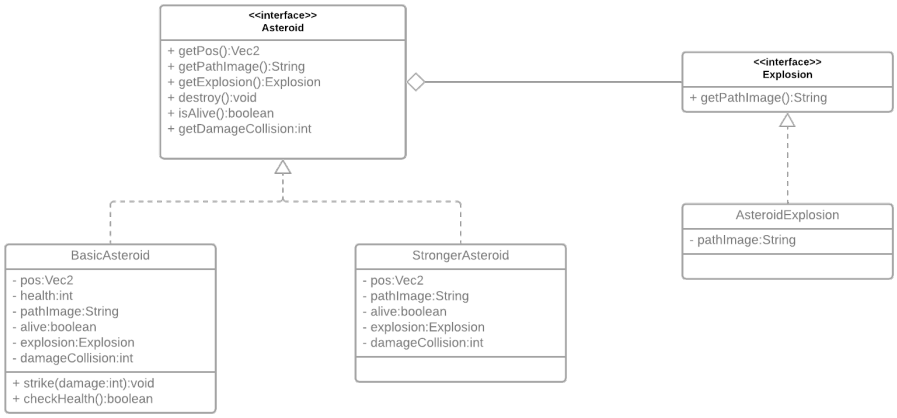

Figura 2.14: Schema UML Asteroid

Gestione delle collisioni

Mi sono occupato della gestione delle collisioni, uno degli aspetti importanti ai fini dello svolgimento della partita. Ho definito due metodi che verificano l'avvenuta collisione tra:

- La nave del giocatore ed una nave nemica
- La nave del giocatore e gli asteroidi

Sulla base di questi due si appoggiano i metodi di controllo che vengono invocati periodicamente per l'aggiornamento delle statistiche a seconda dei risultati ottenuti. Ho definito dei metodi per verificare l'avvenuta collisione tra la nave del giocatore e un asteroide, oppure tra un proiettile ed un asteroide. Non ho gestito la collisione tra nave nemica e asteroide perche altrimenti avrebbe semplificato la difficoltà del gioco.

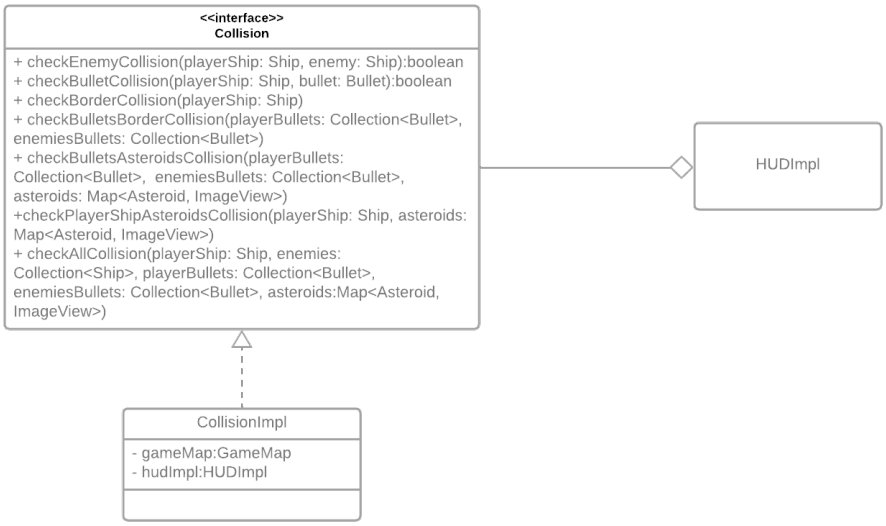

Figura 2.15: Schema UML Collision

Game Loop

Per lo sviluppo del game loop, ho scelto di affidarmi alla classe astratta **AnimationTimer** di **JavaFX** che si basa sui metodi start() (necessario per l'avvio dell'animazione) e stop() (per l'interruzione dell'animazione). Ho realizzato una classe **GameAnimation** che estende AnimationTimer, ridefinendo così il comportamento del metodo handle(long now) che verrà eseguito ad ogni frame.

