

Relazione progetto per “Programmazione a Oggetti”

Mario Biavati
Michele Ravaioli
Manuel Tartagni
Francesco Valentini

25 aprile 2022

Indice

1	Analisi	
1.1	Requisiti	2
1.2	Analisi e modello del dominio	3
2	Design	
2.1	Architettura	4
2.2	Design dettagliato	5
3	Sviluppo	
3.1	Testing automatizzato	9
3.2	Metodologia di lavoro	10
3.3	Note di sviluppo	12
4	Commenti finali	
4.1	Autovalutazione	13

Capitolo 1

Analisi

Il progetto scelto dai membri del gruppo consiste nella realizzazione di un videogioco partendo dalla costruzione di un game engine. Il concept e la meccanica di questa applicazione sono ispirati al gioco per mobile “No Humanity”, che abbiamo osservato e studiato per capire come strutturare dalle basi il progetto. Abbiamo ideato quindi un gioco stile “bullet-hell” dove il giocatore deve comandare un palloncino finché non scoppia. L’obiettivo è resistere più a lungo possibile mentre si è continuamente bombardati con oggetti pericolosi. L’applicazione è stata chiamata “Don’t Pop” per far intendere immediatamente ai giocatori lo scopo del gioco.

1.1 Requisiti

Requisiti funzionali

- L’applicazione dovrà avere tre componenti principali, facilmente intercambiabili tra loro: il menù di gioco, la scena di gioco e la classifica dei punteggi. L’utente potrà interagire con ogni scena per eseguire azioni, come per esempio cambiare scena.
- Durante l’esecuzione dell’applicazione dovrà essere visibile solo una scena per volta, quindi o menù, o scena di gioco, o classifica.
- Il menù dovrà essere la scena principale, quella che si mostra all’ avvio del videogioco. Dovrà permettere all’ utente di poter cambiare il proprio nome utente e di avviare il gioco vero e proprio.
- La scena di gioco dovrà avere al suo interno l’oggetto che il giocatore manovra (in questo caso dovrà essere un palloncino) e l’indicatore del punteggio del giocatore. Col passare del tempo il punteggio aumenterà e compariranno i “nemici”, ovvero gli oggetti che se toccati col palloncino lo faranno scoppiare. Il gioco deve terminare quando si tocca un nemico col palloncino.
- Ci saranno quattro tipi di nemici, divisi per comportamento: proiettili, laser, palle spinate ed esplosioni. I proiettili proseguono con moto

rettilineo a velocità costante lungo tutta la scena di gioco. I laser compaiono con posizione casuale nella scena e dopo aver atteso un tempo predefinito, si attivano e diventano pericolosi (quando viene creato non è nocivo, mostra solamente in che punto comparirà); Le palle spinate si muovono con moto parabolico e vengono lanciate dai lati della scena; Le esplosioni, come i laser, compaiono nella scena casualmente e detonano dopo un tempo predefinito, coprendo un'area maggiore dei laser.

- Durante il gioco compariranno dei potenziamenti che se toccati col palloncino daranno vari effetti al giocatore.
- La classifica si dovrà aprire quando il gioco termina e dovrà mostrare il punteggio appena ottenuto del giocatore corrente e la classifica dei giocatori appena aggiornata. Dovrà permettere all'utente di poter tornare al menù principale oppure di rigiocare.

Requisiti non funzionali

- Il giocatore dovrà essere in grado di controllare il palloncino con rapidità e precisione, così da poter schivare i nemici. La scena di gioco sarà quindi molto interattiva.

1.2 Analisi e modello del dominio

L'applicazione deve essere in grado di cambiare la scena attiva su comando dell'utente, e potrà cambiare solamente in tre scene diverse: menù, gioco e classifica. La parte più problematica da progettare è quella del gioco. Ogni videogioco per poter funzionare ha bisogno di un *game loop*, un ciclo ripetuto fino alla conclusione che permette di far progredire lo stato di gioco. Nel ciclo il game engine dovrà eseguire per ogni oggetto di gioco: l'aggiornamento dello stato dell'oggetto, a seconda del comportamento che l'oggetto stesso dovrà seguire; Un controllo delle collisioni col giocatore, e agire di conseguenza se ne rileva una; Aggiornamento della grafica dell'oggetto nella scena; Calcolo del punteggio totale.

La meccanica interna di gioco verrà delegata alla classe con il compito di eseguire un ciclo che aggiorna lo stato e la creazione dei nemici e del giocatore, ovvero la loro posizione e le eventuali collisioni col giocatore.

Quest'ultima avrà anche il compito di gestire le factory riguardanti nemici e powerup in maniera da essere sincronizzata con il tempo di gioco.

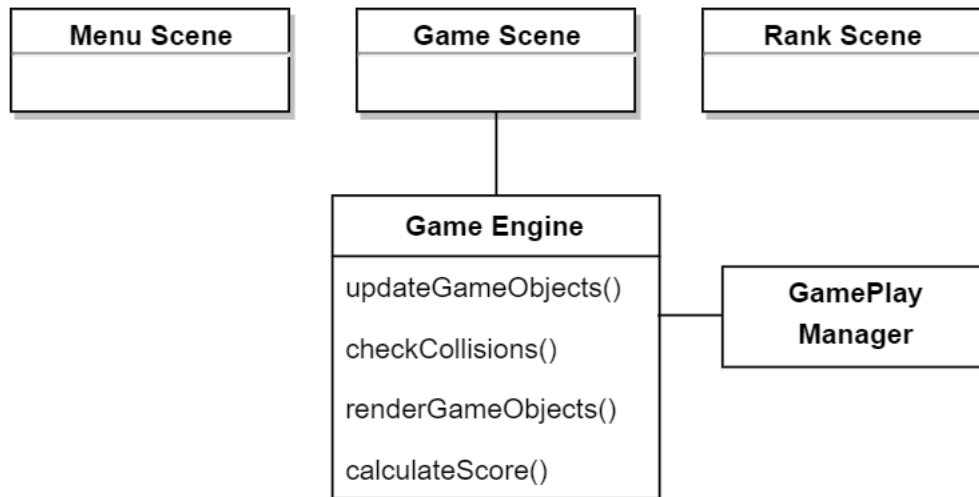


Fig. 1.1 : Design proposto per l' applicazione

Capitolo 2

Design

Il design di questo progetto si propone di sfruttare differenti design pattern e i principali concetti di programmazione a oggetti.

2.1 Architettura

Per avere un cambio di scena semplice e rapido, è stata progettata la classe `GameApplication`. Questa classe offrirà dei metodi pubblici per poter cambiare con facilità la scena corrente, senza però avere il controllo dello stato dell'applicazione. Il compito di cambiare scena non spetterà a questa classe, ma alle altre classi che gestiscono le singole scene, ovvero i gestori del menù, del gioco e della classifica, che modificano/gestiscono le scene a loro connesse. Facendo in questo modo, ogni gestore deciderà da sé quando passare alla prossima scena, inoltre risulta più semplice aggiungere scene nuove.