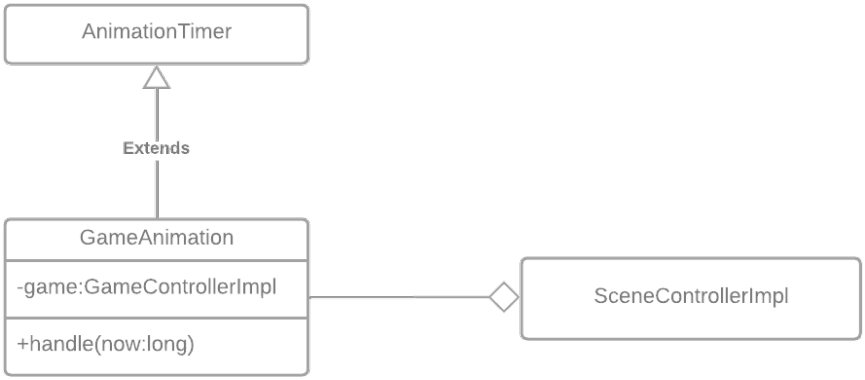

Figura 2.16: Schema UML Game Engine Loop

2.2.4 Giuseppe Pintus

AI nemici

Per l'AI dei nemici, il loro movimento è semplice, si girano verso il nemico e sparano soltanto se è in quella direzione, giustamente ogni tipo di nemico con caratteristiche leggermente diverse.

Gestione dei nemici

Per la gestione dei nemici e dei bullet, ho deciso di inserire un classe intermedia per avere il riuso della classe bullet non definitivamente legata alla classe ship, inserendo quindi la classe gun è possibile definire diverse combinazioni con cui una nave potrebbe attaccare. per la creazione dei nemici, ho deciso di optare per delle classi astratte, secondo il pattern Template e strategy, ridefininendole in delle factory, così è possibile sfruttare nemici con personalizzazione differenti senza dover ri-implementare del tutto la classe derivata da ship.

Sistema di Ranking

Abbiamo deciso di implementare una classifica locale dei punteggi, dove verranno visualizzati i nomi dei giocatori con i relativi punteggi. L'utente nel menù Scores potrà vedere i punteggi dei migliori giocatori e confrontarli. I nomi e i punteggi vengono salvati in una Map, questa viene elaborata per essere salvata su un file locale. Per ogni giocatore verrà salvato soltanto il punteggio più alto.