L'architettura della scena di gioco è costruita tenendo separata la parte di model dalla parte di grafica. L'unica interazione presente tra grafica e model si trova nel GameEngine, all'interno del *game loop*, il quale delega la rappresentazione degli oggetti ai componenti Renderer dell'oggetto stesso, utilizzando il *pattern Strategy* per implementare le diverse modalità di rappresentazione grafica. Lo stesso pattern è usato per il controllo delle collisioni con il player, usando l'interfaccia Collider.

All'interno del *game loop* dovranno essere eseguiti oltre all'aggiornamento degli oggetti di gioco presenti anche il ciclo di spawn dei nemici e il calcolo del punteggio, che vengono delegati a classi apposite.

Il ciclo di spawn verrà gestito dalla classe SpawnManager che delegherà la creazione dei singoli nemici e power up alle relative factory, seguendo il *pattern Builder*.

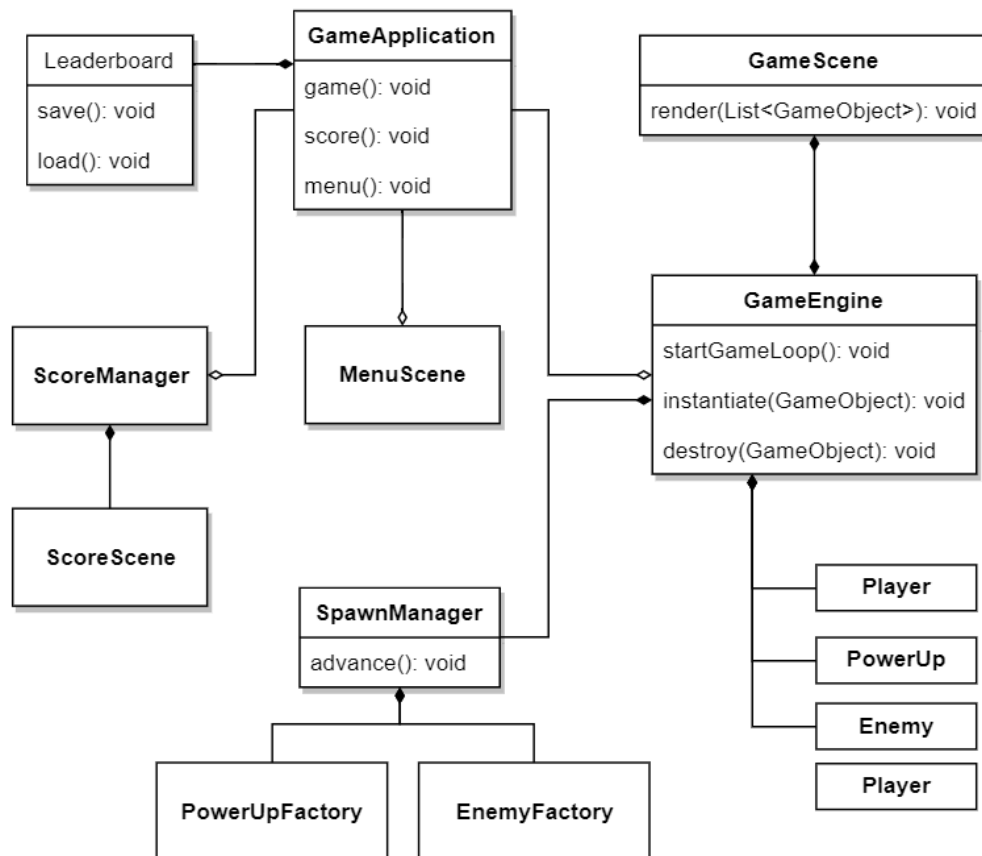


Fig. 2.1 : Schema generale dell'architettura usata

2.2 Design dettagliato

Michele Ravaioli:

- **Diverse modalità di collisione**

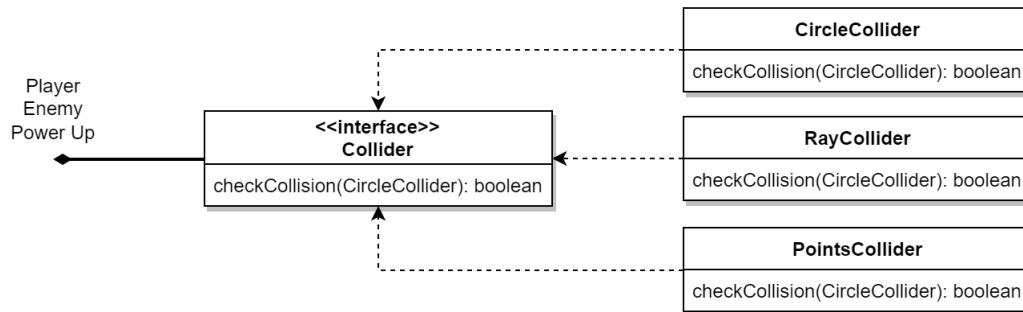


Fig. 2.2 : Piccolo schema che mostra la soluzione adottata riguardo alle collisioni.

Problema: Gli oggetti di gioco dovranno avere un modo di calcolare le proprie collisioni, tuttavia per molti oggetti queste modalità sono simili e quindi ripetute. Inoltre dovranno essere potenzialmente intercambiabili (se un nemico passa da quadrato a cerchio il metodo di collisione cambia).

Soluzione: Il sistema della gestione delle collisioni utilizza la *pattern Strategy*, come da figura (sopra). Il player, i nemici e i power up avranno associato un proprio Collider, che può essere implementato in più modi differenziandosi nel tipo di calcolo da eseguire per il controllo delle collisioni. Collider è implementato come `CircleCollider` (calcola come se fosse un cerchio), `RayCollider` (come se fosse una retta) e `PointsCollider` (come un insieme di punti nello spazio). Tutti i calcoli delle collisioni partono dal presupposto che il player sia composto da un `CircleCollider`.

- **Diverse modalità di renderizzazione degli oggetti**

Problema: Tutti gli oggetti di gioco dovranno essere renderizzati sulla scena di gioco senza ripetizioni di codice e in maniera diversa tra loro. Inoltre dovranno poter cambiare renderizzazione durante l'esecuzione del gioco.