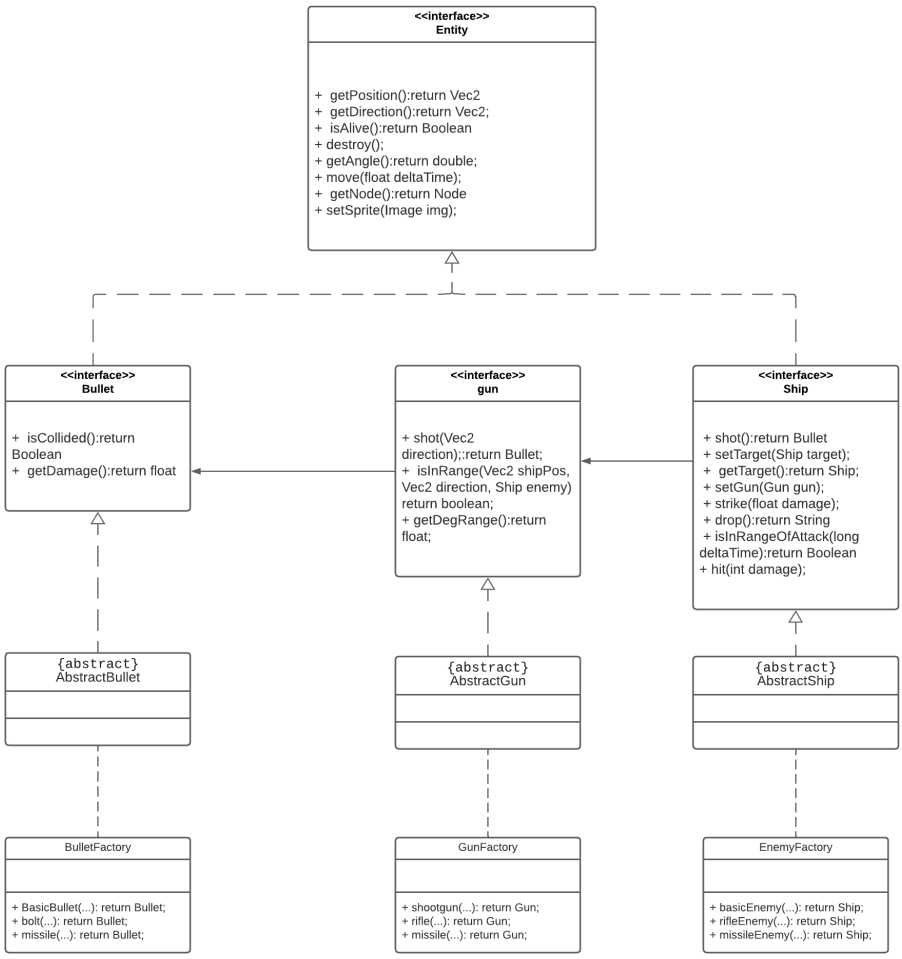

Figura 2.17: Schema UML - Nemici

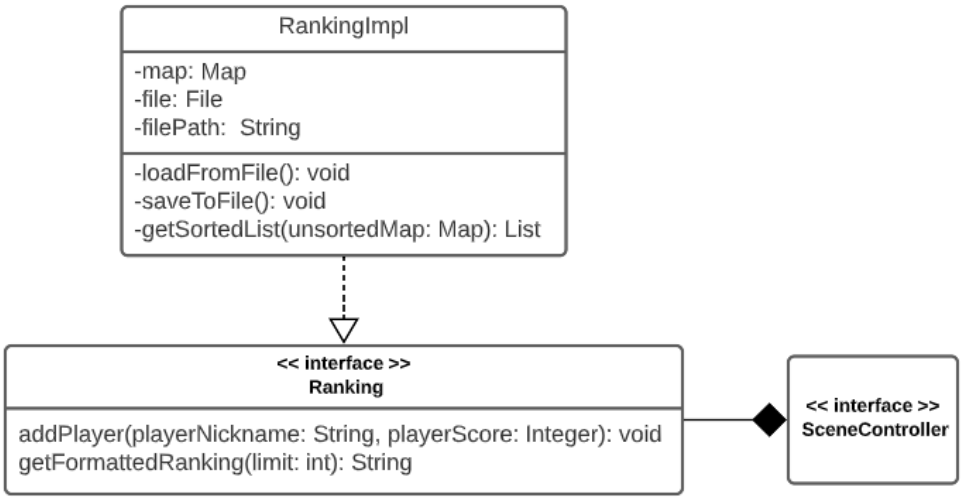

Figura 2.18: Struttura del sistema di ranking.

Capitolo 3

Sviluppo

3.1 Testing Automatizzati

- CollisionTest: Ho deciso di effettuare dei test che riguardassero i due metodi principali delle Collisioni per accertare che avvenga effettivamente la rilevazione. In tal modo, prima di effettuare l'aggiornamento delle statistiche, ho verificato che tutto funzionasse correttamente.
- PlayerTest: test del movimento, della vita e se può sparare
- HUDTest: test su tutti i comportamenti in gioco dell'HUD.
  - **PowerUp**: il powerUp viene visualizzato correttamente.
  - **Punti**: ho testato l'incremento dei punti.
  - **Vita**: ho testato l'incremento, il decremento dei punti vita e se i punti vita scendono a 0 il *gameStatus* viene settato a false, causando la fine del gioco.
- InputControllerTest: Test della corretta manifestazione delle task, testandole sia separatamente sia mettendole in conflitto attivando gli input ad esse associate.
- EnemyTest: Testing del movimento e traiettoria di sparo.

3.2 Metodologia di lavoro

3.2.1 Daniele Martignani

Avendo deciso, durante la fase di analisi, di utilizzare strumenti quali Gradle e JavaFX, ho voluto assumermi la responsabilità di creare l'opportuno am-

biente di progetto. Tuttavia durante il processo mi sono scontrato con alcune difficoltà e la collaborazione con i miei compagni è stata necessaria. I miei ruoli in questo progetto sono stati:

- **HUD**: creazione e gestione di quest'ultimi.
- **Input**: gestione degli input dell'applicativo.
- **Menu**: gestione e controllo delle varie finestre di gioco.
- **GameMap**: creazione e gestione del campo di gioco.

3.2.2 Giuseppe Pintus

I miei ruoli in questo progetto sono stati:

- ship: modello del nemico generalizzato
- bullet: modello del proiettile generalizzato
- gun: modello delle armi
- factory: creazione dei nemici, delle armi e dei proiettili

3.2.3 Cristian-Daniel Stratu

Ho svolto più lavoro sulla parte di model in **MVC** cercando sempre di semplificare per il benessere mio e di miei compagni nel gruppo e poi aiutare dalle altre parti dov'era necessario, come aggiungendo delle idee o aiutare risolvere errori. Con le mie implementazione ho cercato sempre di realizzare tutto un po' staccato da uno e l'altro per dare via alla fluidità di più classifiche come più tipi di **PlayerShip**, più tipi di **Gun** per il **Player**, forse anche la possibilità' di più giocatori. Anche con **PlayerShipController** ho cercato di semplificare un po' tutto in modo che si deve fare un solo call di metodo per tenere aggiornato lo sprite.

Riguardando la parte di **DVCS** e' stato abbastanza bene lavorare insieme, avendo ognuno il suo branch per lavorarci dentro ed esperimentare. I vari merge conflict non sono stati complicati.

3.2.4 Atanasov Atanas Todorov

Creazione degli asteroidi all'interno della mappa. Aggiunta delle animazioni in seguito ad una collisione tra nave del giocatore e asteroide. Gestione della barra della vita di ogni nave presente nella mappa. Gestione delle varie collisioni possibili all'interno della partita e conseguente aggiornamento delle statistiche di gioco. Implementazione del game loop attraverso l'utilizzo della classe Abstract **AnimationTimer**.

3.3 Note di sviluppo

3.3.1 Daniele Martignani

- Lambda Expression
- JavaFX
- Stream

3.3.2 Giuseppe Pintus

- Lambda
- JavaFX

3.3.3 Cristian-Daniel Stratu

- Lambda
- JavaFX
- Optional

3.3.4 Atanasov Atanas Todorov

- Lambda
- JavaFX
- Stream
- Uso di librerie esterne: AnimationTimer

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Daniele Martignani

All'inizio dello sviluppo del progetto non avevo idea di come si sarebbe concluso il progetto, perché avevo diversi dubbi su come progettare ed implementare alcune parti. Complessivamente sono molto contento del risultato finale, raggiunto grazie all'impegno costante di tutti i membri del gruppo. Le conoscenze acquisite durante il corso sono state indispensabili durante lo sviluppo del gioco. Non avevo mai usato JavaFX e FXML prima, quindi ho dovuto guardare diverse guide su come utilizzarli, oltre a studiarne la documentazione.

4.1.2 Cristian-Daniel Stratu

Mark Twain "If I Had More Time, I Would Have Written a Shorter Letter"

Questo progetto mi ha fatto vedere di nuovo le parte belle e le parte meno belle della programmazione in generale. Penso di aver fatto un lavoro soddisfacente ma se c'era più tempo alla mia disposizione si poteva migliorare abbastanza tutte le mie parti, ma se non era una deadline avrei fatto di tantissimi delay che il progetto non avrebbe mai visto un realizzo. In tutto mi se dato un nuovo gusto alla programmazione che vorrei continuare ad esplorare nel futuro, specialmente nel mondo dei video giochi.