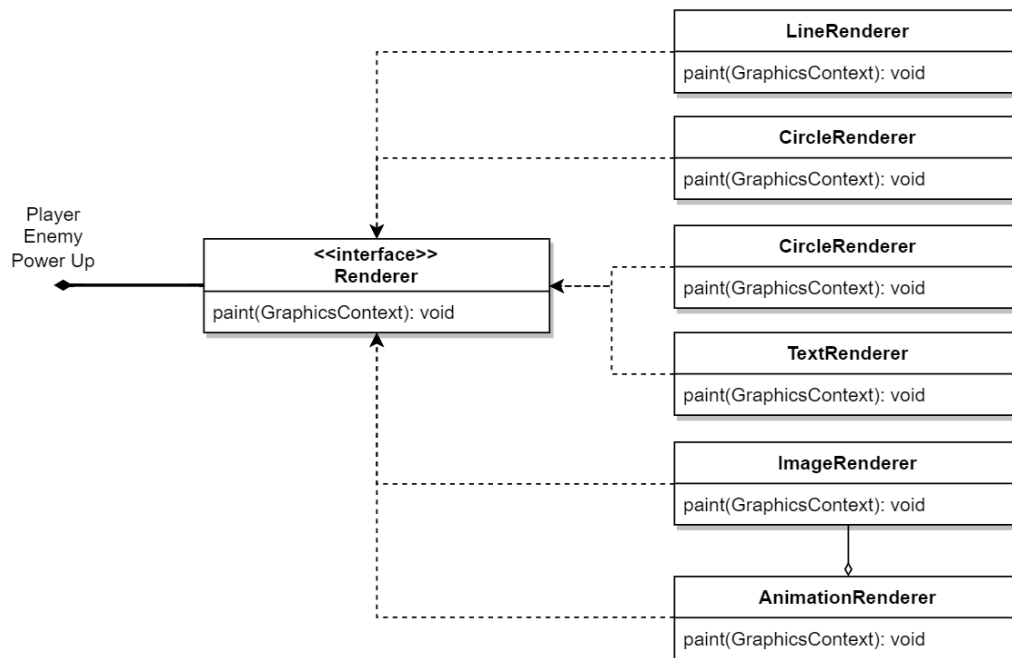


Fig. 2.3 : Piccolo schema che mostra la soluzione adottata riguardo alla renderizzazione degli oggetti di gioco.

Soluzione: Come per le collisioni, si è utilizzato il *pattern Strategy*. Si è creata quindi un'interfaccia *Renderer* che in base all'implementazione scelta dall'oggetto di gioco possa mostrare immagini, linee, cerchi oppure testi.

- **Ripetizione di codice in player, nemici e potenziamenti**

Problema: Sia player, sia nemici sia potenziamenti presentano caratteristiche comuni, infatti tutti e tre hanno una posizione nello spazio, un Collider, un *Renderer* e un metodo *update()* per aggiornare il proprio stato di gioco.

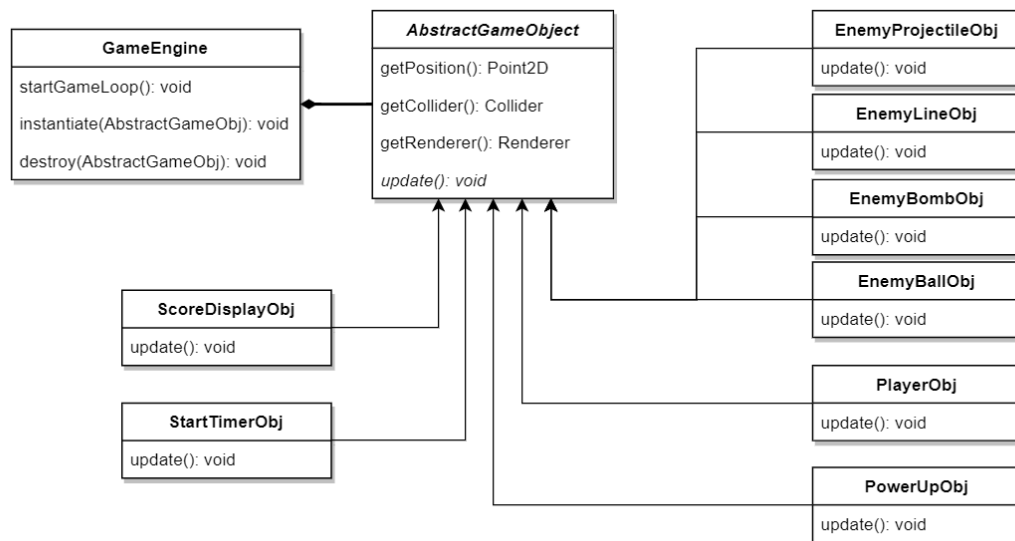


Fig. 2.4 : Piccolo schema che mostra la soluzione adottata riguardo agli oggetti di gioco.

Soluzione: Per evitare la ripetizione del codice si è utilizzata una classe astratta **AbstractGameObject** che contiene tutte le caratteristiche comuni degli oggetti di gioco (contiene i metodi getter e setter). **AbstractGameObject** ha un metodo astratto `update()` da implementare nelle classi che la estendono, così da poter avere oggetti con comportamenti differenti. Il **GameEngine** userà quindi non le singole implementazioni dei nemici, player e power up, ma degli **AbstractGameObject** i quali verranno aggiornati, controllati e renderizzati dentro il *game loop*.

Manuel Tartagni:

- Game Engine come god-class

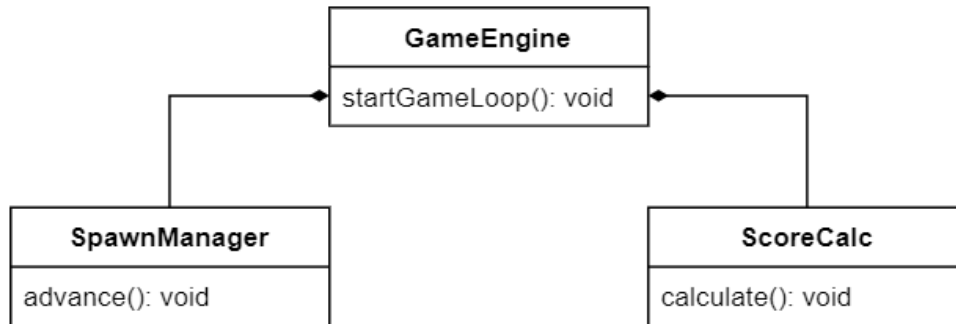


Fig. 2.5 : Piccolo schema che mostra le classi addette al gameplay.

Problema: Poiché deve eseguire aggiornamenti, controlli collisioni, rendering, calcolo punteggio e spawn dei nemici, il codice del **GameEngine** risulta lungo e poco leggibile.

Soluzione: Il problema della creazione viene delegato allo **SpawnManager** per la creazione randomica dei potenziamenti e temporizzata per i nemici secondo i timer associati ad ogni categoria di nemico. Anche il calcolo del punteggio viene assegnato a un'altra classe **ScoreCalc**, la quale calcola basandosi sul tempo di gioco trascorso ed effettuando eventuali operazioni sul punteggio dovute ai power up.

- Creazione dei nemici

Problema: I costruttori dei nemici sono complessi e ricchi di parametri, pertanto il codice della gestione dello spawn risulta poco leggibile e molto complesso.