4.1.3 Atanasov Atanas Todorov

Questo progetto mi ha permesso di ampliare le mie conoscenze riguardanti la programmazione ad oggetti, soprattutto grazie anche alla collaborazione con i componenti del mio gruppo. Sono contento di aver svolto questa esperienza di gruppo perché la ritengo utile in un'ottica futura, dove in questo settore è fondamentale il lavoro di squadra. Non sono soddisfatto al 100% del lavoro da me svolto perché avrei potuto migliorare alcune cose ma sicuramente mi servirà da lezione per le prossime volte.

4.1.4 Giuseppe Pintus

All'inizio del progetto ero molto titubante perché essendo tutti studenti senza un leader con esperienza, pensai che sarebbe andato male il progetto, però mi son ricreduto che nonostante tutto siamo riusciti a realizza una bella applicazione, migliore delle mie aspettative. In conclusione, interagendo con i miei compagni ho avuto l'occasione di approfondire e imparare ed approfondire le librerie di JavaFX.

Appendice A

Guida utente

All'avvio del appare il menu principale dove può cominciare a giocare, riguardare lo score precedenti e un menu per cambiare i controlli. I controlli di default sono:

- W - Avanti
- S - Indietro
- A - Gira sinistra
- D - Gira destra
- K - Spara
- L - Usa potenziamento

Dopo aver premuto il play e aver scelto un nickname il gioco comincia.

Figura A.1: Start Up screen

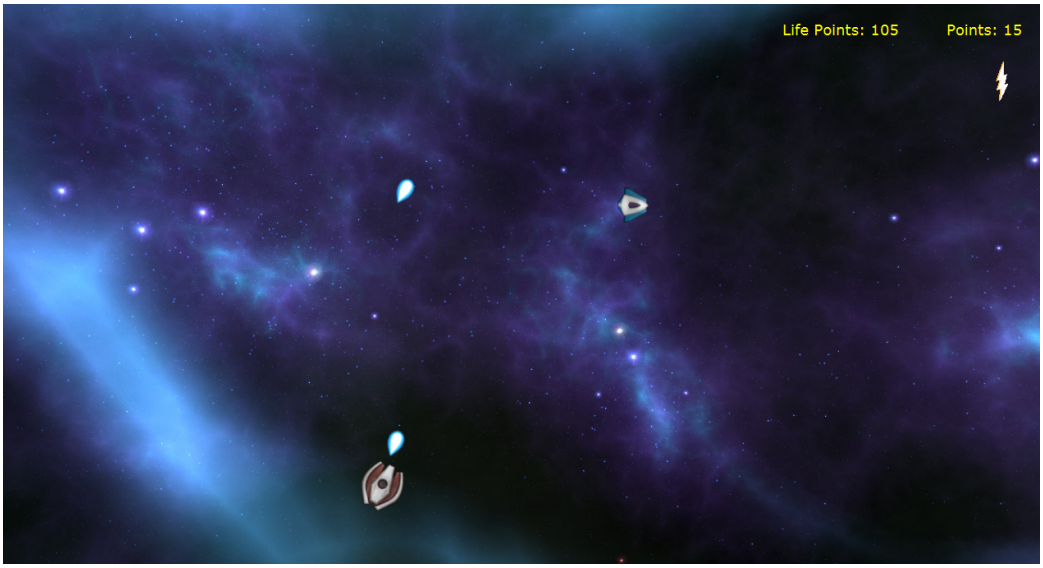

Figura A.2: Gameplay screen

L'obbiettivo e' di raggiungere il punteggio più alto possibile. Mentre il gioca continua il spawn rate degli nemici comincia a crescere, aumentando la difficoltà. Nello stesso tempo più punteggio il giocatore fare, più livelli avrà. Ogni livello aumenta la vita massima di 5%, ogni 3 livelli aumenta il fire rate (sempre di 5%) a cui può sparare ed ad ogni 5 livelli aumenta il danno del Player Gun. Oltre a questo, ogni 15 punti il giocatore può ricevere un PowerUp. Attivando il PowerUp raddoppia il fire rate e danno.

Una volta morto il score verrà ricordato in un file che si vede sotto il menu SCORES.

Appendice B

Esercitazioni di laboratorio

Appendice C

Parte Extra

C.0.1 Daniele Martignani

Classi:

- GameMap
- WindowManager
- HUDInterface, HUDImpl, HUDLife, HUDPoints, HUDPowerUp
- SceneManager
- InputController
- BasicFXMLController, ControlsController, EndGameController, MainMenuController, NicknameController, SceneController, SceneControllerImpl, ScoresController
- EventController
- sorgenti .fxml

Per quanto riguarda i controller delle GUI ho consultato il seguente progetto come riferimento:
`https://bitbucket.org/danysk/oop19-barzi-eddie-belloni-sofia-rossi-davide-vissar`
`src/master/`.
Ho consultato questo progetto perché avevo letto nelle regole che si potevano utilizzare come riferimento i progetti presenti nella repository presente in fondo alle regole. Ho utilizzato questo progetto come riferimento soltanto per quanto riguarda la gestione della GUI e dei suoi controller.

C.0.2 Cristian-Daniel Stratu

Classi:

- PlayerShip
- PlayerBullet (dentro BulletFactory)
- PlayerGun (dentro GunFactory)
- PlayerShipController
- Status
- Una parte di HudPowerUp per potenziamento
- PlayerValues
- PlayerGunValues
- Parti di GameController riguardando il PlayerShip
- Una parte per GameAnimation per implementare weighted random enemy spawner

Alcune operazioni per calcolare il movimento hanno usato come riferimento la classe Vector2 della libreria libGDX (`https://github.com/libgdx/libgdx/blob/master/gdx/src/com/badlogic/gdx/math/Vector2.java`) per il fatto di aver usato come libreria iniziale libGDX prima di cambiare per la nuova libreria Vec2 di FXGL.
Altri riferimenti erano vari tutorial su questo tipo di gioco per capire la parte di implementazione di Player (in vari linguaggi come Ruby, C# dentro Unity).

C.0.3 Atanasov Atanas Todorov

Classi:

- Collision, CollisionImpl
- GameAnimation
- Asteroid, BasicAsteroid, StrongerAsteroid, AsteroidFactory
- Explosion, AsteroidExplosion
- AsteroidValues

- Una parte di GameMapImpl per la gestione grafica degli asteroidi, delle animazioni e delle barre di vita

Ho consultato i seguenti link per la creazione del GameAnimation:
`https://edencoding.com/animation-timer-speed/`
`https://docs.oracle.com/javase/8/javafx/api/javafx/animation/AnimationTimer.html`

C.0.4 Giuseppe Pintus

Classi:

- Entity
- Bullet, BulletFactory, AbstractBullet
- Gun, GunFactory, AbstractGun
- Ship, AbstractShip
- EnemyFactory
- Ranking, RankingImpl

C.0.5 Classi create insieme:

- GameController, GameControllerImpl
- Launcher, Main
- Tutte le classi contenute in utilities

C.0.6 Storia della repository

Abbiamo creato un progetto iniziale vuoto, poi ogni componente ha scritto le prime classi ognuno sul proprio branch ed infine abbiamo fatto un merge nel branch production dove poi abbiamo lavorato finché non abbiamo deciso di abbandonare libGDX e utilizzare javaFx, a questo punto abbiamo ricreato un branch vuoto col nuovo progetto in cui abbiamo spostato uno ad uno i file java(senza usare git, semplicemente copiando il vecchio progetto, per una serie di problemi a rimuovere libGDX) , dopodichè abbiamo pushato in un unico commit il nuovo progetto con javaFX nel branch javaFXonly.

C.0.7 Redazione

Per la relazione abbiamo preso fin dall'inizio spunto dal progetto OOP di PACMAN(https://bitbucket.org/danysk/oop19-barzi-eddie-belloni-sofia-rossi-davide-vissani-filippo/src/master/).

Commenti

Per quanto riguarda l'origine delle idee o soluzioni adottate non sappiamo nemmeno noi come poter rispondere alle sue domande, molte cose si sono presentate come problemi inaspettati e risolti pian piano, tramite ricerche veloci (ad esempio stackOverflow).