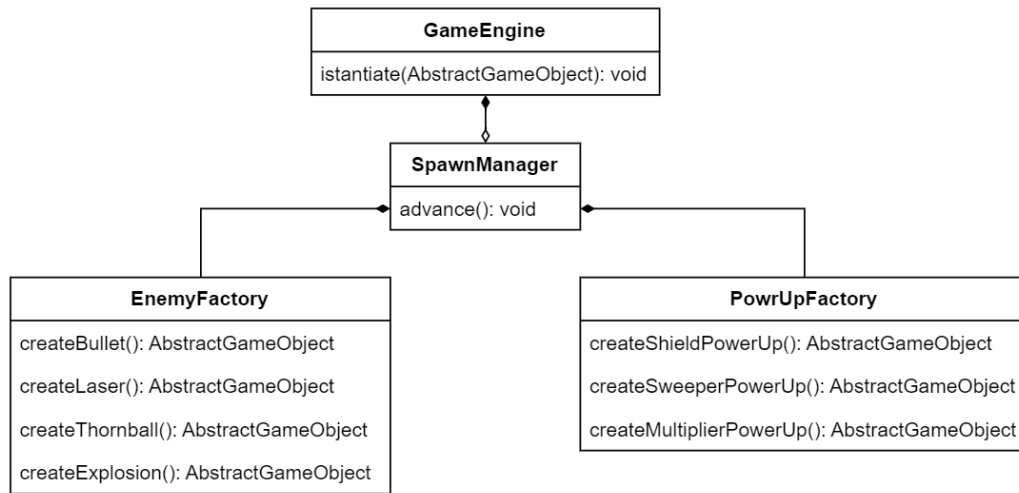


Fig. 2.6 : Piccolo schema che mostra i builder utilizzati.

Soluzione: Seguendo il *pattern builder* lo SpawnManager crea tramite le due factory (EnemyFactory e PowerUpFactory), come nella figura (sopra). I potenziamenti vengono creati in maniera randomica mentre i nemici seguiranno dei tempi stabiliti dai rispettivi timer segnati nello SpawnManager, differenti per ogni tipologia di nemici. Lo SpawnManager dovrà solamente istanziare nel GameEngine gli oggetti creati con le factory secondo i timer.

Francesco Valentini

- Gestione del coordinamento tra i vari oggetti

Problema: Gli oggetti di gioco devono potersi coordinare tra loro, scambiandosi messaggi per ottenere informazioni e/o impartire ordini ad altri oggetti. Inoltre, bisogna gestire le conseguenze della collisione tra il player e i nemici, garantendo il corretto funzionamento del gameplay.

Soluzione: La classe GameEngine si occupa del coordinamento tra gli oggetti di gioco, eseguendo gli stessi controlli in un ciclo che si ripete 60 volte al secondo (60Hz):

1. Aggiorna (incrementa) il tempo di gioco
2. Chiama lo SpawnManager per fare spawnare i nemici
3. Resetta la destroyQueue, rimuovendone tutti gli oggetti
4. Aggiorna tutti gli oggetti nella scena di gioco
5. Controlla se è avvenuta una collisione con un powerup
6. Controlla se è avvenuta una collisione con un nemico (se affermativo, fa terminare il gioco)
7. Renderizza tutti i cambiamenti (powerup ottenuto, morte del player, spostamento degli oggetti sullo schermo)
8. Per ridurre il carico sulla CPU, una volta finita la computazione, ordina al Thread corrente di fare la *sleep* per il tempo rimanente del frame (1 Frame = 1/60 s)
9. Calcola la durata del frame: questo calcolo è particolarmente utile agli oggetti e ai renderer nel caso si verificano dei *lag*. In questo modo il gioco eviterà di rallentare e procederà a ritmo normale, eventualmente saltando dei frame

Oltre al game loop appena descritto, il GameEngine si occupa anche di:

- Cambiare l'immagine del palloncino in base ai powerup ottenuti
- Riprodurre la musica di sottofondo
- Riprodurre dei suoni quando viene attivato un powerup
- Invocare la GameApplication al gameover per visualizzare la leaderboard
- Aggiungere nemici e powerup da istanziare in due rispettive liste, i cui elementi verranno istanziati dalle Factory apposite e renderizzati dalla GameScene
- Distruggere nemici e powerup, aggiungendoli alla destroyQueue. Questi oggetti verranno distrutti nel frame seguente.
I nemici possono essere distrutti solo utilizzando il powerup Sweeper,

mentre i powerup vengono sempre distrutti dopo essere stati utilizzati, o dopo 5 secondi se non vengono mai presi.

- Applicare un powerup, gestendone gli effetti
- Terminare il gioco, al game over

Il GameEngine in questo modo deve dipendere solo dalla GameApplication, la quale ne creerà una nuova istanza ad ogni nuova partita.

Il GameEngine implementa il pattern Observer, in quanto controlla ad ogni ciclo se è avvenuta una collisione tra player e nemici e, se affermativo, ordina al player di morire, interrompe il game loop e invoca la GameApplication per mostrare la ScoreScene col punteggio corrente (oltre alla classifica).

Allo stesso modo, implementa nuovamente il pattern Observer, perché controlla ad ogni ciclo se è avvenuta una collisione tra player e powerups e, se affermativo, applica il powerup ordina all'AudioManager di riprodurre il suono corrispondente al powerup attivato. In particolare, se il powerup attivato è lo *shield* o il *multiplier*, ordina al player di impostare l'immagine corrispondente, mentre se il powerup attivato è lo *sweeper*, svuota la lista dei nemici - e al ciclo seguente ordinerà, come di consueto, alla GameScene di renderizzare questi cambiamenti.

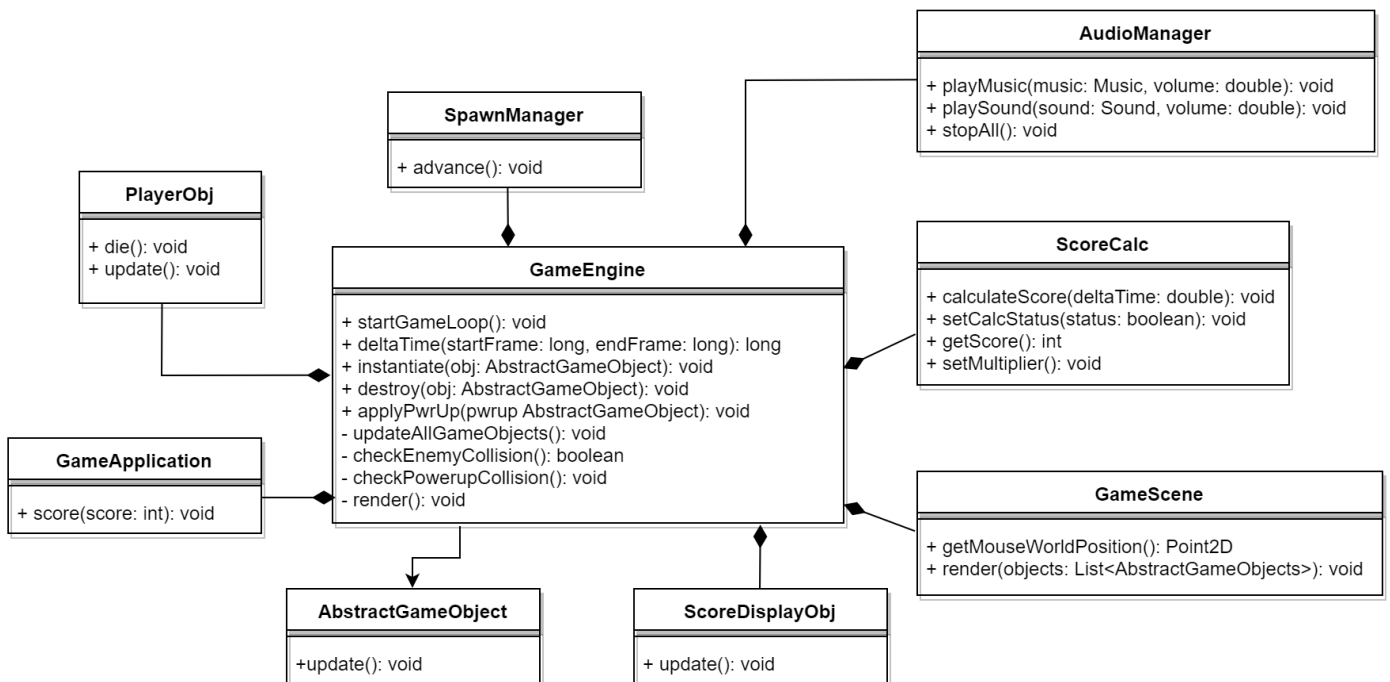


Fig. 2.7 : Schema UML che mostra le classi controllate direttamente dal GameEngine.

- Calcolo del punteggio

Problema: Il GameEngine deve calcolare il punteggio corrente ad ogni frame, incrementandolo a un ritmo base di 15 punti al secondo. Nel far ciò bisogna tenere conto anche di eventuali *powerup Multiplier*, i quali raddoppiano l'incremento del punteggio a 30 punti al secondo.

Tuttavia, se il GameEngine dovesse occuparsi di questo calcolo, si violerebbe il principio di singola responsabilità per ogni classe, oltre a renderlo ancora più complesso di così.

Soluzione: Creiamo una nuova classe, chiamata ScoreCalc, che si occupa di:

- Incrementare il punteggio ogni 4 frame ($60 / 15 = 4$)
- Gestire il conteggio dei frame prima di incrementare il punteggio
- Impostare il valore e la durata del moltiplicatore (5 secondi)
- Decrementare la durata rimanente del moltiplicatore ad ogni frame
- Bloccare il conteggio del punteggio prima dell'inizio del gioco (nei 3 secondi di countdown) e dopo il game over

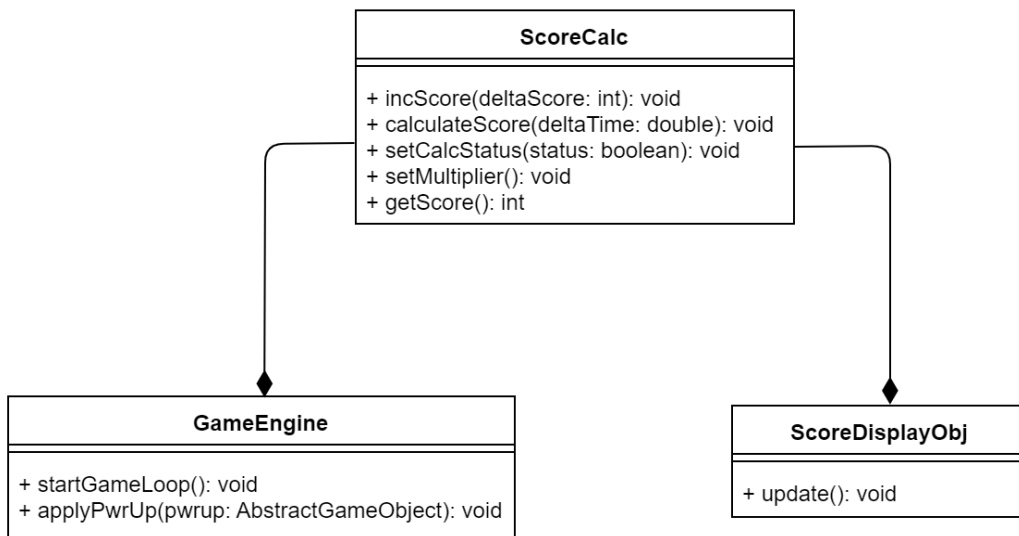


Fig.2.8 : Schema UML che mostra le classi che utilizzano la ScoreCalc.

- Visualizzazione scene di gioco +
Controllo della logica all'interno della GUI

Problema: Dobbiamo avere una classe che mostri un'interfaccia grafica, occupandosi anche dell'aspetto grafico, e che contemporaneamente controlli anche il comportamento che deve avere all'interazione con l'utente.

Chiaramente, tutto ciò non è realizzabile in una sola classe, violerebbe pesantemente il vincolo di singola responsabilità della classe e renderebbe il codice risultante estremamente complesso da comprendere e mantenere.

Soluzione: Innanzitutto abbiamo suddiviso le classi in base al tipo di scena che mostravano, creando così le classi MenuScene e ScoreScene. Tuttavia, il codice JavaFX necessario nelle singole classi sarebbe tuttora eccessivamente caotico, dato che andremmo a mischiare aspetti di visualizzazione della GUI con quelli relativi al suo comportamento.

Quindi, abbiamo deciso di separarli in più file (non necessariamente classi, data la natura di JavaFX), applicando un pattern pseudo-MVC:

- **View:** Viene definita strutturalmente nei file FXML e rifinita graficamente nei file CSS. Il file FXML viene caricato dalla classe ScoreScene.
- **Controller:** Le classi MenuScene e ScoreScene fungono da Controller, visto che gestiscono l'intero funzionamento della GUI di loro competenza: caricano la struttura dal file FXML, creano il FXML Controller e applicano l'aspetto descritto nel file CSS.
Anche le classi MenuSceneController e ScoreSceneController sono dei Controller: si occupano di specificare ciò che deve succedere quando l'utente interagisce con la GUI: quando viene cliccato un bottone, vanno a chiamare le funzioni corrispondenti della GameApplication.
- **Model:** La ScoreSceneController si salva i dati relativi alla classifica corrente, richiedendone le informazioni allo ScoreManager (il quale funge da classe Model), che a sua volta interrogherà la Leaderboard per *fetchare* i dati.
La ScoreSceneController ha così una certa parte di Model, rendendola una classe ibrida Controller-Model. Le motivazioni dietro questa scelta sono spiegate nel punto seguente.

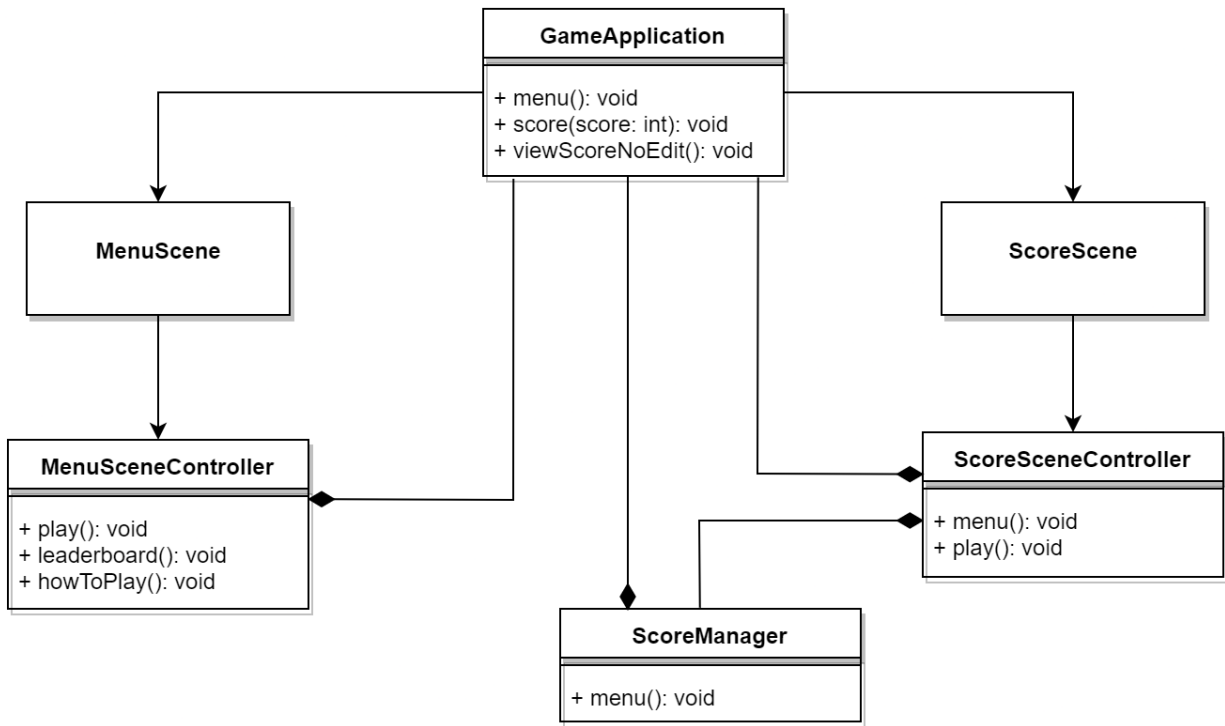


Fig 2.9 : Schema UML che mostra l'organizzazione delle classi di interfaccia grafica.

- Caricamento dei dati della classifica dei punteggi nella tabella

Problema: La ScoreSceneController deve caricare i dati della classifica dei punteggi, manipolandoli in modo tale da poterli inserire agevolmente in una tabella.

Tuttavia, la Leaderboard, per motivi di semplicità, salva su file solo le informazioni relative al nome del player e il punteggio che ha ottenuto, evitando di specificare informazioni superflue come la posizione in classifica del player (player rank) - parametro che, se specificato, costringerebbe la Leaderboard a modificare la lista dei record dopo ogni game over, aggiornando tutti i player rank.

Lo ScoreManager, invece, si occupa solo di gestire il salvataggio e il caricamento da file. Se gli dessimo il compito ulteriore di manipolare i dati della Leaderboard per adattarli meglio a una tabella, gli staremmo dando due compiti diversi, contravvenendo al SRP.

Soluzione: Abbiamo integrato questa funzionalità direttamente nella ScoreSceneController, la quale converte i dati della Leaderboard, sotto forma di Pair, in dei RankItem (nuova classe creata ad-hoc). Questi ultimi contengono il nome e il punteggio del player (come i Pair della Leaderboard e

dello ScoreManager), ma aggiungono anche il rank del player corrente. In questo modo possiamo inserire i dati nella tabella in un colpo solo (sfruttando l'auto match delle colonne di JavaFX), semplificando il codice. Se li avessimo aggiunti riga per riga, avremmo dovuto chiamare ripetutamente lo ScoreManager per ottenere i dati della corrispondente riga della tabella e avremmo avuto un costo computazionale maggiore.

Abbiamo deciso di delegare l'interrogazione della Leaderboard allo ScoreManager per non dare alla ScoreSceneController il compito di controllare se la Leaderboard vada aperta in modalità *read-only* o *read-write*. In questo modo, oltre ad aver reso il codice più semplice, abbiamo applicato il pattern Proxy, in quanto la ScoreSceneController non effettua direttamente l'accesso ai dati contenuti nella Leaderboard, ma li ottiene tramite lo ScoreManager.

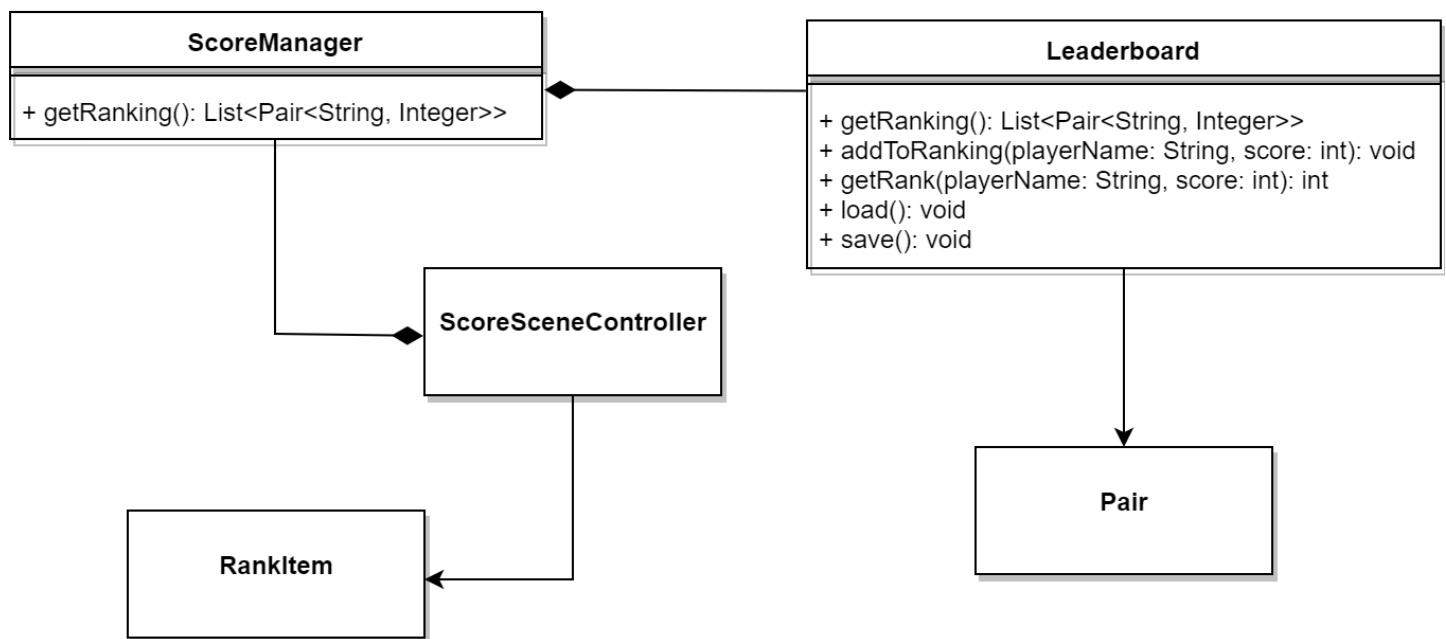


Fig. 2.10 : Schema UML che mostra il funzionamento dell'accesso ai dati della classifica da parte della ScoreSceneController.

Mario Biavati

Problema: I nemici devono avere dei comportamenti sia individuali che universali tra tutti gli oggetti.

Soluzione: L'implementazione dei vari nemici segue il pattern *Template* in cui la classe astratta `AbstractGameObject` definisce alcuni metodi e parametri comuni a tutti gli oggetti, come la posizione, la grafica e le collisioni. Il resto del codice è unico per tipo di nemico, e deve quindi essere individualmente per ogni nemico. Esempi di questo tipo di codice sono il tipo di movimento (se necessario), il comportamento modificabile nel tempo tramite parametri dei propri componenti, l'interazione con elementi statici del gioco (punteggio, giocatore, tempo trascorso), ecc...

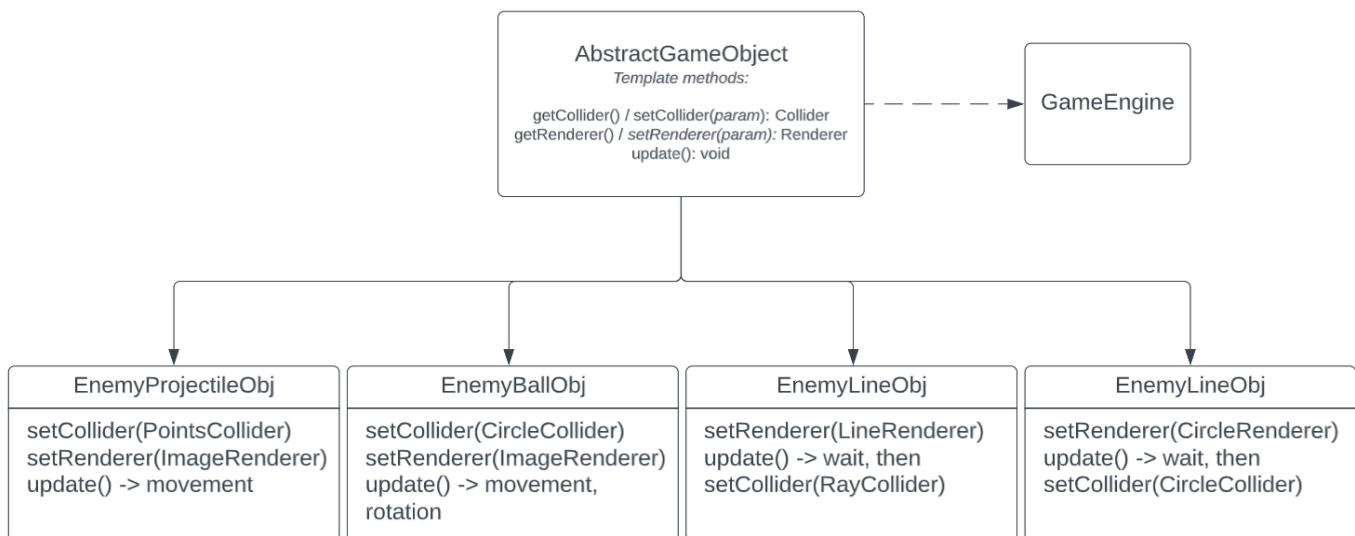


Fig. 2.11 : Schema che rappresenta le implementazioni dei nemici

Problema: I powerup usano la stessa classe, ma necessitano funzioni e immagini diverse a seconda del tipo assegnato.

Soluzione: Usando il pattern *Strategy* l'oggetto `PowerUpObj` decide quale powerup diventare in base ai parametri a esso passati da `PowerupFactory`. Il codice legato all'attivazione ed effetto del powerup è gestito dalla *god class* `GameEngine` e dal `PlayerObj`, in base a che powerup è stato attivato.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Il test delle varie funzionalità è stato eseguito utilizzando la libreria JUnit5 di Java, in modo tale da avere un test automatico delle classi. Seppur siano test minimali e semplici, sono importanti per capire il funzionamento dei vari componenti dell'applicazione.

Sono stati eseguiti test per le collisioni, dove si provano le varie implementazioni di Collider, così da vedere se la collision detection è calcolata correttamente sia per i cerchi sia per le rette.

Ulteriori test sono stati fatti per il salvataggio dei punteggi dei giocatori, per verificare il corretto funzionamento della classifica.

3.2 Metodologia di lavoro

Si è cercato di suddividere il carico di lavoro tra gli sviluppatori del progetto in maniera bilanciata, così da non avere membri del gruppo che hanno programmato più degli altri. Anche la parte grafica del progetto è stata divisa equamente. Il lavoro è stato suddiviso nella seguente maniera:

Michele Ravaioli - gli è stato assegnato il compito di programmare i Collider e le varie implementazioni delle collision detection (package game.collider); I Renderer e le rispettive implementazioni (package game.renderer); la classe AbstractGameObject (package game.model); La GameScene come parte di grafica (package game.ui). Nel package game.util ha creato la classe Point2D e ha aiutato nello sviluppo delle classi GameEngine, AudioManager, SpawnManager (game.engine), Leaderboard e ScoreCalc(game.util).

Manuel Tartagni - gli è stato assegnato il compito di implementare le classi delle factory dei nemici (EnemyFactory , package: game.engine) e dei potenziamenti (PowerUpFactory , package: game.engine) e le relative

implementazioni. Sviluppo della classe WhereToSpawn più (game.util) e RandomInt (game.util) utilizzata per la scelta del lato spawn. Nel Package game.engine ho implementato lo SpawnManager e il GameApplication (funge da controller). Classe grafica HowToPlay e HowToPlayController (entrambe nel package game.ui).

Francesco Valentini – Classi assegnate:

- **GameEngine (game.engine)**
Questa classe è il *core* del videogioco: gestisce il coordinamento tra i vari componenti, impartendo a ciascun oggetto il compito da svolgere. In particolare, ordina alla GameScene di fare il *refresh* della scena di gioco 60 volte al secondo (60Hz).
- **ScoreManager (game.engine)**
Gestisce il caricamento dei dati nella ScoreScene: invoca la Leaderboard, fa il *fetch* dei dati, li salva in un array e infine li mette al posto giusto dentro la tabella della ScoreScene.
- **MenuScene (game.ui) + controller, fxml, css**
Mostra il menù principale del gioco, permettendo al giocatore di iniziare il gioco, visualizzare la classifica corrente, o vedere le regole di gioco in una schermata apposita.
- **ScoreScene (game.ui) + controller, fxml, css**
Mostra la classifica dei 50 giocatori migliori in una tabella, scartando quelli con i punteggi più bassi. Se questa classe viene invocata dopo il gameover, mostra anche il punteggio del *player* corrente.
- **Pair (game.util)**
Utility class che memorizza coppie di valori `<String, Integer>`.
- **RankItem (game.util)**
Memorizza tutti i dati relativi a un giocatore: posizione in classifica, nome e punteggio ottenuto.
Il controller della ScoreScene converte i dati ottenuti dalla Leaderboard da un array di Pair a un array di RankItem: mentre scorre la classifica della Leaderboard, si salva la sua posizione nel campo apposito di RankItem. In questo modo, la PropertyValueFactory (nel file menuscene.fxml) deve semplicemente fare la *fetch* dei dati contenuti in questo array, invece di scorrere la lista nella Leaderboard ad ogni iterazione, diminuendo la complessità computazionale da $O(n^2)$ a $O(n)$.

- **ScoreCalc (game.util)**
Calcola e memorizza il punteggio corrente, tenendo conto di eventuali moltiplicatori.

Mario Biavati - Gli è stata assegnata l'implementazione delle classi del model, quindi nemici (EnemyProjectileObj, EnemyLineObj, EnemyBallObj, EnemyBombObj), powerup (PowerUpObj), grafica del timer (StartTimer), grafica del punteggio (ScoreDisplay) e la classe del giocatore (PlayerObj). Ha contribuito anche ad aggiungere effetti sonori per eventi come ottenere powerup o essere colpiti, e ha contribuito alla classe menuScene.

Utilizzando il DVCS Git, abbiamo lavorato sul progetto contemporaneamente, evitando di sovrapporci. Ognuno lavorava sui package assegnati, senza creare conflitti con altri file.

Il progetto è stato sviluppato usando l'IDE Eclipse, come progetto di JavaFX. Per poter esportare l'applicazione in un jar cross-platform, siamo infine passati a un progetto Maven, così da usare il plugin per il shadow jar.

3.3 Note di sviluppo

Per il progetto è stata utilizzata la libreria JavaFX per la grafica, scelta per motivi di performance e praticità.

Michele Ravaioli:

- Uso di lambda expressions nei cicli for each del GameEngine;
- Uso di stream per il ciclo di controllo delle collisioni nel GameEngine;
- Uso di funzioni di libreria di JavaFX per le implementazioni del metodo render(GraphicsContext) dell'interfaccia Renderer;
- Uso di lambda expressions Runnable per l'esecuzione degli eventi nella classe ScoreCalc (onMultiplierStart e onMultiplierEnd);

Manuel Tartagni:

- Uso di funzioni di libreria di JavaFX per le implementazioni delle classi HowToPlay e del rispettivo controller HowToPlayController.
- Utilizzo di JavaFX Scene Builder per la creazione della scena grafica e per la creazione del file .fxml
- Utilizzo di file Fxml per creare la scena e associare al bottone presente l'azione corrispondente.

Francesco Valentini:

- Utilizzo di lambda e stream per effettuare iterazioni sulle liste del GameEngine
- Utilizzo di Scene Builder per velocizzare la creazione di una scena grafica in FXML
- Utilizzo di file CSS per definire l'aspetto della GUI in modo dettagliato

Mario Biavati:

- Uso di funzioni di libreria di JavaFX per renderer e MenuScene
- Utilizzo di SceneBuilder per MenuScene
- Utilizzo di lambda expressions in ScoreDisplayObj

Capitolo 4

Commenti finali

Grazie al tempo speso nella stesura dello schema finale del progetto, comprendente tutte le varie classi, abbiamo decisamente semplificato la programmazione successiva, dato che avevamo già una traccia da seguire. Per tutti e quattro, la parte decisamente più complessa è stato imparare a costruire l'interfaccia grafica, in quanto eravamo principianti nell'uso della libreria JavaFX.

4.1 Autovalutazione

Michele Ravaioli: Ritengo di aver eseguito un buon lavoro, soprattutto nella parte di analisi e di design. Mi sono impegnato molto nel cercare la migliore progettazione possibile per il game engine e per la meccanica di gioco, facendo molte prove per conto mio. Complessivamente sono soddisfatto del lavoro svolto, infatti molte parti del progetto possono essere riusate per progetti futuri (es: il game engine, collisioni, renderer, cambio scene etc...). Inoltre ho imparato a usare bene la libreria di JavaFX, che utilizzerò molto d'ora in poi.

Manuel Tartagni: Ritengo di aver lavorato bene e sono soddisfatto del lavoro svolto in quanto ha coperto non solo la parte di model ma anche quella grafica. Ritengo utile la parte grafica in quanto mi ha permesso di utilizzare strumenti come il java fx Scene Builder e i file di formato xml.

Francesco Valentini: Sono decisamente contento del risultato che abbiamo ottenuto: nonostante tutte le varie difficoltà che abbiamo incontrato, tipiche di ogni progetto di questo tipo, siamo riusciti a realizzare un videogioco a mio parere carino e molto giocabile. Vedere la nostra idea diventare realtà mi ha dato un grande senso di soddisfazione.

Personalmente, ritengo di aver svolto bene il mio lavoro, mettendoci della dedizione, soprattutto nella parte di realizzazione dell'interfaccia grafica con JavaFX. È stato lì che ho imparato a usare lo Scene Builder per velocizzare la scrittura di codice FXML, che ho poi sempre revisionato, e a integrare dei file CSS per definire nei dettagli la GUI.

La scrittura del GameEngine è stata molto impegnativa, soprattutto perché è stata la prima volta in cui mi sono cimentato nella creazione di un videogioco, ma sono sicuro di aver imparato molte abilità che mi potranno servire in futuro.

Mario Biavati - Mi ritengo soddisfatto del lavoro svolto sul model, e delle soluzioni che ho implementato per i diversi tipi di nemici, per il giocatore e per gli elementi di grafica in gioco. Ho imparato molto dal progetto: come usare JavaFx, come usare SceneBuilder, come usare e modificare i file FXML e CSS, e come usare Git e GitHub in modo spedito ed efficace. L'unico rimpianto è che la parte a me assegnata si è rivelata meno impegnativa del previsto. Ho quindi cercato di assistere in altri ambiti al meglio delle mie abilità, ma comunque avrei voluto contribuire di più al progetto.

Appendice A

Per avviare l'applicazione è necessario lanciare il comando:

```
java -jar download/dont-pop.jar
```

Per aprire il progetto su Eclipse è necessario:

1. Clonare il repository;
2. Importare il progetto facendo *Import -> Existing Project into Workspace*.

Per avviare il progetto su eclipse è necessario:

1. Creare una nuova Maven Build configuration;
2. Inserire nel campo base directory `${workspace_loc:/Dont-Pop}`
3. Inserire nel campo goals `clean javafx:run`
4. Cliccare su Run.

Per avviare i test automatici è necessario:

5. Creare una nuova Maven Build configuration;
6. Inserire nel campo base directory `${workspace_loc:/Dont-Pop}`
7. Inserire nel campo goals `clean javafx:run`
8. Cliccare su Run.